COMP1040: The Craft of Computing

Stephen Gould
stephen.gould@anu.edu.au

Mark Reid
mark.reid@anu.edu.au

October 11, 2016

Copyright © 2015–2016, Stephen Gould and Mark Reid. All rights reserved.

Contents

1 Overview **1**
1.1 Summary . . . 1
1.2 Organisation and Principles . . . 2

2 Assessment and Practice **5**
2.1 Assessment Policy . . . 5
2.2 Weekly Exercises . . . 6
2.3 Assignments . . . 7
2.4 Projects . . . 7

3 Lectures **9**
3.1 Lecture 1: Welcome to The Craft of Computing . . . 10
3.2 Lecture 2: Introduction to Coding . . . 16
3.3 Lecture 3: Software Development Basics—Tools and Evironments 25
3.4 Lecture 4: Data Structures I . . . 38
3.5 Lecture 5: Execution Models and Control Flow . . . 49
3.6 Lecture 6: Data Formats and Files . . . 60
3.7 Lecture 7: String Processing . . . 69
3.8 Lecture 8: Functions . . . 76
3.9 Lecture 9: Computer Architecture . . . 86
3.10 Lecture 10: Data Structures II . . . 95
3.11 Lecture 11: Objects and Classes . . . 106
3.12 Lecture 12: Tools and Practices . . . 116
3.13 Lecture 13: Reading Source Code . . . 123
3.14 Lecture 14: Libraries and APIs . . . 128
3.15 Lecture 15: Searching for Help . . . 134
3.16 Lecture 16: Advanced Revision Control . . . 139
3.17 Lecture 17: Visualising Data . . . 147
3.18 Lecture 18: Visualising Data II . . . 158
3.19 Lecture 19: Debugging Strategies . . . 166
3.20 Lecture 20: Software Design . . . 176
3.21 Lecture 21: Collaborating . . . 186
3.22 Lecture 22: Refactoring Code . . . 189
3.23 Lecture 23: Advanced Programming . . . 197
3.24 Lecture 24: Regular Expressions . . . 207
3.25 Lecture 25: Data Pipelines and Command Line Processing . . . 212
3.26 Lecture 26: Optimising Code . . . 220
3.27 Lecture 27: Defensive Programming . . . 230

3.28 Lecture 28: Programming Languages 235
3.29 Lecture 29: Distributing Software 240

4 Tools **245**
4.1 Python 245
4.2 PyCharm 246
4.3 GitLab 246
4.4 Configuring PyCharm 246

Chapter 1

Overview

1.1 Summary

Knowing how to effectively use computational tools to perform data analysis, simulation, and visualisation is a crucial skill in a number of industries and academic disciplines. The aim of this course is to provide skills for tackling the "messiness" of real-world computer systems, programming languages, and data. Unlike many other computer science courses which explain a single area in depth, the focus of The Craft of Computing is on understanding core principles that allow students to quickly and confidently learn and apply a variety of computational tools to several different types of problem. Furthermore, by the end of the course students will be able to readily adapt their know-how to new programming languages and software development tools. The skills developed in this course will be a great asset to students in their undergraduate life and future careers.

A companion course, The Art of Computing, teaches computational *thinking* in contrast to computational *doing* taught in this course.

1.1.1 Learning Outcomes

Students completing this class will:

- Appreciate the creative possibilities computation brings to multiple disciplines.
- Be able to solve practical problems in various domains using appropriate software tools.
- Be able to understand, modify, debug, and write small programs in a high-level programming language.
- Be able to translate learned programming skills to new programming languages, tools, and contexts.
- Understand the culture, conventions and resources to do with software development, deployment, and use.

1.1.2 What this course is not

This course is about *craft*, not science—it is about building and using tools, not analysing them. This course is not intended as a substitute for introductory computer science courses that systematically introduce students to a programming language so they will be able to write robust, performant code from scratch. Instead, we want to show students how to adapt existing code and/or write small amounts of simple data processing code that "glues" together existing libraries to solve a problem. Some of this philosophy is captured by the principles of post-modern programming.

This course is also not about programming, although programming is an important skill, it is only part of the picture. The environments, tools, existing libraries, and cultures around solving computational problems are of equal or greater importance. It is not necessary to understand every aspect of a programming langauge to be able to use it to solve a problem at hand.

1.1.3 Topics

The emphasis of this course of "getting things done with computers" is reflected in the following choice of topics:

- Introduction to computational environments and tools.
- Computer fundamentals and computational thinking.
- Learning programming languages and libraries.
- Data flow and visualisation.
- Collaborating and distributing code.

1.1.4 Programming Language

The course uses Python as the primary programming language for teaching. Students are not expected to have prior programming experience and an introduction to Python will be given during the first few weeks of the course. However, the course makes no attempt to be a comprehensive tour of all of Python's features and students will be expected to learn some aspects of the language on their own.

The principles learned in the course are applicable to many programming languages and environments. Students will have the option of applying the knowledge and skills developed in the course to other programming languages through two open-ended projects.

1.2 Organisation and Principles

1.2.1 Course Snapshot

The workload model for the course assumes that an average student spending 10 hours per week on the course should be capable of achieveing a Credit level grade. Contact hours include three 50 minute lectures and one laboratory session per week. In addition lectures and tutors will hold office hours and be available online for help with weekly exercises and assignments.

<table>
<tr><th>Wk</th><th>Lectures and Labs</th><th colspan="2">Assessments</th></tr>
<tr><td>1</td><td rowspan="4">Learning to program</td><td rowspan="7">Exercises
1 hr/wk
(10%)</td><td rowspan="3">Assignment 1
5 hr/wk (10%)</td></tr>
<tr><td>2</td></tr>
<tr><td>3</td></tr>
<tr><td>4</td><td rowspan="2">Assignment 2
5 hr/wk (20%)</td></tr>
<tr><td>5</td><td rowspan="3">Doing interesting things with data</td></tr>
<tr><td>6</td><td rowspan="2">Assignment 3
5 hr/wk (20%)</td></tr>
<tr><td>7</td></tr>
<tr><td colspan="4">Teaching Break (2 weeks)</td></tr>
<tr><td>8</td><td rowspan="6">Collaborating,
best practices,
and advanced topics

Beyond
COMP1040</td><td rowspan="6">Exercises
1 hr/wk</td><td rowspan="6">Project
6 hr/wk
(40%)</td></tr>
<tr><td>9</td></tr>
<tr><td>10</td></tr>
<tr><td>11</td></tr>
<tr><td>12</td></tr>
<tr><td>13</td></tr>
</table>

1.2.2 Vignettes and Exercises

The course will be delivered through a series of lectures broken into short vignettes that highlight the main aspects of the course topics. Course material will be reinforced through laboratory sessions and assessable weekly exercises, individual assignments, and group projects.

As much as practical, we will aim to present small exercises alongside each vignette to allow students to play with the ideas the vignette introduces. Additionally, online exercises that introduce common programming “gotchas” (e.g., bad variable scope, variable typos, off-by-one errors, etc.) so students gain experience fixing these common problems.

The exercises will be presented through an online, interactive programming environment (i.e., CodeBench) and shift towards working with code repositories, command-line tools, editors, IDEs, etc. as the course progresses.

1.2.3 Assignments as Real-World Projects

Since one of our main aims is to give students the skills and confidence to solve real-world problems with computing tools, the format of assignments will aim to closely match what it is like to work “in the wild”. That is, data will need to be found and possible cleaned and transformed; appropriate libraries will need to be found (and installed if necessary); submission will involve pushing code to a repository along with notes and a report; etc.

1.2.4 First Lecture

We want the first lecture to be primarily a demonstration of what we hope to students to be able to do once they successfully finish the course. Ideally, the demonstration will be around 30 minutes of live coding, showing how to go from raw data or idea, through data processing, looking up how to use language and library APIs (e.g., Googling answers on StackExchange), to finally produce a visualisation.

1.2.5 Laboratory Sessions

Weekly laboratory sessions will be staffed by tutors and lecturers to help students who are having difficulty with the concepts developed in class. The first laboratory session will be dedictaed to ensuring that students are able to access, and are familiar with, the software development tools that will be used throughout the course. Thereafter, laboratory sessions will be driven by specific student needs as they arise. Laboratory attendance is not compulsory.

1.2.6 Seeking Help

By the end of the course students will be equiped with enough knowledge and skills to be able to seek help from web resources such as tutorial sites and newsgroups. During the course students will be able to seek help in a more structured way.

- For help relating to course logistics students should talk to course staff after lectures or during laboratory sessions, or send an email to the course email address.
- For help with weekly exercises students can make use of the online chat facility.
- For help with assignments students should raise the question in the issue tracking system within GitLab and assign the issue to course staff.

The following resources may be useful:

[1] Punch and Enbody, "The Practice of Computing Using Python (2nd Ed.)", *Pearson Education, Inc.*, 2013.

[2] Lutz, "Learning Python (5th Ed.)", *O'Reilly Media*, 2013.

[3] `https://www.python.org/`

[4] `http://www.tutorialspoint.com/python/`

[5] `http://pythontutor.com/`

[6] `http://stackoverflow.com/`

[7] `https://gitlab.cecs.anu.edu.au/groups/comp1040`

Chapter 2

Assessment and Practice

The primary objective of this course is to equip students with computational skills that will benefit their university studies and future careers. As such, a significant amount of time will be devoted to practice. This is reflected in the structure of the assessments, which include:

- weekly exercises (10%);
- three individual assignments (10%, 20% and 20%);
- one substantial group project (40%).

There will be no final exam for this course. Final marks for the course may be moderated by the Research School of Computer Science examiners' meeting.

2.1 Assessment Policy

Exercises. Exercises are aimed at practicing a specific concept. These should be done individually and submitted through the online CodeBench tool. A set of ten exercises is assigned each week throughout the semester. Each weekly set contributes up to 1% of the final course grade. We will take the best ten weekly exercises over the semester (for a maximum of 10%).

Assignments. Learning a new skill is often best done in teams. In this course we encourage discussion of assignments but the final assignment submission must be your own work. In particular, you should not consult written or electronic notes from other students when preparing your solution.

Project. Projects are intended to be done in groups (of up to four students); collaboration is essential and the final submission must be a joint submission. Students should indicate a rough breakdown of contributions from each group member.

2.1.1 Evidence of Own Work

The course staff reserve the right to ask students to explain parts of work submitted for assessment. Marks may be adjusted if the student cannot adequately explain the work.

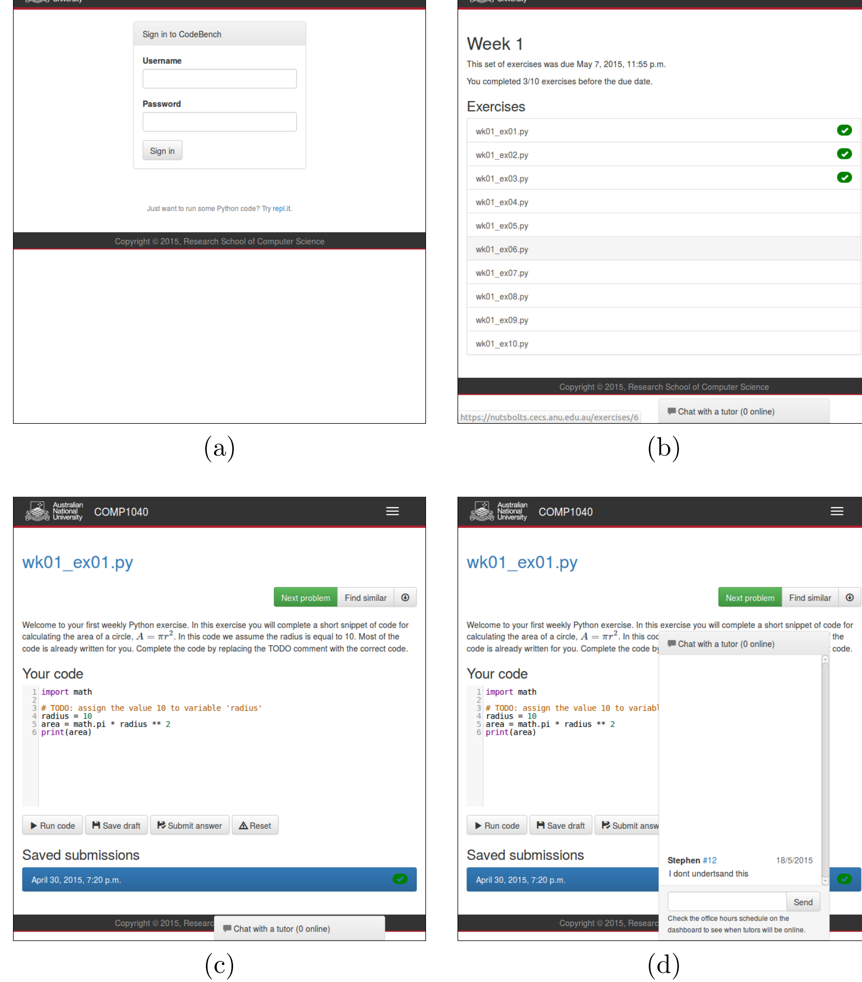


Figure 2.1: Example screenshots from online framework for weekly exercises.

2.1.2 Late Penalty

Students are expected to work consistently on assignments and projects throughout the semester and keep an up-to-date software source code respository. Marks will be awarded based on material submitted (i.e., in the repository) at the assessment due date and time. Consistent with ANU policy, extensions will not be granted for mismanagement of time or resources. A doctor's certificate is required to receive an extension as a result of illness.

2.2 Weekly Exercises

Weekly exercises provide students with an opportunity to practice programming and debugging while at the same time being exposed to a wide variety of coding examples. Exercises are submitted via the online CodeBench framework and are graded automatically. Students receive immediate feedback and have the

opportunity to correct errors and resubmit (before the deadline).

The framework provides students with a dashboard showing their progress. A small set of mandatory exercises (taking an estimated one hour maximum to complete) will be assigned per week. In addition, a student can elect to practice on more exercises (not for credit). Exercises do not need to be completed in one session—students can save exercises and return to them later. Submitting an exercise automatically saves it (and evaluates it for correctness).

Students can login to the framework at `https://codebench.cecs.anu.edu.au` using their ANU username and password.

Figure 2.1 shows screenshots from the framework. Figure 2.1(a) shows the CodeBench login screen, which requires a student (or course staff member) to enter their username and password. Figure 2.1(b) shows the list of exercises for Week 1 and indicates which exercises have already been completed. An example exercise is shown in Figure 2.1(c). Once complete the student can elect to automatically proceed to the next exercise in the weekly set. Students can also choose to find similar (non-assessable) exercises to practice on. Writing and executing code can be done within the browser. Alternatively, the student can download the code (and resources) and experiment using the standard tool chain if they are more comfortable doing so.

Figure 2.1(d) shows an online chat tool that is available within the framework for students to seek help from course staff. If course staff are not online the messages get cached. The main CodeBench dashboard shows a calendar of office hours for when students can expect course staff to be online.

In addition to the assigned weekly exercises, students can practice on non-assessable exercises. These are tagged with keywords that facilitate finding exercises on a particular topic. Some standard tags are: indexing, loops, strings, functions, conditionals, containers, files, expressions, etc.

2.3 Assignments

Instructions for each assignment will be provided 2–3 weeks prior to the due date. Starter code for assignments will be made available via the course GitLab account. Students should fork the assignment repository and change the visibility to private. Assignments should be submitted via GitLab and must be the student's own work.

Assignments should be completed in Python 3. Only libraries supplied with the Anaconda distribution can be used unless otherwise stated in the assignment instructions.

A set of tests will be provided with the assignment starter code but additional tests may be run by the course staff when assessing the submitted code. Code will also be marked for being well structured, clearly designed, and easy to follow.

2.4 Projects

Projects involve the development of a more substantial piece of software than assignments. They also involve working in a team. Projects are marked based

on submission of the project code (40%), report (40%) and an in-class oral presentation (20%).

Any libraries (and, in deed, programming language) can be used for the projects but external code and other people's work must be clearly acknowledged. Students must discuss projects and intended programming languages with course staff prior to commencing the project. If a project is not done in Python then detailed installation instructions for entire software suite must be provided.

Chapter 3

Lectures

Lecture sessions run for 50 minutes. They will be a combination of traditional lecturing combined with and live coding demonstrations. It is expected that students actively participate in the lectures. We hope to video record all lecture sessions but students will gain more by attending and participating in the lectures. A tentative lecture schedule is provided below.

Wk	Lecture A (9am Mondays)	Lecture B (2pm Tuesdays)	Lecture C (9am Wednesdays)
1	Welcome and Demo	Introduction to Coding	S/W Dev. Basics
2	Data Structures I	Execution Models	Data Formats and Files
3	String Processing	Functions	Computer Architecture
4	Data Structures II	(guest lecture)	Objects and Classes
5	S/W Tools and Practices	Reading Source Code	Libraries and APIs
6	Searching for Help	(guest lecture)	Advanced Rev. Control
7	Visualising Data	(guest lecture)	Visualising Data II
		—semester break—	
8	Debugging Strategies	S/W Design	Collaborating
9	(public holiday)	(no lecture)	(no lecture)
10	(public holiday)	Refactoring Code	Advanced Programming
11	Regular Expressions	Data Pipelines	Optimizing Code
12	Defensive Programming	Programming Languages	Software Distribution
13		(project presentations)	

Table 3.1: Tentative lecture Schedule.

Time permitting some additional advanced topics may be covered. These include: client-server programming, databases, concurrency, and Jupyter notebooks.

Non-compulsory laboratory sessions will be run each week. The lab in the first week will be dedicated to helping students set up their development environment. Thereon lab sessions will be unstructured—students are free to attend any lab session and raise questions about material covered in lectures or seek help on exercises or assignments.

3.1 Lecture 1: Welcome to The Craft of Computing

Learning Outcomes

- Course welcome and logistics
- Live coding demonstration touching on the key concepts that will be developed throughout the course
- Information on first week laboratory sessions for installing software and setting up development environments

Overview

The first lecture is primarily a demonstration of what we hope students will be able to do once they successfully finish the course. The demonstration will be around 45 minutes of live coding, showing how to go from raw data or idea through to data processing, looking up how to use language and library APIs (e.g., Googling answers on StackExchange), and collaborating to finally produce a visualisation that can be shared in a report or web page.

3.1.1 Logistics and Overview

- Introduction of course staff, lecture times, and lab times.
- What this course is about and handout of questionnaire.
- Overview of assessment and expected workload (assignments, projects and exercises).
- Where to get course notes or revisit recorded lectures.
- Let's get started...

3.1.2 Demo: Actors and Movies

During this demo you will be exposed to the process of gluing code together to solve a specific task. The focus is on "getting stuff done" rather than detailed planning and analysis, and will touch on some key philosophies of the course—searching the web for help, directed reading of API documentation, and testing as you go. Along the way you will also be introduced to the Python programming language, the PyCharm IDE, and GitLab for revision control (the online exercise tool CodeBench will not be introduced until Lecture 2).

Importantly, you will see many things that will be new to you during this lecture. Don't worry about trying to understand all of the concepts now. By the end of the course you will be able to do everything we show here, and more!

The task for this demo is as follows: We are given a dataset containing a list of movies and actors who starred in them. The dataset was extracted from IMDB and preprocessed to only include the top 250 most popular movies (and actors who appear more than twice in these movies). Our goal is to understand which actors often appear in movies together.

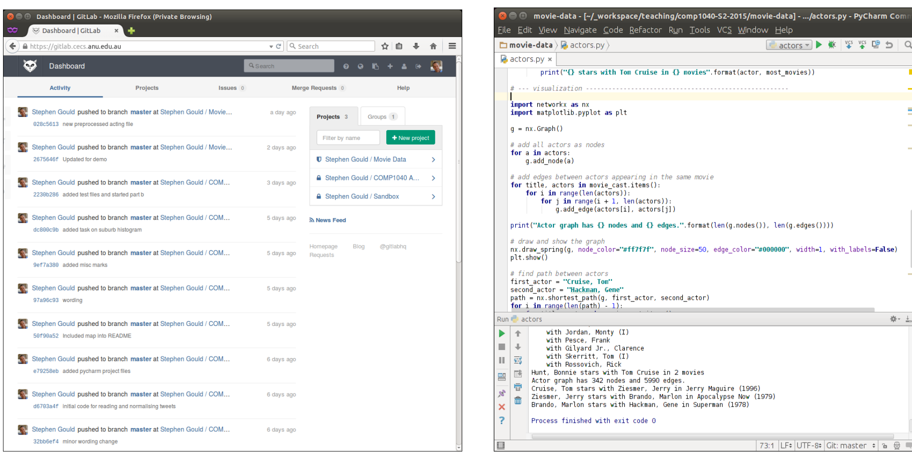


Figure 3.1: Snapshots of two of the tools used in this course–the GitLab repository web interface and the PyCharm integrated development environment (IDE).

Starting the Project and Reading the Data

- Fork dataset project on GitLab (`https://gitlab.cecs.anu.edu.au/comp1040/comp1040-movie-data`).
- Start PyCharm and choose to "Check out from Version Control". Copy and paste the URL from GitLab and choose the destination directory. This will clone a local copy of the Git project. Say "Yes" to opening the cloned project.
- Open the `acting.json` file and look at format.
- Add a new file to the project (`File|New...`). Call it `actors.py`.
  - Click "OK" to add the new file to Git.
- Write some comments about what the code is doing
- Write code to open and read the dataset into an appropriate data structure
  - Look up how to load a `json` file (`https://docs.python.org/3/library/json.html`)
  - Import the `json` library
  - Store in a dictionary of actors indexed by movie
- Test that the code works
- Commit and push code to the repository. Show that the modified code has appeared in GitLab.

Python Code

```python
import json

# Read the database of actors and the movies that they appear in,
# and store as a map of movie titles to actors.

movie_cast = dict()

with open('acting.json') as file:
    data = json.load(file)
    for actor, movies in data.items():
        for title in movies:
            if title not in movie_cast:
                movie_cast[title] = list()
            movie_cast[title].append(actor)

# check that we have read the file correctly by
# looking at one or two movies
print(movie_cast["Star Wars (1977)"])
print(movie_cast["Rain Man (1988)"])
```

Find Most Common Co-Star for a Given Actor

- Write a function that given an actor's name will find all actors that appear with them in any movie
- Test the function with some examples
- Write some code to find the actor who appears the most with a given actor
- Test using the costars found previously
- Commit and push the code

Python Code

```python
# write a function for finding the costars of a given actor.
def find_costars(star):
    costars = []
    for title, actors in movie_cast.items():
        if star in actors:
            print("{} stars in {}".format(star, title))
            for actor in actors:
                if actor != star and actor not in costars:
                    costars.append(actor)
    return costars

# test the function with "Tom Cruise"
star = "Cruise, Tom"
costars = sorted(find_costars(star))
print("Co-stars of {}: {}".format(star, '; '.join(costars)))
```

Python Code

```python
# write a function that finds movies in common between two actors
def movies_in_common(star, costar):
    movies = []
    for title, actors in movie_cast.items():
        if star in actors and costar in actors:
            movies.append(title)
    return movies

# for each costar, find out how many movies they have in common
shared_movie_count = {}
for costar in costars:
    shared_movie_count[costar] = len(movies_in_common(star, costar))

# find the costar with most movies in common and show those movies
max_count = max(shared_movie_count.values())
for costar, n in shared_movie_count.items():
    if n == max_count:
        print("{} stars with {} {} times".format(costar, star, n))
        print("   in {}".format(movies_in_common(star, costar)))
```

Visualization

- Add a collaborator to the GitLab project (via `Settings|Members`)
- Collaborator opens PyCharm, clones project on separate machine, and makes sure the code runs
- Add visualization of links between all actors that appears in a move together
- Search NetworkX documentation for how to do this
- Commit and push the code

Python Code

```python
import networkx as nx
import matplotlib.pyplot as plt

# extract the set of actors from movie cast
all_actors = set([actor for cast in movie_cast.values()
                        for actor in cast])

# create a graph and add the actors as nodes
g = nx.Graph()
for a in all_actors:
    g.add_node(a)

# add edges between actors appearing together in the same movie
for title, actors in movie_cast.items():
    for i in range(len(actors)):
        for j in range(i + 1, len(actors)):
            g.add_edge(actors[i], actors[j])

# draw and show the graph
nx.draw(g, node_color="#ff7f7f", node_size=50, edge_color="#000000",
    width=1, with_labels=False)
plt.show()
```

Visualization II

- First project member updates to get the visualization code and makes sure it runs
- Adds an item to the issue tracking system asking for a feature that highlights the path between any two actors
- Second project member views issue tracker
- Adds functionality to highlight a path between any two actors and marks issue as done
- Show some examples with pairs of actors and also output the sequence of movie-actor pairs along the path
- Commit final code to repository and close issue

Python Code

```
# find path between actors
first_actor = "Cruise, Tom"
second_actor = "Hackman, Gene"
path = nx.shortest_path(g, first_actor, second_actor)

for i in range(len(path) - 1):
    for title, actors in movie_cast.items():
        if (path[i] in actors) and (path[i + 1] in actors):
            print("{} stars with {} in {}".format(path[i],
                path[i + 1], title))
            break
```

Python Code

```
# draw and show path on the graph
path_edges = list(zip(path, path[1:]))
node_labels = dict([(actor, actor) for actor in path])

positions = nx.spring_layout(g)
nx.draw(g, positions, edge_color="#7f7f7f", node_size=50,
    width=1, alpha=0.25)
nx.draw_networkx_edges(g, positions, edgelist=path_edges,
    edge_color="#0000ff", width=4)
nx.draw_networkx_nodes(g, positions, nodelist=path,
    node_color="#7f7fff", node_size=50)
nx.draw_networkx_labels(g, positions, labels=node_labels)
plt.show()
```

3.1.3 Next Lecture

- Introduction to programming and Python
- Using the CodeBench exercise framework
- Release of first assignment

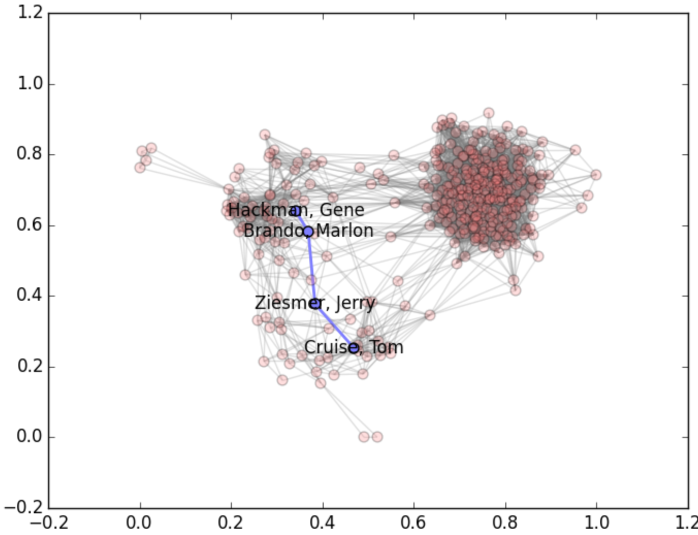


Figure 3.2: Visualisation of the path between Tom Cruise and Gene Hackman in the top 250 movies database.

3.2 Lecture 2: Introduction to Coding

Learning Outcomes

- Understand that software is implemented as a sequence of instructions which determine the behaviour of a program.
- Understand some basic Python syntax and constructs including comments, variable assignment, and simple expressions.
- Know what can go wrong when writing software and that testing should be an integral part of software development.

Overview

In this lecture we will discuss the concept of a program as a sequence of instructions that are interpreted by a computer. However, emphasis will be placed on software being written for people not computers and hence the importance of good design, naming conventions, and comments. The distinction between declaritive versus imperative knowledge will be demonstrated. A small program will then be studied to introduce some basic programming constructs and illustrate what can go wrong when writing programs (i.e., bugs) and how to mitigate these problems.

3.2.1 Programming Languages

There are many, many different programming languages. Some differ only slightly in syntax and style, others differ drammaticaly in the way that you think about a program. In this course we will teach the Python programming language, but by the end of the course you should have sufficient knowledge to be able to learn any new programming language by yourself. You will also be competent at designing, coding, and debugging small programs in Python.

So what is a program?

A program (written in a programming language) is a sequence of instructions that gets intrepretted by a computer to achieve some computational task. But, while it is important for the computer to be able to interpret the program, it is equally important that a program be understandable to other programmers. This is because programs need to be checked for correctness (i.e., tested) and maintained—requirements change over time. Always keep in mind that *programs are written for people not for machines.*

Some programming languages do not execute instructions sequentially but they will not be considered in this course.

Declarative versus Imperative Knowledge

An important distinction when thinking about programming is that between declarative knowledge and imperative knowledge. Consider the definition of a square root: *The square root of a number x is a number y such that the product of the number y with itself is equal to x.* Mathematically we write

$$\sqrt{x} = y \text{ subject to } y^2 = x \text{ and } y \geq 0$$

This is an example of declarative knowledge. It tells us what the square root is but does not tell us how to compute it. Imperative knowledge (sometimes

called procedural knowledge), on the other hand, gives us a recipe or procedure for performing some task. For example, the following algorithm tells us how to compute a square root.

1. **Initialize.** Start with some guess, y.
2. **Check.** If y^2 is approximately equal to x then stop.
3. **Update.** Otherwise compute $y \leftarrow \frac{1}{2}\left(y + \frac{x}{y}\right)$.
4. **Repeat.** Go to Step 2.

Python is an imperative programming language. We need to tell the computer how to perform a task.

3.2.2 A First Example in Python

Let's now consider a simple example of writing a program to calculate the area and circumference of a circle with radius 10cm using formalae $A = \pi r^2$ and $C = 2\pi r$, respectively. In Python we could write:

In Python the ** operator indicates exponentiation, so `10 ** 2` means 10^2.

Python Console

```
>>> print(3.141 * 10 ** 2)
314.1
>>> print(2 * 3.141 * 10)
62.82
```

This is fine if all we wish to do is perform this calculation once. However, if the calculation were used in some larger piece of software there would be some problems. First, the numerical expressions are not easy to interpret and obscure the calculation being done. Second, we would need to recompute the value each time we needed it in the program. Third, if we ever changed the radius we would need to work out how to change the expression. An alternative is to use variables and comments to help the reader of our program (which could be our future selves) understand what we are trying to do

A **variable** in a computer program is used to store or reference data.

Python Code

```
# calculate area and circumference of a circle

import math

radius = 10
area = math.pi * radius ** 2
circumference = 2 * math.pi * radius
print(area, circumference)
```

The code may look more involved but it's actually easier to understand what it is doing and has the advantage that we can avoid recalculating the area or circumference each time they are needed (since they are stored in variables `area` and `circumference`, respectively). Note that the variables also act to document the code. Another way to document code is through comments. Comments are intended to explain the code to yourself and to other programmers. They are ignored by the computer. In Python comments start with a # character. Like much of good programming commenting is a matter of style—you do not need to go overboard or repeat what is obvious in the code.

Comments help other programmers understand your code.

Python Code

```
# This Python code snippet calculates the area and circumference
# of a circle of radius 10cm.

# Import the math module so that we can use the constant math.pi
# instead of having to define it ourselves.
import math

# Now set the radius of the circle to 10 by assigning the value
# 10 to a variable called radius.
radius = 10

# The following two lines calculate the area and circumference of
# the circle using the value in the variable called radius. The
# results are stored in variables with names corresponding to the
# value being stored.
area = math.pi * radius ** 2
circumference = 2 * math.pi * radius

# Print both the area and circumference that were just computed.
print(area, circumference)
```

We can also easily compute the area and circumference of a circle with different radius by simply changing the `radius` variable. Later, we will learn about *functions* which makes this even easier. Note that you should always remember to change your comments when you change your code. Comments that are inconsistent with the code can be very confusing.

Operator Precedence

Like in standard mathematical notation operators in expressions are evaluated in a certain order. For example, in the calculation of area in the code above the radius is squared (`radius ** 2`) before it is multiplied by π (`math.pi * radius ** 2`). Parentheses can be used to control the order of evaluation. So

```
(a * b) ** 2
```

computes $(ab)^2 = a^2b^2$ which is different from

```
a * b ** 2
```

Table 3.2 lists the operator precedence for standard Python operators.

Variables

A *variable* in a computer program is used to store or reference data. We saw examples in the code snippets above. Variables names should be descriptive to help document the code and make it more maintainable. Often programmers will use naming conventions to further help maintainability of the code. For example, a variable might be called `name_str` where the suffix `_str` is used to indicate that the variable contains a string.

Programming languages also put restrictions on what can be used as a valid variable name. For example, certain *keywords* are reserved by the langauge and cannot be used (the `import` and `print` keywords in the code snippets above are two examples). In Python variable names can contain letters, numbers and underscores ("_"), and must between with a letter or underscore. Variable names

Often underscores are used to separate words in variable names, e.g., `area_of_circle`. This is just a convention. Another common convention is CamelCase where the first character of each word is capitalised (e.g., `areaOfCircle`).

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>or</td><td>Boolean OR</td></tr>
<tr><td>and</td><td>Boolean AND</td></tr>
<tr><td>not</td><td>Boolean NOT</td></tr>
<tr><td>in, not in, is, is not,
<, <=, >, >=, <>, !=, ==</td><td>Comparisons</td></tr>
<tr><td>|</td><td>Bitwise OR</td></tr>
<tr><td>^</td><td>Bitwise XOR</td></tr>
<tr><td>&</td><td>Bitwise AND</td></tr>
<tr><td><<, >></td><td>Shifts</td></tr>
<tr><td>+, -</td><td>Addition and subtraction</td></tr>
<tr><td>*, /, //, %</td><td>Multiplication, division, remainder</td></tr>
<tr><td>+x, -x, ~x</td><td>Positive, negative, bitwise NOT</td></tr>
<tr><td>**</td><td>Exponentiation</td></tr>
</table>

Table 3.2: Operator precedence for standard Python operators ordered from lowest to highest precedence. Operators listed on the same row have the same precedence and most are evaluated from left to right (see the Python documentation for exceptions).

are case sensitive. Other characters, such as space and symbols, have special meaning and cannot be used in variable names.

There are three important operations that can be done with variables. These are creating the variable, assigning a (new) value to the variable, and extracting a value from the variable. Usually a value is assigned to a variable when it is first created (sometimes called initialisation). A variable cannot be used in an expression before it is created. For example, in the circle code above we first assigned the value 10 to the variable `radius` before using the variable `radius` in the expressions for calculating area and circumference.

Variable assignment is done using the equals operator (`=`). The left-hand side of the operator contains only the name of the variable being assigned to and the right-hand side of the operator contains the expression that is being assigned. So

```
hypotenuse = math.sqrt(3 ** 2 + 4 ** 2)
```

is legal, where as

```
hypotenuse ** 2 = 3 ** 2 + 4 ** 2
```

is not. The expression (on the right-hand side) is always evaluated before the value is assigned. This allows you to update a variable based on its current value. For example,

```
count = count + 1
```

increments the value stored in the variable `count` by one and stores the result back in `count`.

Some variables/constants (and other functionality) are provided as extensions to the language via *libraries* (sometimes called modules). The `import math` statement in example code above gave our small program access to a host of pre-existing variables and functions. In particular, we made use of the `math.pi` variable which defines π up to machine precision. Try running the

Libraries will be covered in detail in a future lecture.

command `help(math)` in a Python shell to see what else is provided in the `math` library. Alternatively visit the online Python documentation at `http://docs.python.org/3/library/math.html`.

Program Execution Flow

At the core of any programming language is the concept of control flow, which dictates the sequence in which lines of code are executed in the language. In the circle example above each line of code was executed once in order. However, it is possible to have more elaborate control via conditional execution and loops. An example is the Babylonian algorithm for computing the square root of a number, which repeats the same basic calculation over and over. In Python it would be implemented as follows.

Python Code

```
# Function to compute a square root using the Babylonian algorithm
def square_root(x):
    y = 0.5 * x
    while (y * y != x):
        y = 0.5 * (y + x / y)
    return y
```

The `while` keyword tells Python that the following indented statements are to be executed repeatedly until the condition in parentheses is met. We will cover loops and conditional execution in detail in a later lecture.

Statements perform actions; **Expressions** return values.

In Python we refer to a line of code (roughly) as a *statement*. Statements can be thought of as the smallest standalone unit within a programming language. They are commands that perform some action (sometimes called a side effect) but do not return any values. For example, assigning the result of some computation to a variable is a statement. Statements often contain *expressions*. The calculation of the area of a circle

```
math.pi * radius ** 2
```

is an expression. Expressions can act as statements but statements cannot act as expressions.

We can test the above Babylonian algorithm code by *calling* or *evaluating* the function on some test cases:

Python Console

```
>>> square_root(4.0)
2.0
>>> square_root(25.0)
5.0
>>> square_root(4.41)
2.1
>>> square_root(10.0)
?
```

Return Values and Side Effects

Beginners to programming often confuse return values and side effects. For example, the expression `"hello" + "world"` returns a value—the character string `helloworld`—which can be assigned to a variable (or used in a larger

expression). The statement `print("helloworld")` does not return anything, but does have the side effect of displaying `helloworld` on the screen. This can be particularly confusing in the Python console which displays the value returned by an expression if it is not explicitly assigned to a variable.

Python Console

```
>>> print("helloworld") # statement
helloworld
>>> "hello" + "world"   # expression (console evaluates and prints)
helloworld
```

Getting User Input

So far our program is fairly limited—it can only calculate the area and circumference of a circle with fixed radius, namely 10cm. Our program would be much more useful if it could calculate the area and circumference of a circle with arbitrary radius. The following code snippet prompts the user to enter a radius each time it is run.

Python Code

```
# calculate area and circumference of a circle
# with radius from user input

import math

radius = float(input("Enter radius:"))
area = math.pi * radius ** 2
circumference = 2 * math.pi * radius
print (area, circumference)
```

Note the line `radius = float(input("Enter radius:"))`. This line does a number of things. First, it prompts the user to enter a radius. Second, it waits for user input. Third, it converts the text that the user inputs into a floating-point number. Last, it allocates the variables `radius` and assigns the user-entered value to it.

We will discuss variable **types** in a later lecture.

Whitespace, Indentation, and Continuation

Formatting of your code is important not just to make it easy for humans to read but the layout of the code also affects how it is interpretted by the computer. Whitespace are characters that do not print (e.g., space, tab and end-of-lines). Whitespace separates keywords and make your code more readable. For the most part, you can place whitespace anywhere in your program. However, there are some special exceptions.

Indentation, i.e., whitespace at the start of a line, is special in Python (not so in most other programming languages). In Python, indentation defines a block of code (called a *suite* in Python) whose execution is conditioned on some test or part of a loop. We will see examples of indentation when we discuss conditional execution, loops, and functions.

Python is also sensitive to line endings. Usually the end of the line delimits the end of a statement. If you want a statement to continue over multiple lines (e.g., to make the code more readable) you can use the backslash character (\),

```
area_of_circle = \
  math.pi * radius_of_circle ** 2
```

Continuation is particularly useful when printing out long strings, e.g.,

```
print("It is the mark of an educated mind to be able to", \
    " entertain a thought without accepting it---Aristotle.")
```

will print the single sentence *"It is the mark of an educated mind to be able to entertain a thought without accepting it—Aristotle."* Note that continuation does not work for comments. Every line of a comment must begin with a # character.

3.2.3 What Can Go Wrong With Programs?

There are many things that can go wrong when writing software. This is why it is important to test thoroughly and test often. Many programmers will write a suite of tests before they write the code. Errors in software are generally called bugs. Remember, a good programmer is not someone who writes correct code the first time, but someone who can quickly detect and correct bugs and make it easy for others to do so too. We will discuss debugging strategies in great detail in a later lecture. Here we summarise the types of errors that can occur in a program.

The first electronic computers, assembled from vaccuum tubes, got very hot and attracted insects. Every now and then an insect would touch an electric circuit and fry a component (and itself). The process of replacing the component became known as **debugging**.

Syntax Errors

Syntax errors are errors where the code that has been typed in is malformed in some way. That is, the code does not follow the rules of the programming language. For example, using a variable before it is declared is a syntax error. Syntax errors are the easiest to detect because Python stops execution and warns us about them. However, interpretting these warnings and correcting errors is something that requires practice.

Runtime Errors

Runtime errors are not due to anything wrong in the way that the program was written but are a result of trying to perform some illegal or ill-defined operation. For example, dividing any number by zero is not defined but Python may not know that the demonenator in some expression is zero until the program is run. When Python does try to evaluate the expression it will stop executing the program and print an error message. Detecting and preventing this type of error will make your program much more robust.

Another common runtime error involves invalid or illegal memory accesses. This can result in crashing the program or worse undetected data corruption. A very common cause of illegal memory access occurs when accessing elements outside of the bounds of an array. We will discuss this more in Lecture 4.

Logical/Algorithmic Errors

Logical errors are much more subtle. These are errors that do not prevent your program from running, rather they result in an incorrect result being obtained. For example, forgetting to square the radius in the calculation of the area of

a circle will produce the wrong result (except in the case where the radius is exactly zero or one) but Python will very happily run the code. Only testing and careful code inspection will pick up these sorts of errors.

Computational Limitations

All computers perform numerical calculations using finite precision arithmetic. Some errors occur because we are operating near the limitation of the computer's ability to represent a number. This type of error is very common when testing for equality of two floating-point numbers. For example, on a 64-bit computer $2^{31} + 2^{-22}$ evaluates to 2^{31}. Try the following in Python,

Unlike in mathematics, in Python the operator == is used to test for equality while the operator = is used for variable assignment.

Python Console

```
>>> a = 2.0 ** 31
>>> b = a + 0.5 ** 22
>>> print(a == b)
True
```

The result of the comparison should be `False` but Python prints `True`.

Our code for the Babylonian algorithm above contains this type of error. For many input values the code works fine. However, the comparison `y * y != x` is susceptible to errors caused by numerical precision, and the program may never terminate. A solution is to recognise that any calculation is only performed up to some fixed precision. The inequality test can then be changed to `abs(y * y - x) > 1.0e-6` where we are only checking whether the difference between y^2 and x is greater than one millionth.[1] With this change we would find square-roots up to a precision that is sufficient for most purposes.

Python Code

```
# More robust version of the Babylonian algorithm
def square_root(x):
    y = 0.5 * x
    while abs(y * y - x) > 1.0e-6:
        y = 0.5 * (y + x / y)
    return y
```

Python Console

```
>>> square_root(10.0)
3.1622776604441363
```

CodeBench Exercise Framework

The CodeBench framework has been designed for this course to give you practice in writing and debugging small snippets of code. The use of CodeBench for completing and submitting weekly exercises will be demonstrated in the lecture. To access to CodeBench go to `https://codebench.cecs.anu.edu.au/`, select the appropriate course (including semester and year), and login with your ANU credentials.

[1] In most programming languages `1.0e-6` in interpretted as 10^{-6} where the `e` is used to represent 10 to some power. So `5.3e7` means 5.3×10^7 or $53,000,000$.

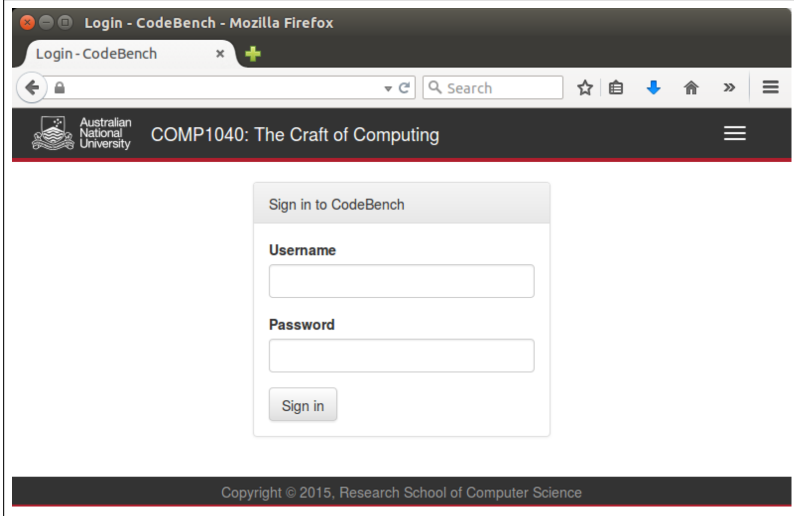


Figure 3.3: The CodeBench login screen.

3.2.4 Next Lecture

- The software development process
- Text editors and IDEs
- Saving and running programs
- Introduction to revision control and issue tracking

3.3 Lecture 3: Software Development Basics—Tools and Evironments

Learning Outcomes

- Understand the basic process for developing software (design, implement, test, and revise).
- Understand that software source code is stored in text files on a computer and interpretted or compiled to produce executable machine instructions.
- Understand how files are stored within a filesystem.
- Be able to navigate a filesystem to create, modify, copy, and delete files and directories.
- Understand the difference between raw text files and formatted text such as in word processing documents.
- Understand the role of a text editor versus a word processor in writing software.
- Install and perform basic tasks (opening files, running code) using an IDE (PyCharm).
- Be able to start and use a Python console to interact with code.
- Install and perform basic revision control tasks (checking in and out code) using GitLab and PyCharm.
- Understand issue reporting and tracking.
- Understand the relationship between IDEs and underlying tools (e.g., interpreters, version control, file system, etc.)

Overview

In this lecture we introduce text editors and integrated development environments as the mechanisms for developing code. We will discuss the tools that will be used in the course and provide instructions for how they can be installed on Windows, Linux, and Mac OS/X. A brief overview of filesystems and the distinction of source code versus data (e.g., a Word document) will be given.

3.3.1 Text Editors and IDEs

Writing, running, and testing code usually involves using several different tools: a documentation browser; an editor for writing the code; the file system for saving and loading code and data; an interpreter or compiler to execute code; and version control for managing code changes and sharing. As its name suggests, an *Integrated Development Environment* or IDE is an application that provides an environment for pulling together all the various tools required to develop code. These are usually presented as a unified "workbench" of windows and tabs that allow you easily shift between writing, running, debugging, and versioning code. IDEs also typically include extra features such as *syntax highlighting* (colouring keywords, strings, and variables) and *auto-completion* (showing lists of accessible variables, functions, and function arguments).

We will use the Community Edition of the PyCharm IDE (`https://www.jetbrains.com/pycharm/`) in this course. It is a free, easy to use IDE that

Software versions will change from time-to-time and the interface that you see may appear slightly different from that described in these notes. However, the general concepts remain fairly constant between versions.

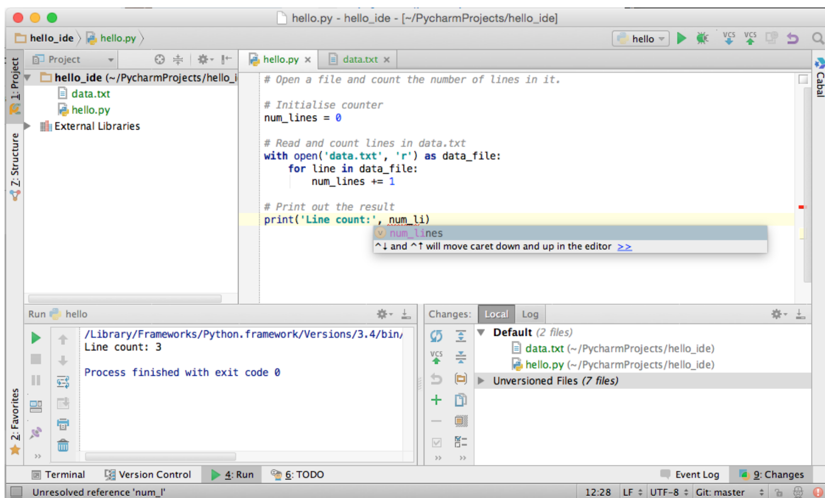


Figure 3.4: An overview of the PyCharm Community Edition IDE.

is specifically designed for Python development. It also has extensive online documentation and video tutorials available at

`https://www.jetbrains.com/pycharm/documentation/`.

Figure 3.4 shows a screenshot of PyCharm in action. At first sight it looks quite complicated but we will explain what all the various tabs, frames, pop-up, etc. do later in the course. For the moment, here is a quick sense of what is going on:

- Python code is being edited in the large panel in the top right. Various keywords, such as `with` and `open` are coloured blue and strings are coloured in green. We use the same colour scheme for syntax highlighting in these notes. There is also a pop-up box showing that `num_l` can be auto-completed to `num_lines` by pressing (Enter). You can press (Esc) to close the pop-up without auto-completing.

- The top left panel is a file browser showing the files in the current project, including `hello.py`, the file being edited. If the files are not showing you can double-click on the project name (just below the red, amber and green window controls) to show the panel. Double-clicking on filename will open that file for editing.

- The bottom left panel shows the result of executing the code ("`Line count: 3`"). Blue text is status information added by the IDE.

- The bottom right panel shows which files are under version control and have uncommitted changed. Display this panel using menu item `VCS|Show Changes View`. You can get PyCharm to show you the difference between your file and the last commit by right-clicking on the filename and choosing `Git|Compare With...` from the context menu.

The layout of panels can be customized to your own liking. If you ever get into trouble with the layout you can select `Window|Restore Default Layout`

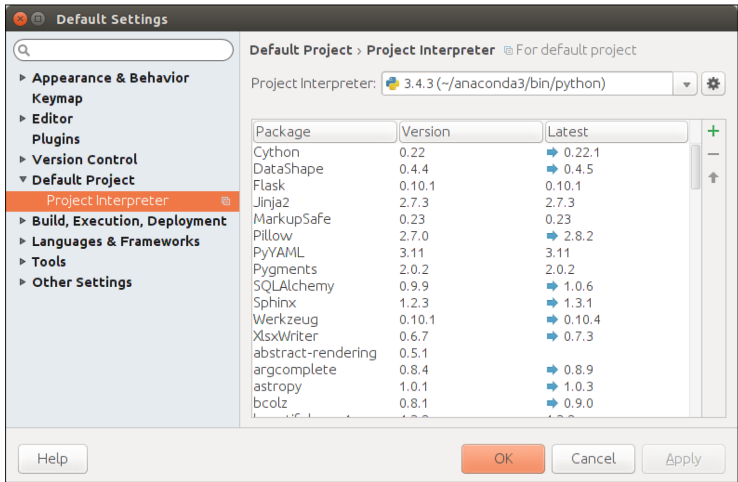


Figure 3.5: Screenshot showing how to set the default Python intrepreter in the PyCharm IDE on a Linux system.

from the menu to get back to the standard environment. The same effect results from holding down the Shift key and pressing F12 (denoted (Shift-F12)). Such key combinations are known as a hot keys or accelerator keys and appear next to menu items.

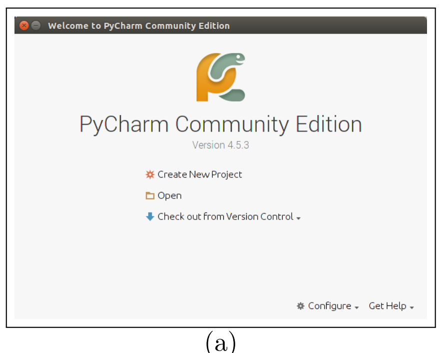
Installing and configuring PyCharm

PyCharm is available for all major operating systems and can be downloaded from `https://www.jetbrains.com/pycharm/download/`. See inline documentation for detailed installation instructions for your operating system.

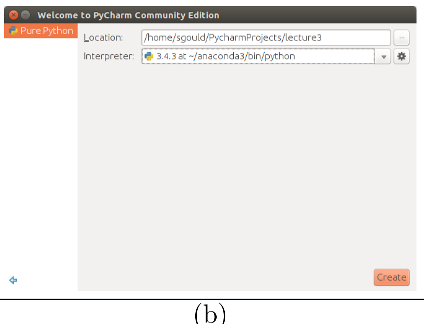
Run PyCharm like you would any other application on your system. On some operating systems an icon may have been installed on your desktop. Otherwise you will need to find where PyCharm was installed. On Linux and Mac OS X you can start PyCharm from a command shell by executing `pycharm`.

The first thing you will want to do is configure PyCharm to use the Anaconda distribution of Python, which we assume you have already installed on your system (if not, see the installation instructions in Chapter 4). This can be done on a per-project basis, but it is easiest if you set the default as follows:

- Select `Configure|Settings` or `Configure|Preferences` from the “Welcome to PyCharm” window. If you have already opened a project you can modify the settings using the `File|Default Settings` menu item.
- Select `Default Project > Project Interpreter` from the Default Settings window, which should have just appeared.
- In the drop-down list choose the Anaconda Python interpreter. A list of packages installed with the distribution will appear. Figure 3.5 shows a screenshot of what you should see.
- Click Apply and then OK. PyCharm may perform some preprocessing of the Anaconda libraries, which may take some time.



(a)



(b)

Figure 3.6: (a) The PyCharm start up window allows you to create new projects, open existing projects, or checkout a project from a version control system. (b) Giving a new project a name and setting the interpreter.

Writing and running a program

PyCharm is a fully featured professional IDE. We will not be using all of its features in this course. The main features that we will be interested in are:

- Starting a new project and re-loading an existing one
- Creating new Python files
- Editing files and features of the editor (syntax highlighting, auto-completion, error detection, and in-built documentation)
- Running code and observing output
- Integration with version control systems (VCS)

Throughout this lecture we will use a running code example of reformatting some text. Specifically, assume we are given a file of names in the form of `LASTNAME, Given Name`. We want to process the file to produce a list with the last name and first initial only. For example, given the input file

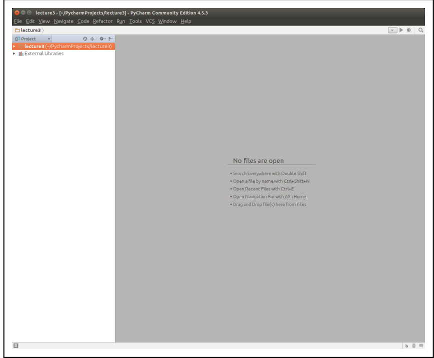
```
GOULD, Stephen
REID, Mark
...
```

we want to produce

```
S. Gould
M. Reid
...
```

Our first step is to create a new PyCharm project. From the PyCharm start window select `Create New Project`. See Figure 3.6. We are asked to give our project a location where it will be stored on the filesystem and select an interpreter. Make sure the interpreter is set to the Anaconda 3 version of Python. The last part (directory) of the location can be thought of as the project name. Click the “Create” button.

The PyCharm code editor will appear with an empty project. Our second step is to start adding files to the project. We will copy the input names file



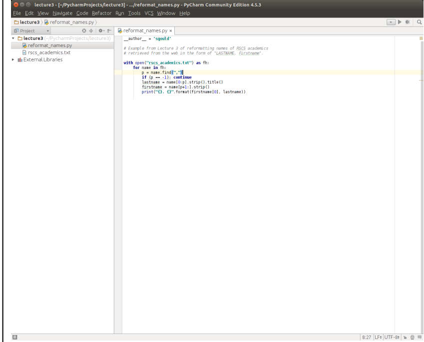


Figure 3.7: An empty PyCharm project and project with files already added and source code open.

into the project location (using standard operating system file copying) thereby adding it to the project. Next we will create a new Python file for writing our code: Select `File|New...` (or press Alt-Insert) and click "Python File". Give the file a name, say "`reformat_names.py`." Notice the `.py` extension indicating that the file will contain Python source code. The editor will automatically open the file for us to start editing.

Our third step is to write the code to perform the processing we want. Notice how PyCharm highlights keywords, strings and comments in different colours and styles. This is known as *syntax highlighting*, and many programmers find it easier to read code highlighted in this way. The editor also performs code completion and indicates pairs of matching brackets. This is particularly useful on the last line where we have two closing brackets.

Python Code

```
1  # Example from Lecture 3 of reformatting names of RSCS academics
2  # retrieved from the web in the form of "LASTNAME, Firstname".
3
4  with open("rscs_academics.txt") as fh:
5      for name in fh:
6          p = name.find(",")
7          if (p == -1): continue
8          lastname = name[0:p].strip().title()
9          firstname = name[p+1:].strip()
10         print("{}. {}".format(firstname[0], lastname))
```

Our last step is to test our code by running it. Either right-click on the code and choose "Run reformat_names" Ctrl-Shift-F10 from the context menu or select `Run|Run...` Alt-Shift-F10 from the main menu and choose "reformat_names". If you make further edits and want to run the code again you can simply press the Play button or hit Shift-F10. A panel will appear at the bottom of the PyCharm window with the programs output. The message "Process finished with exit code 0" tells you that the program finished normally, i.e., there were no errors.

Remember to save your edits often with `File|Save All` or Ctrl-S.

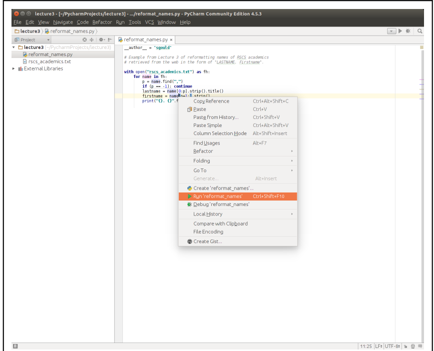
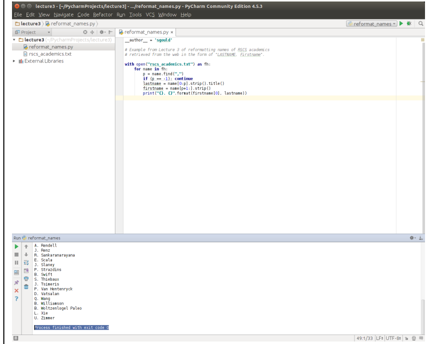
 

Figure 3.8: Running code in PyCharm and inspecting the output.

Using the Python console

As well as using the PyCharm IDE to write programs, the IDE can also let you "play" with code snippets through the *Python console*. Like the CodeBench environment that we use in this course, the console is a great place to experiment with code and play with the examples in these notes.

To start the Python console from within PyCharm select `Tools|Python Console...` from the main menu. A new panel will open up (usually at the bottom of the PyCharm window). Click anywhere within the panel to start interacting with the console. The console can be closed by clicking the red cross (or pressing Ctrl-F4) or hidden by clicking the Hide button at the top-right of the panel header (or pressing Shift-Escape). A hidden console retains it's state. **Tip:** the up and down arrow keys within the console window allow you to bring back previous statements.

Throughout these notes we will use examples of interactions with the Python console. An example of a console interaction is given below. Lines that are entered by the user always begin with `>>>`. Lines that do not begin with `>>>` show the value of the expression on the previous input line.

Python Console

```
>>> x = 10
>>> 2 * x
20
>>> 3 * x
30
```

In the above example a variable `x` was set to the value 10 on the first line. The second line of user input asks the Python interpreter to evaluate the expression `2 * x` (i.e., "2 times x"). The value of this expression is 20 and is shown on line 3. Similarly, line 4 shows the user entering the expression `3 * x` and its value, 30, is shown on the last line.

3.3.2 Filesystems and Operating Systems

Operating systems, such as Windows, Mac OS X and Linux, manage the interface between application programs and hardware resources. Part of an operating

system is a *filesystem* which organises how persistent data is stored on the computer. Filesystems for all major operating systems share a common hierarchy of files and directories (sometimes called folders). When you write software the source code is typically stored across multiple files within a directory for a given project. You should become familiar with navigating the filesystem for your operating system (either via a graphical interface or the command line).

Sandboxing

Because a software project lives within a directory on the filesystem you can have multiple copies of the same project on a single machine (revision control systems also allow you to be developing multiple concurrent branches of a project—more on this later). This allows you to explore ideas without worrying about affecting your main code development. For example, you may want to try changing an algorithm or generate a series of experimental results. If things work out you can merge your changes with the main source code. Otherwise you can simple delete the directory. This concept of being able to work on different copies of the code is known as *sandboxing*.

3.3.3 Code Versus Data

Many students new to programming get very confused about the distinction between code (i.e., programs) and data. Let's try dispel some of that confusion with the following points:

- *Source code* is a textual description of a program in some programming language. It is a mix of statements and comments that gets interpretted by the computer to run the program.
- Source code, being text, is different from other types of documents you may have come across, such as Word documents. The latter are typically *binary* files in some proprietary format that describe, among other things, how the document should be displayed. Text editors (or IDEs) are used to view and modify source code. They cannot typically be used to view and modify Word documents. Moreover, Word is not a good choice for viewing and modifying source code.
- Programs often operate on *input data*. This can be anything from a user-entered input to a set of files to an image or video stream. The program run on different data may produce different output, itself data.
- Programs are typically parameterised in some way. For example, you may be developing a WebApp and your software needs to know the URL where the application deployed. This information is called *configuration data* and differs from the input data that a program operated on. For small programs (where you have access to the source code) configuration data can be contained within the code (known as *hardcoding*). However, for large projects it's better practice to keep configuration data separate.
- Source code can sometimes be compiled into *byte code* or *machine instructions*, which is a binary representation of the program designed to be executed by the machine. In this sense code can be thought of as data.

3.3.4 Basic Revision Control in PyCharm

Revision control (or version control or source control) is a software engineering practice supported by a set of tools for managing software development. In particular, revision control systems keep track of changes made to source code, facilitate collaboration and integrate issue tracking. There are many different revision control systems but they all provide the same basic functionality, and many are integrated within the PyCharm IDE. The revision control system that we will be using in this course is called Git (`https://git-scm.com`).

Warning: Different revision control systems may use different terminology.

Central to a revision control system is a repository (or "repo" for short), which stores files, mainly source code, and keeps track of changes. Files are moved between the repository and a programmer's *working directory*. Git differs from many revision control systems in that it is distributed. This means that every programmer has his or her own copy of the repository. Usually a central remote repository also exists to allow programmers to collaborate on the same code base. Remote repositories also provide off-site backup, which is another important practice in software development. In this course the remote repository is provided by a framework called GitLab, which also includes issue tracking, wikis and a web portal for creating and managing projects. The portal can be accessed at `https://gitlab.cecs.anu.edu.au`.

Setting up Git in PyCharm

- Select `Configure|Settings` from the "Welcome to PyCharm" window. If you have already opened a project you can modify the settings using the `File|Default Settings` menu item.
- Select `Version Control > Git` from the Default Settings window, which should have just appeared.
- Click the `Test` button next to the "Path to Git executable" box. If the test fails make sure that Git is installed on your system and that the path is correct.

Working with Git Repositories

Creating. The first task to perform when working with revision control systems is to set up a repository. There are two basic ways to create a repository in GitLab.[2] The first way to create a repository is to *fork* an existing one (belonging to another user). This creates an new repository (in your account) by copying over files and history from the existing repository. Importantly, any further software development on the code in either repository will be independent of the other. In a later lecture we will discuss how multiple programmers can collaborate on the same repository.

In the GitLab web interface find the repository that you want to fork and click the "Fork" button. After forking you'll want to check the Visibility Level in the project settings to make sure it's Private (unless you want otherwise). By default GitLab sets the visibility level to be the same as the existing project.

The second way to create a repository is to start a new one on the remote server. Click the "New project" button on the GitLab desktop and enter information for new repository. Once a remote repository is forked or created from

[2]More advanced methods for creating repositories, such as from existing files, will be discussed in Lecture 12.

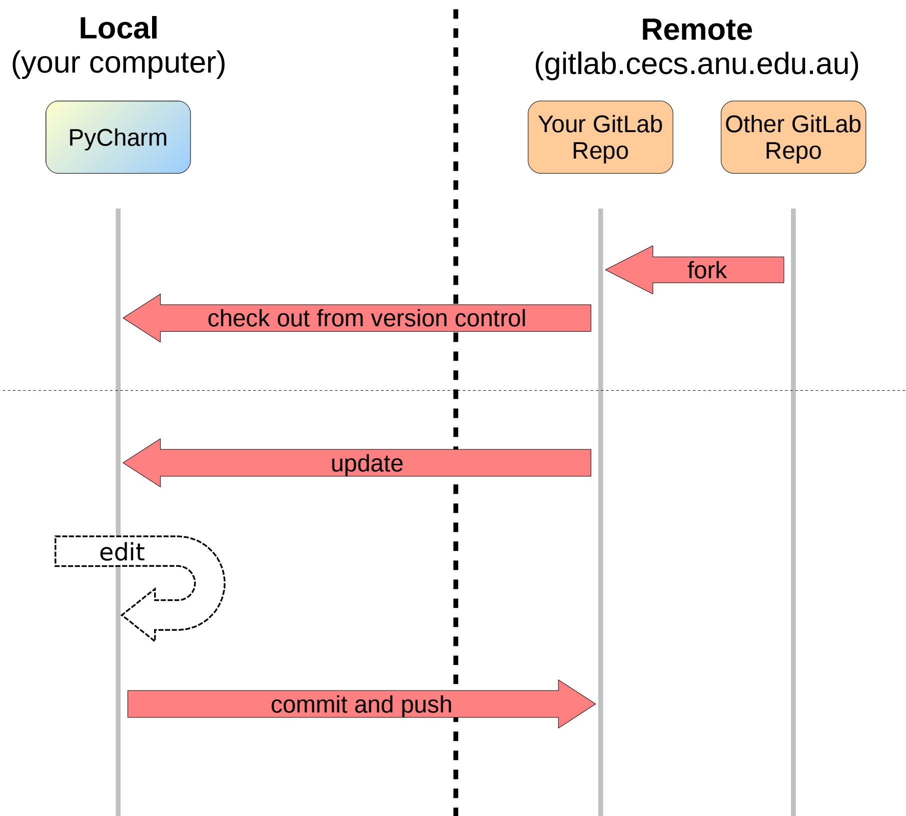


Figure 3.9: Overview of the most common operations using GitLab version control from within PyCharm.

scratch you will need to get a local copy in your working directory before you can add and modify files.

Cloning. The process of obtaining a local copy of a repository is known as *cloning*. In PyCharm select `Check out from Version Control` from the start screen or `VCS` menu, and choose `Git`. Enter the URL for the respository and the directory where you want the code to be stored locally. PyCharm will usually provide a suggestion for the local directory based on the repository name.

The URL for the Git repository can be found in GitLab and will have one of the following forms:

```
git@gitlab.cecs.anu.edu.au:<user>/<project>.git
```

for authentication via SSH (see below), or

```
https://gitlab.cecs.anu.edu.au/<user>/<project>.git
```

for password authentication. The latter is easier and the suggested method for this course.

Clicking "Clone" will connect to the remote server and pull down a local copy of the repository for you to start working. An illustration of this process and other common PyCharm version control operations is shown in Figure 3.9.

Editing. Once you have a cloned copy of the repository you can start editing code. Git automatically keeps track of changed files—when you add a new file to a project in PyCharm you will be asked if you want the file to be revision controlled. You can also add a file later by choosing the `Git|Add` (Ctrl-Alt-A) option from the `VCS` menu.

Changes you make will not be reflected in the repository until you explicitly *commit* them. This allows you to control how often changes are tracked. It also allows you to easily undo changes (remember the sandboxing concept). To compare your changes to the last commit (or any previous commit) select `Git|Compare with...` from the `VCS` menu. You can rollback your changes by pressing the "Revert" button (Ctrl-Alt-Z), but be warned that once you revert your changes will be gone.

Committing Changes. At some point you will want to update the remote repository with your changes. This is done by performing a *commit* followed by a *push* Ctrl-K. Whenever you commit you will be asked to provide a comment, or commit message, that describes the changes you made. It is a very good habit to provide meaningful descriptions. The GitLab web interface will show you what files have been updated and also provide a commit history.

The difference between **commit** and **push** will be addressed in a later lecture.

Updating Changes. If mutliple programmers are working on a project, or if your developing across different machines, you'll want to update your local copy of the code with changes pushed to the remote repository from other computers. This is known as a *pull*. In PyCharm choose `VCS|Update Project...` (Ctrl-T).

Sometimes Git will not be able to automatically merge changes from the remote repository if you have been editing the same part of the source code. This is known as a *conflict*. Dealing with conflicts and other more sophisticated use of git revision control, such as branching, merging and rolling back code across mutliple commits, will be covered in a later lecture.

Populating from Existing Files. If we've started a project in PyCharm like we did at the beginning of the lecture before creating a GitLab repository we

can still revision control the files. Instead of cloning the remote repository, we *import* the local code into the repository. Choose `VCS|Import into Version Control|Create Git Repository`. Select the directory which you want to add to revision control (this will usually be the location of your project) and click OK. Now manually add the files you want to keep in your repository. When you next *commit* and *push* your changes PyCharm will ask you for the URL of your remote repository (with the cryptic message "define remote"). Enter the same form of URL as you would when cloning (i.e., copied from GitLab).

Project Visibility and Membership. GitLab projects can be made public, private or internal (meaning that only users with an account on the system, i.e., ANU students and staff, can see them). In general, you will want to make your projects for this course private. You can change/check the visibility level from `Settings` menu for the project.

Another important concept is *project membership*. One of the great things about revision control is that it simplifies collaboration. Project members are individuals who all have access to the project files. Different membership types controls who gets what permissions (e.g., read-only). For now you and the course staff will be the only members of your projects. Later in the course, when you do group work, projects will have multiple members.

Setting up SSH Keys

Every time PyCharm performs a transaction with the remote repository (e.g., clone, push or pull) the interaction needs to be authenticated. The easiest way to do this is via usernames and passwords (PyCharm will cache your credentials so that you only need to enter them once). **This is the method we recommend in the course.**

An arguably more secure method of authentication is via a cryptographic protocol involving SSH keys. Briefly, a key pair is generated with one key (the private key) stored securley on your computer and the second key (the public key) stored on the server. Whenever Git needs to perform a transaction the keys are used to authenticate the user instead of needing usernames and passwords.

The difficulty with SSH keys is that they can be cumbersome to set up the first time (and each operating system has a slightly different way of generatring and managing keys). We provide an overview of the steps involved in setting up SSH keys here but recommend that you consult online documentation if you plan to use keys.

- Generate public and private keys (typically via `ssh-keygen -t rsa`)
- Log in to GitLab, choose Profile settings (gear icon), then SSH Keys
- Click "Add SSH Key" (see Figure 3.10)
- Give your key a descriptive name (e.g., "home computer") so that you can recognise it later. You can store multiple keys on GitLab if you work on multiple computers (e.g., desktop and laptop).
- Find where your public key is stored on your computer and copy it to GitLab. **Do not copy your private key.**
- Click "Add key"

Don't forget the `git@gitlab.cecs.anu.edu.au:<user>/<project>.git` form for cloning repositories if you are using SSH keys.

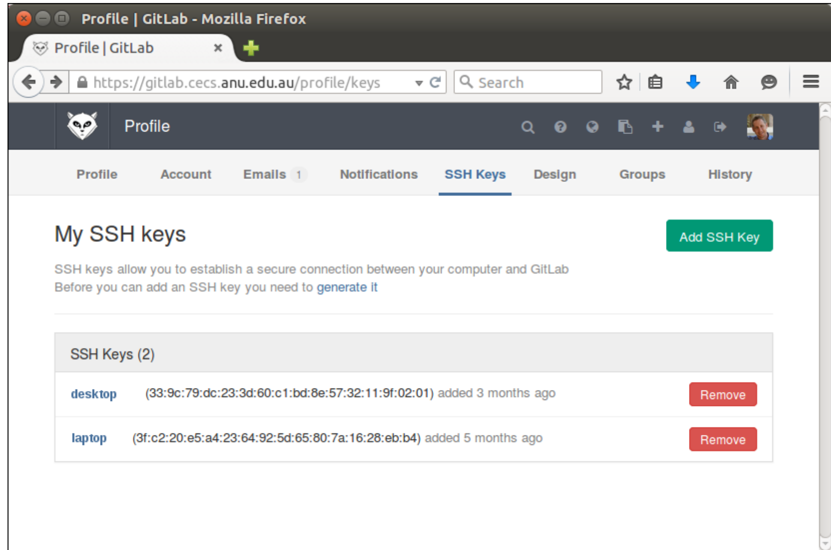


Figure 3.10: Screenshot of GitLab web interface showing SSH keys that are registered with the account. Assessible from `https://gitlab.cecs.anu.edu.au/profile/keys`.

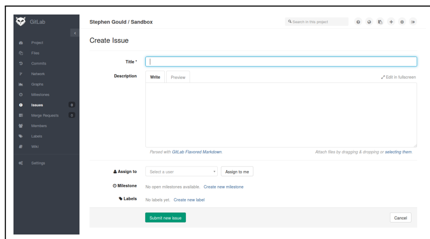
Dos and Don't s of Revision Control

Revision control is a powerful tool and it is good to get yourself into the discipline of using it for all your projects no matter how small and even if you're not collaborating with others. It may seem like a lot of overhead at first, but once you starting using revision control you'll find it saves a lot of time in the long run. Here are a few tips on using revision control effectively.

- Commit and push code regularly
- Pull remote changes before you start editing to help avoid conflicts
- Don't add very large data files to the repository, especially if they are not changing and can be sourced from elsewhere
  - However, do document where data you need for the project can be obtained from
  - and write scripts, which can be revision controlled, for automatically retrieving it
- Don't add files that can be regenerated from source
- Don't forget to add new files before committing

3.3.5 Issue Tracking

Software development is often done in teams. The software engineer testing a piece of software (and reporting bugs) is not necessarily the same person who wrote it. Even when they are the same (i.e., you find and fix your own bugs), there may not be time or resources to fix the bug immediately. Furthermore,



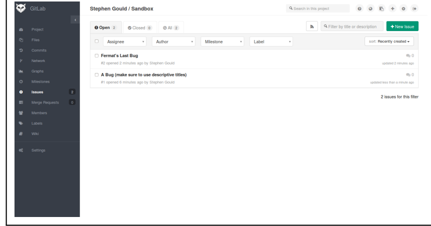


Figure 3.11: Issue reporting interface in GitLab. Left shows interface for reporting a new issue; right shows list of open issues.

even if the bug can be fixed straight away it's prudent to keep a record of bugs and solutions in case they reappear in the future—you will also want to write a regression test for the bug (more on this in a later lecture). For all these reasons *issue tracking* or bug tracking is an important part of the software development process.

In addition to a description of the bug, issues can record and many other aspects including who reported the issue, who is assigned to fixing it, the severity of the issue, and any milestones (e.g., software release dates) affected by the issue. As such issue tracking is not limited to bug fixing and can be used to manage feature requests and track project milestones.

GitLab has an integrated issue tracking and milestone system that lets to manage issues linked to a particular project. Figure 3.11 shows screenshots of the GitLab issue reporting interface.

3.3.6 Next Lecture

- Variables and expressions
- Data structures

3.4 Lecture 4: Data Structures I

Learning Outcomes

- Understand that variables are used to store (or reference) data in a computer program and that variables can have different types.
- Understand that variables exist within a scope and what happens when the contents of a variable are changed or copied.
- Understand that two variables can reference the same data.
- Appreciate conventions for naming variables and why descriptive names are important.
- Recognise and practice using basic containers: strings, tuples, arrays, and lists.
- Consider how problem data may be represented within a computer program.

Overview

This lecture revisits variables and fundamental datatypes (booleans, integers and floating-point numbers). Some basic data structures are then introduced, including strings, tuples, and lists. The lecture ends with a brief case study on representation and modeling.

3.4.1 Variables and Expressions

A *variable* is used to store or reference data. Unlike in mathematics where a variable can refer to an unknown quantity ("find x?"), in imperative programming the "value" of a variable must always be well defined.

A variable must be initialised (also known as defined or declared) before it is used. In Python a variable is defined by assigning a value to it—in some other programming languages you need to explicitly declare a variable before assigning to it. So, in Python,

```
my_counter = 0
```

both defines the variable `my_counter` and sets its value to zero. Likewise,

```
my_list = []
```

defines the variable called `my_list` and initialises it to be an empty list (more on lists and other data types later).

Once a variable has been defined we can use its value in expressions. So

```
print(my_counter)
```

will display the value stored in `my_counter` and

```
my_second_counter = my_counter + 1
```

will declare a variable called `my_second_counter` and set its value to whatever the value of `my_counter` is plus one.

We can also modify the value of an existing variable by assigning a new value to it. The old value of the variable is lost. Importantly, the expression on the right-hand side of the equal sign is evaluated *before* assigning to the variable on

the left-hand side. This allows the old contents of the variable to be used in assigning the new value. For example,

```
count = count + 1
```

increments a counter variable and is very common to see in programs.

Data Types

Associated with the data stored in a variable (or any object for that matter, e.g., a literal value in an expression or the return value for the entire expression) is a *data type*. This is very important because the data is interpretted differently depending on its type, and certain operations are only valid (or behave differently) depending on the type. For example, integers and character strings (words and sentences) are different types. It is possible to add two integers but it is not possible to add an integer to a string. Python allows you to add two strings with the result that the strings are concatenated.

Python Code

```
1 print(5 + 10)          # legal
2 print("5 + 10")        # legal
3 print("cat" + "dog")   # legal
4 print("cat" + 5)       # illegal
```

It should be clear from the above example that integers are specified by typing numbers whereas strings are surrounded by quotation marks (either single or double). Importantly, `5` and `"5"` are different types (the first is an integer and the second is a string) and behave differently. Furthermore, `5 + 10` is an expression that evaluates to `15` whereas `"5 + 10"` is a literal string.

Other fundamental data types are Booleans, which hold values `True` or `False`, and floating-point numbers (or floats). The latter are specified by including a decimal point so `5` is an integer whereas `5.0` is a float. Integers and floating-point numbers can behave differently during arithmetic. In Python 2, integer division would *floor* to the nearest integer (so `5 / 2` would evaluate to `2`). However, in Python 3 the behaviour was changed so that the result gets automatically upgraded to a floating-point number.

Sometimes we need to explicitly convert between data types. Here Python provides conversion functions. Most often this will be done to convert between numerical data types and strings. For example,

Python Code

```
1 answer = 42
2 sentence = "The answer is " + str(answer)
```

More sophisticated string formatting will be discussed later.

The `input` function used to obtain user input returns a string. So often you will see type conversion performed after obtaining some input. The conversion can be nested:

Python Code

```
1 n = int(input("Enter a number between 1 and 10: "))
2 if not 1 <= n <= 10:
3     print("The number you entered is not between 1 and 10")
```

Sometimes programmers like to add suffixes to variables to remind you of their type (e.g., `age_int` and `height_float`). Not everyone likes this practice.

Note that a variable can change the type of data that it refers to during execution of a program by assigning it a value of a different type. So the following is perfectly legal (in Python).

Python Code
```
age = input("Enter your age:")     # age holds a string
age = int(age)                     # age holds an integer
```

Swapping the Contents of Two Variables

It is often desireable to swap the contents of two variables so that at the end of the swap each variable holds the contents that used to be held by the other variable. A traditional way to do this is to introduce a new temporary variable.

Python Code
```
# swap contents of a and b
tmp = var_a
var_a = var_b
var_b = tmp
```

Python offers a convenient shortcut using multiple assignment. Here the right-hand side of the equal sign is evaluated before assigning to the variables on the left-hand side so the contents of the variables are correctly swapped.

Python Code
```
# swap contents of a and b
var_a, var_b = var_b, var_a
```

Modification Shortcuts

We saw above that we can modify the contents of a variable by assigning a new value to it. Moreover, the old contents of the variable can be used in an expression for defining the new value to be assigned. A very common operation is to increment or decrement a counter variable by some fixed amount. For example,

Python Code
```
up_counter = up_counter + 1
down_counter = down_counter - 1
```

Python (and many other programming languages) offers a shorthand way of using the variable on the left-hand side as the first operand of an expression on the right. This allows the above increment and decrement to be written as

Python Code
```
up_counter += 1
down_counter -= 1
```

The following table describes all the assignment operators.

Operator	Description
`=`	Simple assignment.
`+=`	Add and assign.
`-=`	Subtract and assign.
`*=`	Multiply and assign.
`/=`	Divide and assign.
`%=`	Modulus and assign.
`**=`	Exponentiate and assign.
`//=`	Divide, floor, and assign.

Table 3.3: Assignment Operators in Python.

Scope

Not all variables that have been defined are available everywhere within a program. There are good reasons for limiting the scope of a variable. It helps with code design and maintainability, and preventing data corruption. The *scope* of a variable is defined by the blocks of code (also namespace) where access to the variable is valid. We will see this when we discuss functions. When a variable goes out of scope it's contents are, in general, lost. A variable with *global scope* is accessible from anywhere.

Aliasing

We have been fairly informal in our discussion of variables so far by saying that a "variable stores data". Most of the time this way of thinking about a variable is fine. Technically, however, a variable is a symbolic name that references data (which exists separate to the variable name itself). This is an important distinction because it means that two variables can reference the same data, a situation called *aliasing*. Without understanding aliasing, your code could appear to have unexpected side effects.

Consider the following example.

Python Console

```
1 >>> x = 13
2 >>> y = x
3 >>> y
4 13
5 >>> y = 42
6 >>> y
7 42
8 >>> x
9 13
```

The assignment to variables `x` and `y` behaves as expected (our view of variables as storing data, in this case the numbers `13` and `42`). Now contrast the previous example with the following code, which modifies elements in a list. We discuss lists in more detail later in this lecture.

Python Console

```
>>> x = [1, 2, 3]
>>> y = x
>>> x
[1, 2, 3]
>>> y
[1, 2, 3]
>>> y[0] = 42
>>> y
[42, 2, 3]
>>> x
[42, 2, 3]
```

Surprisingly, modifying the first element of the `y` list results in a change in the `x` list. What is going on here? The answer is aliasing. Both symbolic names (variables) `x` and `y` reference the same underlying list (data), and the code modifies that underlying list. The former code, on the other hand, changed the *data that variable* `y` *references*.

The behaviour that was probably intended and perhaps more intuitive, i.e., modifying a different list stored in `y`, is demonstrated by the following code. Note the invocation of the `.copy()` method on Line 2.

Python Console

```
>>> x = [1, 2, 3]
>>> y = x.copy()
>>> x
[1, 2, 3]
>>> y
[1, 2, 3]
>>> y[0] = 42
>>> y
[42, 2, 3]
>>> x
[1, 2, 3]
```

Side effects caused by aliasing happen frequently when passing variables into functions or assigning a container data type to another variable without the `.copy()` method. We will cover this topic again when we discuss functions in a later lecture.

3.4.2 Strings

A string is a special data type that is used to store a sequence of characters. In Python strings are of type `str`. Strings can be of any length (including empty) and are defined by enclosing a sequence of characters in matching single or double quotes.

Python Code

```
string_A = "Hello World"  # valid string
string_B = 'Hello World'  # valid string, same as string_A
string_C = "Hello World'  # invalid string, mismatching quotes
```

Sometimes you will want to add special characters to a string. Most of the time you can just type the character, but what happens when you want to include a quotation mark within a string? One solution is to use single quotes

to delimit a string that includes a double quote (e.g., `'"Hello World"'`) and double quotes to delimit a string that includes a single quote (e.g., `"Alice's restaurant"`). However, you can probably see that this solution has its limitations.

The alternative, adopted by all programming languages, is to use an *escape character*, usually a backslash (\), that changes the meaning of the character following. So we can now write `"\"Hello World\""` and `'Alice\'s restaurant'`. If we want to include an actual backslash in our string we need to escape the backslash itself (e.g., `"\\"`). Special non-printing characters can also be included in this way. The two most common of these are the newline character `\n` and the tab character `\t`.

String processing is something that computers are very good at, and a great deal of programming concerns the manipulation of string. We will see lots of examples and devote an entire future lecture to string processing. For now, we cover some basic printing of strings.

The simplest way to display information to the user is to print it to the screen. Strings, integers, floats, and other objects can all be output using the print function.

In Python 3 **print** was changed from a keyword, part of the language, to a function.

Python Code

```
print("The Craft of Computing")
print(1040)
```

A more sophisticated way to display output is by string formatting. Technically, `format` is a method called on a string that modifies the string. For example,

Python Code

```
import math
print("The number pi is {:.2f}".format(math.pi))
```

prints the string "The number pi is 3.14". Here the format specifier ":.2f" takes the first argument of `format`, interprets it as a float, and displays it to two decimal places. Some other formatting examples are shown below:

To print braces within a format string double them up, i.e., {{}}.

Python Code

```
print("{} and {}".format("Bart", "Lisa"))
print("{0} and {1}".format("Bart", "Lisa"))
print("{1} and {0}".format("Bart", "Lisa"))

for i in range(100):
    print("{:4d}".format(i))
```

Individual characters in a string can be accessed using square brackets, `[]`, also known as the **index operator**. The following code prints out a string one character at a time.

Python Code

```
message = "Hello World"
for i in range(len(message)):
    print(message[i])
```

Note that the first character in the string has index 0. A negative integer within the square brackets indexes the string from the back, so `message[-1]` accesses the last character in `message`. Other patterns are possible and will be discussed in Lecture 7.

3.4.3 Basic Data Structures

Python (and many other modern programming languages) have a rich set of data structures that can help store objects and solve problems. In this section we discuss two of the most basic data structures: tuples and lists.

Tuples

A *tuple* is an immutable fixed-length ordered collection of elements. The elements of a tuple can be arbitrary objects of different types. A tuple is specified as a comma separated sequence of expressions enclosed in pathentheses (round brackets). The following example constructs a 2-tuple representing a particular location on the unit circle:

Python Code

```
import math
theta = 2.0 * math.pi * 45.0 / 360.0     # 45 degrees in radians
point = (math.cos(theta), math.sin(theta))
```

We can also use the `tuple` keyword to cast from another data type, such as a string, to a tuple. For example,

```
tuple("COMP1040")
```

produces the 8-tuple

```
('C', 'O', 'M', 'P', '1', '0', '4', '0')
```

You can access invidual elements in a tuple using the index operator (square brackets) in much the same way you would access individual characters in a string. Remember that Python uses zero-based indexing—getting indexing wrong is a very common source of error in programs. So in the example above `point[0]` is the first element of the tuple stored in variable `point`.

Lists

A *list* is a mutable ordered collection of objects. Unlike a tuple a list can grow and shrink in size after it has been created. You can also change the value for elements in the list at any time. A list is specified as a comma separated sequence of expressions enclosed in square brackets. Lists can also be built up incrementally as the following example demonstrates.

Python Code

```
names = ["Bart", "List"]    # construct a list with two elements
names.append("Maggie")      # add a third element
print(names)                # print the list
```

One really useful way to create lists in Python is via *list comprehension*. A list comprehension builds a list by executing some operation over elements of

another sequence. The following example constructs a list whose elements are the first 10 perfect squares:

Python Console

```
>>> perfect_squares = [(x + 1) ** 2 for x in range(10)]
>>> perfect_squares
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

And here is another example of list comprehension:

Python Code

```
fullnames = [n + " Simpson" for n in names]
print(fullnames)
```

We will discuss iteration and show more examples of list comprehension throughout the course. For now, it is enough to recognise the basic pattern. Now let's put strings, tuples, and lists together in an example program.

Example 3.4.1. An *anagram* is a word or phrase that can be made by rearranging the letters of another word or phrase. The meaning of the phrases do not need to be related, but it can be amusing when they are. For example, "listen" is an anagram of "silent" and "a decimal point" is an anagram of "I'm a dot in place".

In this case study we consider the problem of finding Twitter anagrams, i.e., two tweets that are anagrams of each other. For simplicity we will ignore non-alphabetic characters. So given two tweets (of at most 140 characters) how would we go about determining if they are anagrams?

One approach would be to pre-process the tweets to remove non-alphabetic characters, sort the letters in the resulting strings, and compare them. We will call the final sorted string a signature for the tweet. This seems like a lot of work (but can actually be done in a few lines of Python). Let's have a look at code for computing a signature from a tweet.

Python Code

```
import string

tweet = "Everything is always too good to be true!"
letters_only = ''.join(filter(str.isalpha, tweet))
lower_case_only = letters_only.lower()
sorted_lower_case = sorted(lower_case_only)
signature = ''.join(sorted_lower_case)
```

The code can be written in a more compact way by feeding the output of one step directly into the next step. Here is the code for comparing two tweets.

Python Code

```
import string

tweet_A = "Everything is always too good to be true!"
tweet_B = "I guess I have to try to go to bed early now."

sig_A = ''.join(sorted(''.join(filter(str.isalpha, tweet_A)).lower()))
sig_B = ''.join(sorted(''.join(filter(str.isalpha, tweet_B)).lower()))

if (sig_A == sig_B):
    print("The tweets are anagrams")
else:
    print("The tweets are not anagrams")
```

This code is pretty good, but if we are searching through a very large number of tweets it may be inefficient. Why?

An alternative is to represent a tweet (i.e., its signature) as a histogram of letters. The histogram only needs to be 26 elements long (and each element ranges between 0 and 140). We then compare histograms. Here is the code.

Python Code

```
import string

lower_case_A = ''.join(filter(str.isalpha, tweet_A)).lower()
sig_A = [lower_case_A.count(i) for i in string.ascii_lowercase]

lower_case_B = ''.join(filter(str.isalpha, tweet_B)).lower()
sig_B = [lower_case_B.count(i) for i in string.ascii_lowercase]

if (sig_A == sig_B):
    print("The tweets are anagrams")
else:
    print("The tweets are not anagrams")
```

The alternative signature is much smaller than in our first approach. Can you think of an even better representation for comparing a very large number of tweets? Hint: consider the frequency of letters in English.

Like strings, individual elements of a list can be accessed using the index operator `[]`. More complex patterns that address multiple elements within a list, known as *slices*, are also possible and will be discussed in Lecture 7.

3.4.4 Representation and Modeling

The basic data structures just discussed come up all the time in solving problems with programming. They are not simply useful containers for storing data. We can also exploit their properties to make solving a problem easier, and will see plenty of examples of this when we look at more data structures in a later lecture. However, data storage is only one issue we face when solving problems. Other aspects of representation and modeling are also very important as the following example shows. Plenty of practice and experience will help you to develop intuition into what representations and data structures work best for different sorts of problems. This is a big part of the craft of computing.

Example 3.4.2. Consider the famous *Birthday Paradox*, which states that in a small gathering of just 23 people there is 50% chance that two of the people share the same birthday (ignoring the year). Now, suppose we wish to write a program to experimentally verify the Birthday Paradox. We will run repeated trials. In each trial we randomly sample 23 birthdays and then check whether any two birthdays within the trial are the same.

We are left with one key question—how do we sample a birthday? One way would be to represent a birthday as day and month (i.e., `DD-MM`) and first sample the month followed by the year. This has two issues both due to the fact

that months contain a different number of days. First, there is not a uniform probability of being born in any month. Second, once we sample the month we need to sample the day knowing the number of days in the month. Ignoring the first issue (and for simplicity also ignoring leap years) we end up with the following code where we store birthdays in a *list* of *2-tuples*.

Python Code

```
import random
month_days = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

number_of_trials = 1000
same_birthday_count = 0

for trial in range(number_of_trials):
    birthdays = []
    for i in range(23):
        month = random.randint(1, 12)
        day = random.randint(1, month_days[month - 1])
        birthdays.append((day, month))

    if (len(set(birthdays)) < 23):
        same_birthday_count += 1

p = float(same_birthday_count) / float(number_of_trials)
print("probability of same birthday is " + str(p))
```

Here we check for duplicate birthdays by converting the list of birthdays into a set. Since a set discards repeated elements it's size will be less than 23 if duplicate birthdays were sampled. Thus all we need to do is check the size of the resulting set.

An alternative is to represent birthdays in terms of day-of-year. This representation is much better for the problem at hand and lets us write much more succinct code (where again we have ignored leap years).

Python Code

```
import random
number_of_trials = 1000
same_birthday_count = 0

for trial in range(number_of_trials):
    birthdays = [random.randint(1, 365) for i in range(23)]
    if (len(set(birthdays)) < 23):
        same_birthday_count += 1

p = float(same_birthday_count) / float(number_of_trials)
print("probability of same birthday is " + str(p))
```

Thinking about representation is very important both for ease of implementation and maintenence, and correctness of your code. A very famous bug known as the *Y2K Bug* was caused by an inappropriate date representation. Many programs written before the year 2000 used two digits to represent the year, for example, "97" represented the year "1997". In the early 1990s programmers realised that this was going to cause major errors with possible catastrophic

consequences as the millenium approached and passed. An enormous amount of effort and dollars went into re-writing programs and patching large computer systems to avoid the bug. The Y2K Bug demonstrated the importance of a good representation especially when code is used in large scale systems, varied applications, and over long time periods.

3.4.5 Next Lecture

- Execution models
- Control flow
  - Conditional execution
  - Loops
- Catching runtime errors

3.5 Lecture 5: Execution Models and Control Flow

Learning Outcomes

- Develop a mental model of program execution and be able to mentally step through the execution of a small piece of code.
- Understand that certain program instructions can control what code gets executed depending on the result of some test or state of the system.
- Recognise different control flow patterns, how they map onto language constructs, and when to use them.
- Be able to write simple programs in `python` using `if-then-else`, `for`, and `while` constructs.

Overview

In this lecture we develop the idea of a mental execution model and practice stepping through the execution of small programs. We introduce the idea of conditional execution and repeated execution, and see how they are expressed in Python using `if-then-else`, `for`, and `while` constructs. We also briefly discuss function calls and language constructs for catching runtime errors.

3.5.1 Mental Execution Model of Programs

It is important to develop a mental model of how programs run and to be able to manually step through the execution of a small piece of code to understand what it is doing. Sometimes a graphical visualisation of how code runs can help. The website `http://pythontutor.com/` is an online python tutor developed by Philip Guo that does just that.

In an earlier lecture we mentioned that a program is a sequence of instructions that get interpretted by a computer to achieve some computational task. Some instructions will get *executed* by the program everytime it is run (on all different inputs). But often, to obtain interesting outcomes, some instructions are only executed if certain conditions are met, and other instructions otherwise. This is known as conditional execution.

Let's take as an example a small piece of code for computing the absolute value of a number. Note that Python includes the built-in function `abs` so you would never actually have to write code for this, but it makes a useful example.

Python Code

```
user_input = input("Enter a number: ")
x = float(user_input)
if (x < 0):
    x = -x
print(x)
```

Executing this code in our heads we see that the first line of code prompts the user to enter a number and stores what they type in a variable called `user_input`. The second line converts the user input from a string into a floating-point number and stores the result in a variable called `x`. So if the

user had entered 10 the variable x would now contain the floating-point value 10.0. The next line is where things get interesting. The code checks whether in value stored in x is less than zero. On our example input 10 the test fails and the sequence of indented statements (here only one) that follows the test is skipped. The program proceeds to the final line and prints the value stored in x, namely 10.0.

Now, let's mentally run through the program again but this time enter -10 instead of 10. This time when we get to the negativity check on the third line we will have the value -10.0 stored in x. The test succeeds and the program proceeds to run the indented statement. This statement negates the value in x and assigns the result back to x. The final line of the program is then executed printing out the value 10.0.

Before showing you the code we explained that it would calculate the absolute value of a number. Our mental execution of the code on two example inputs verified that it does this. As always we could have written different code that achieves the same outcome. An alternative is shown below.

Python Code

```
user_input = input("Enter a number: ")
x = float(user_input)
if (x < 0):
    print(-x)
else:
    print(x)
```

And another.

Python Code

```
user_input = input("Enter a number: ")
x = float(user_input)
print(-x if (x < 0) else x)
```

In English we would describe the code above as: *A string is inputted from the user and converted to a number. The program then negates and prints the number if it is negative or just prints out the number otherwise.*

We cover conditional execution and other forms of control flow below. For now, determine what the following program does by mentally executing the instructions on different input values.

Python Code

```
n = int(input("Enter a positive integer:" ))
if (n % 2 == 0):
    n = n / 2
else:
    n = 3 * n + 1
```

The calculation done in the above code is the core calculation done to produce a sequence of numbers known as a Collatz Sequence—you get the next number in the sequence by feeding the result back into the program. Try running the program repeatedly entering in the result of the previous run. Does the sequence of numbers you get always end in an ever repeating cycle of 4, 2, and 1? Later in the lecture we will see how to automate running the program repeatedly.

3.5.2 Control Flow

Control flow refers to the order in which a program executes statements. Without control flow the program will just execute statements in the order in which they appear in the code. Special instructions allow different paths through the code to be followed. This may include:

- jumping to a different location in the code,
- executing a set of statements only if some condition is satisfied,
- repeatedly executing a set of instructions until some condition is met,
- halting execution of the program.

Logical (or Boolean) expressions are used to determine if some condition is met. A logical expression returns a **Boolean** value (i.e., a value that is either `True` or `False`). For example, if we want to test whether the value stored in a variable is positive we could write

In some programming languages a value of zero is considered `False` and a non-zero value is considered `True`.

```
my_var > 0
```

This is an expression that returns either `True` or `False`. The return value can be assigned to a variable or used within a control statement as we will see shortly.

One thing we need to be careful about is testing for equality. In Python (and many other programming languages) the equals sign (`=`) is used for assignment and two equals signs together (`==`) is used to test equality. Consider the following two lines of code

```
var_a = var_b
var_a == var_b
```

The first line is a *statement* that assigns the value in `var_b` to `var_a` while the second line is an *expression* tests whether the value stored in `var_b` is equal to the value stored in `var_a`. When used in a control statement, Python will warn you if you use `=` instead of `==`. This is because assignment does not return a value. Some other programming languages are not so friendly and accidentally using `=` is a frequent source of difficult to detect bugs.

Other useful operators for testing various conditions are:

Operator	Evaluates to `True` if ... otherwise `False`
in	the element (left) is a member of the container (right)
not in	the element (left) is not a member of the container (right)
is	the left and right operands point to the same object
is not	the left and right operands do not point to the same
`<`	the left operand is less than the right operand
`<=`	the left operand is less than or equal to the right operand
`>`	the left operand is greater than the right operand
`>=`	the left operand is greater than or equal to the right operand
`<>`	the left operand is not equal to the right operand
`!=`	the left operand is not equal to the right operand
`==`	the left operand is equal to the right operand

Table 3.4: Comparison operators.

Boolean expressions can be strung together using the logical operators `and`, `or`, and `not` to form more complicated tests. Use parentheses to explicitly

control the order in which operators are evaluated within an expression. For example, the following two lines of code give different results for particular combinations of Boolean variables `a` and `b`.

```
not a or b
not (a or b)
```

Python allows comparison operators to be *chained*. This is quite unique and is not a feature of other programming languages. The most common place to find operator chaining is when checking bounds on a variable. For example,

```
0 < my_int < 10
```

which is logically equivalent to

```
(0 < my_int) and (my_int < 10)
```

If-Then-Else

The `if-then-else` construct is perhaps the most widely used form of conditional execution. We have seen this construct a number of times already. The `else` part is optional. A useful extension is to include optional `elif` blocks which implements `switch` or `case` statements found in other programming languages. Consider the following code snippet.

Python Code

```
day_of_week = int(input("Enter day of the week (1-7): "))
if (day_of_week == 1):
    print("Monday")
elif (day_of_week == 2):
    print("Tuesday")
elif (day_of_week == 3):
    print("Wednesday")
elif (day_of_week == 4):
    print("Thursday")
elif (day_of_week == 5):
    print("Friday")
elif (day_of_week == 6):
    print("Saturday")
elif (day_of_week == 7):
    print("Sunday")
else:
    print("<unknown>")
```

Can you think of a better way of writing this particular piece of code?

Note our use of indentation in the example code above. The indentation not only makes the code more readable, it is required by Python to define the block of code that gets executed (or skipped) if the condition above is satisfied (or not). Other programming languages use delimiters to mark blocks of such code. A sequence of statements at the same level of indentation is considered a block and can be thought of as a unit of code that is always executed together.

In Python a block of code at the same indentation level is called a **suite**.

Note that the order in which we test inputs is important in that the individual conditions need not be mutually exclusive. As soon as a condition is satisfied the corresponding code block is executed and all remaining code blocks are skipped. So the following two code snippets produce different results.

```
if (x % 2 == 0):
    print("x is even")
elif (x < 0):
    print("x is negative")
else:
    print("odd and positive")
```

```
if (x < 0):
    print("x is negative")
elif (x % 2 == 0):
    print("x is even")
else:
    print("odd and positive")
```

Another example that produces different results to the two snippets above but has the same flavour is:

```
if (x < 0):
    print("x is negative")
if (x % 2 == 0):
    print("x is even")
if not ((x < 0) or (x % 2 == 0)):
    print("odd and positive")
```

For Loops and Iteration

Often we wish to perform some operation on each object in a collection. For example, we may want to iterate over a list of assignment grades and count the number of high distinctions. The `for` loop is ideal for exactly this scenario.

Python Code

```
grades = [65, 90, 70, 85, 92, 73, 62, 68, 81, 68]

num_high_distinctions = 0
for g in grades:
    if g >= 85:
        num_high_distinctions += 1
```

The way to read the `for` loop above is as follows: "*for each grade denoted by `g` in the list of grades check if its value is greater than or equal to 85. If so, increment the count of number of high distinctions.*"

The `for` construct can also be used to create a loop that runs for a fixed number of iterations. We do this in Python by using the special `range` function. Other programming languages have other ways to define looping for a given number of iterations. A simple example illustrates the point.

In Python 2, `range` returned a list. In Python 3, it returns an iterator.

Python Code

```
for i in range(10):
    print(i)
```

One thing to be aware of is that the value of the iterator variable goes from 0 to $n-1$ where n is the argument provided to the `range` function. So in the code snippet above the number 10 is not printed. The function can take other arguments that specify the starting value of the iterator and the amount by which it increments on each iteration as demonstrated in the following code which prints out the even numbers between zero and nine (inclusive).

Python Code

```
for i in range(0, 10, 2):
    print(i)
```

When a loop finishes the program continues executing immediately instructions following the loop.

Remember that indentation defines a code suite, so the following two code snippets that sum numbers from 1 to 10 have very different behaviour.

```
sum_x = 0
for x in range(0, 10):
    sum_x += x
print(sum_x)
```

```
sum_x = 0
for x in range(0, 10):
    sum_x += x
    print(sum_x)
```

Example 3.5.1. A substitution cipher is one of the earliest forms of cryptography (now trivially broken) in which a plaintext message is converted to ciphertext, i.e., encrypted, by substituting letters via a lookup table (e.g., "A" is substituted for "M", "B" for "K", etc). The ciphertext is decrypted by performing the reverse substitution. The forward and reverse lookup tables are the encryption and decryption keys.

Assume our encryption key is a lookup table with one entry for each letter of the alphabet (i.e., of length 26). Each entry tells us which letter the corresponding location gets mapped to, so for example, `encryption_key[0]` is the letter that should replace "A" in our ciphertext—remember that indexing in Python is zero based! We can easily construct a random encryption key as follows:

Python Code
```
1 import random
2
3 encryption_key = list(range(26))
4 random.shuffle(encryption_key)
```

Constructing the decryption key requires that we build a reverse lookup table. This is a common coding pattern that you may use often.

Python Code
```
1 decryption_key = list(range(26))
2 for i in range(26):
3     decryption_key[encryption_key[i]] = i
```

Now let's say we have an uppercase letter stored in variable `letter` and we wish to find the substitute for that letter. We need to be able to index our lookup tables starting from zero for "A". Fortunately, the letters "A" to "Z" are encoded consecutively in all character encoding schemes (specifically, UTF-8) and Python provides us with a function called `ord()` which returns the ordinal value for a single character. The function `chr()` takes an ordinal value and gives us back the character. Finding the letter to substitute for is then easy:

Python Code
```
1 new_letter_index = encryption_key[ord(letter) - ord('A')]
2 new_letter = chr(new_letter_index + ord('A'))
```

Given an arbitrary plaintext string we can encrypt it by just iterating over each letter in the string and finding it's substitute.

Python Code

```
ciphertext = ''
for ch in plaintext.upper():
    if str.isalpha(ch):
        ciphertext += chr(ord('A') + key[ord(ch) - ord('A')])
    else:
        ciphertext += ch
```

Note that we did not encrypt non-alphabetic characters. The decryption code is identical except that we use the decryption lookup table instead.

While Loops

Rather than iterating over a collection of objects, if we want to repeatedly run a certain segment of code until some condition is violated (not satisfied) then we can use a `while` loop.

Remember the Collatz Sequence we introduced earlier in the lecture? Instead of us having to manually re-run the code each time a new number in the sequence is calculated, we can write a `while` loop to re-run the code for us. The loop terminates when the sequence reaches the value 1.

Python Code

```
n = int(input("Enter a positive integer:" ))

# check that a positive integer was actually entered
if (n <= 0):
    exit()

# keep generating the next integer until we reach n == 1
print(n)
while (n != 1):
    if (n % 2 == 0):
        n = n / 2
    else:
        n = 3 * n + 1
    print(n)
```

Nesting Loops

Loops can appear within other loops. This is called *nesting*. For example, the code snippet below shows two nested loops (for printing the twelve times table). The inner loop (over variable `j`) is executed in its entirety for each iteration of the outer loop (over variable `i`). This is a classic example of an $O(n^2)$ algorithm.

Python Code

```
for i in range(1, 13):
    for j in range(1, 13):
        print("\t", i * j, end="")
    print("\n")
```

Breaking Out of a Loop Early

Programmers are lazy and sometimes it is cumbersome to explicitly write out the termination condition for a loop. In the latter case code may be more readable if we allow a loop to be exited from some arbitrary point within it. It only makes sense to execute a `break` statement conditionally (i.e., within an if-then construct).

An example may help to illustrate the point. The following code asks the user to enter a list of positive numbers. A loop is used check that all numbers in the list are positive.

Python Code

```
num_list = [int(v) for v in input("Enter list:").split(' ')]
all_positive = True
for n in num_list:
    if (n <= 0):
        all_positive = False
        break
```

Note, that for nested loops the `break` statement will jump out of the smallest enclosing loop. In order to break out of nested loops you will usually need to set a flag or re-write your code to make loop termination easier.

Returning to the Start of a Loop

Another useful construct for controlling program flow is to return to the start of a loop early, i.e., before executing all of the statements inside the loop. Whenever Python encounters a `continue` statement it immediately begins the next iteration of a loop (or exits the loop if already on the last iteration).

The following code iterates through every file in the current directory. If the file has extension "`.txt`" the code opens the file and prints out the first line.

Python Code

```
import os
for filename in os.listdir("."):
    if (filename[-4:] != ".txt"):
       continue

    print(open(filename).readline())
```

The Pass Statement

Python includes a special statement called `pass`, which essentially means "do nothing". This can be very useful when developing code because you often wish to focus on the structure of the code and then fill in the details later. Since Python requires that all loop bodies, if statements and functions be non-empty, `pass` allows you to construct code that is syntactically correct but does nothing. For example,

Python Code

```
for student in student_list:
    # TODO: work out details later
    pass
```

Function Calls

We have already seen many functions. They are indispensable constructs in facilitating code reuse and making software more readable and maintainable. Functions are probably the most powerful concept in a programming language. When a function is called, the code within the function is executed and then, once the function completes, control returns to the next instruction from where the function was called.

Functions can take input parameters and return values (or objects). If a mutable object is passed as an input parameter to a function then the function can modify the contents of that object. Details on defining functions will be presented in a later lecture. For now it is important to be able to use functions and recognise when they are being called within a piece of code. A classic example is the `input` function, which waits for user input then returns execution to the calling program.

Functions need not explicitly return a value, e.g., the `print` function. In such cases Python deems the function to have returned the special object `None`.

Python Code

```
1 msg = input("Enter message:")
2 print("You entered " + msg)
```

Function calls can be nested. That is, a function can call another function including itself—this called recursion and is an advanced programming concept that will be treated in a later lecture.

3.5.3 Catching Runtime Errors

We have seen before that runtime errors can cause programs to crash (in the best case) or produce the wrong output without you knowing (in the worst case). Sometimes it's okay for your program to crash, but often you will want it to handle errors gracefully.

One way to handle runtime errors is to try anticipate every way in which your program can go wrong and guard against each way (it is a useful exercise to do this). This is fine if you can deal with the error locally but sometimes the error occurs deep within your code and you want to handle it somewhere higher up. Python provides a signalling mechanism called **exceptions** that allows runtime errors to "bubble up" your code until they are caught and handled by an exception handler.

Try-Except Construct

The `try-except` construct (sometimes called try-catch) can be used to intercept exceptions and perform corrective actions rather than letting the program crash. Sometimes the same effect can be achieved using conditional execution but `try-catch` is more general and often easier to write.

```
if (denom != 0.0):
    result = num / denom
else:
    print("divide by zero")
```

```
try:
    result = num / denom
except ZeroDivisionError:
    print("divide by zero")
```

In general the code between `try` and `except` can be anything and the offending code may be multiple function calls deep.

An exception in Python can be triggered by the `raise` keyword.

```
try:
    run_some_code()
except:
    handle_exception()
```

A very common use of exception handling is to recover from errors when reading from a file (that possibly doesn't exist).

Python Code

```
fh = None

try:
    fh = open("data_file.txt", "r")
    for line in fh:
        print(line)

except:
    print("error reading from file")

finally:
    if fh:
        fh.close()
```

There are many more options and subtleties in raising and dealing with exceptions which we will not go into in this course. Suffice to say that knowing about exceptions will help you understand when your code fails and programming defensively is a good strategy for preventing incorrect results.

3.5.4 Control Flow Summary

The following table summarises the five basic control flow mechanisms discussed in this lecture.

if-then-else	if <condition>:
	    <do something>
	elif <condition>:
	    <do something else>
	else:
	    <do yet something else>

for loop	for <variable> in <iterable>: <do something>
while loop	while <condition>: <keep doing something>
function call	<variable> = <function call> <function returns here>
try-catch	try: <do something> except: <handle error> finally: <always do this>

3.5.5 Next Lecture

- Data formats
- Reading and writing files

3.6 Lecture 6: Data Formats and Files

Learning Outcomes

- Know that persistent data is stored in files on computer systems.
- Understand the distinction between text and binary data, and that binary data (such as images, audio, and video) need to be encoded before stored in a file.
- Understand the separation of code and data.
- Recognise some standard file formats including HTML and CSV.
- Be able to read and write text files in Python.

Overview

In this lecture we discuss different file formats. We talk about parsing files (particularly text files) to extract information needed in a data processing context, and show how this can be done in Python. We also discuss markup languages and show how they can be created programmatically, which is useful for a number of tasks. We briefly touch on binary file formats used for encoding audio, images, and video.

3.6.1 Text versus Binary Files

A **file** is a sequence of bytes (data) that persists on a disk, or more correctly, within a filesystem. For example, source code for a Python program is stored in a file, as are PDF documents, Powerpoint presentations, email attachments, digital photographs, etc. Files can be created, read, written, and modified.

An important distinction between files is whether they are text or binary. Text files are human-readable—they contain letters, numbers and symbols. Formally, data in a text file is encoded in some unicode character set, which defaults to UTF-8, which controls how the characters are displayed. But, in general, you can "look" into a text file and read it's contents.

A binary file, on the other hand, uses some other encoding scheme. If you just "look" at the contents of a binary file it will appears as random characters. The reason for storing data in binary files is that it often results in a more efficient format for both storage and processing. The downside is that special-purpose code is required to read, modify and write the data (although this is probably the case anyway for lots of binary data like images and video).

Reading and Writing Text Files in Python

Python interacts with files through a *file handle* or *stream*. In order to read or write a file you first need to open the file in a particular mode (read or write). This creates a file handle, which you can use for subsequent operations on the file. After you have finished working on a file you should close the file handle. Multiple files can be open simultaneously.

The following code snippet reads the entire contents of a file into a string variable. We assume the file is called "foo.txt" and is in the same directory where our code is running.

Python Code

```
fh = open("foo.txt", "r")
file_contents = fh.read()
fh.close()
```

However, a more typical scenario would be to read a file line by line as shown in the next code snippet, which reads lines from a file and printing them on the screen. Of course, you would usually do some more interesting processing of the file contents.

Python Code

```
fh = open("foo.txt", "r")
for line in fh:
    print(line, end="")
fh.close()
```

Note that in our code snippet above we added `end=""` to our print statement. This stops Python from printing an additional newline character at the end of each line since one already exists within the string read from the file (and stored in variable `line`).

Writing a file is just as simple:

Python Code

```
fh = open("bar.txt", "w")
for i in range(10):
    fh.write(str(i) + "\n")
fh.close()
```

Files can be opened in various different modes as summarised below.

Mode	Description
`r`	Opens a file for reading. Raises an exception if the file does not exist.
`w`	Opens a file for writing. Clears contents of existing files.
`a`	Opens a file for appending. Contents of existing files are retained.
`r+`	Opens a file for reading and writing. Written data overwrites existing.
`w+`	Opens a file for reading and writing. Existing files are cleared.
`a+`	Opens a file for reading and writing. Written data is appended.

Table 3.5: Different modes in which a file can be opened.

Back to our Lecture 3 example of wanting to reformat names from an input file `rscs_academics.txt`, we can now produce a new output file containing the reformatted names as follows

Python Code

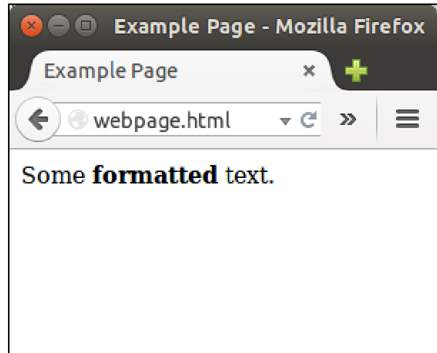
```
# Example of writing reformatted names of RSCS academics to a new file

with open("rscs_academics.txt", "r") as fin:
    with open("formatted_names.txt", "w") as fout:
        for name in fin:
            p = name.find(",")
            if (p == -1): continue
            lastname = name[0:p].strip().title()
            firstname = name[p+1:].strip()
            fout.write("{}. {}\n".format(firstname[0], lastname))
```

which uses the `with` construct for closing files instead of the explicit `fh.close()` used above.

3.6.2 Markup Languages and Data Exchange Formats

A markup language is a system for annotating a text document in order to encode typesetting instructions or semantic tags. A classic example is the HyperText Markup Language (HTML) used to encode documents on the Web. A simple HTML document is shown below.

```
<!doctype html>

<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Example Page</title>
</head>

<body>
Some <b>formatted</b> text.
</body>
</html>
```



It's important to be able to recognise and programmatically manipulate documents formatted in a markup language for two reasons. First, you may want to extract or modify data from one or many documents, such as webpages. Understanding the format allows you to quickly parse the document and extract or modify the information you need. The following example shows you how to read an HTML document and modify the `<title>`. More sophisticated parsing and modification is possible using the `HTMLParser` from the `html.parser` module.

Python Code

```
# rewrite title line in an html document
fout = open("index_new.html", "w")
with open("index.html", "r") as fin:
    for line in fin:
        if line.find("<title>"):
            line = "<title>All Your Base Do Belong to Us</title>"
        fout.write(line)

fout.close()
```

Note again the use of the `with` statement and the line-by-line processing of the file. The above code also only works if the `<title>` and `</title>` tags appear on a line by themselves. Again the `HTMLParser` class can be used to make the code more robust. We omit details here but will give an example in a Lecture 7.

Second, you may wish to produce a document that shows experimental output or analysed results in a convenient format. Consider, for example, an experiment that produces lots of images and writes them to a directory. Now let's assume that you wish to visually compare the images from two different experimental runs (two different directories). The following code snippet produces an HTML document to compare the images from two different directories side-by-side. The document can be viewed in a web-browser.

Python Code

```
import glob
import os

# open HTML document and print header information
fh = open("my_report.html", "w")
fh.write("<html>\n<head><title>My Results</title></head>\n")
fh.write("<body>\n")

# Read all .png files from one directory and assume that
# a file with the same name appears in the second directory.
# Display the images in a table.
fh.write("<table>\n")
fh.write("<tr><th>name</th><th>dir_1</th><th>dir_2</th></tr>\n")
for imgfile in glob.glob("dir_1/*.png"):
    basename = os.path.basename(imgfile)
    fh.write("<tr>")
    fh.write("<td>" + basename + "</td>")
    fh.write("<td><img src='dir_1/" + basename + "'></td>")
    fh.write("<td><img src='dir_2/" + basename + "'></td>")
    fh.write("</tr>\n")

fh.write("</table>\n")

# print footer information and close HTML document
fh.write("</body>\n")
fh.write("</html>\n")
fh.close()
```

XML

XML (Extensible Markup Language) is another very widely used markup language for storing and communicating data. Unlike HTML, in XML users can define their own tags. The following is an example of an XML document for storing student grades for a course.

```
<student>
  <name>Homer Simpson</name>
  <asgn1>10.0</asgn1>
  <asgn2>50.0</asgn2>
  <asgn3>30.0</asgn3>
  <exam>45.0</exam>
</student>
<student>
  <name>Marge Simpson</name>
  <asgn1>70.0</asgn1>
  <asgn2>65.0</asgn2>
  <asgn3>80.0</asgn3>
  <exam>75.0</exam>
</student>
```

XML can be cumbersome to work with, especially for something simple like storing student grades. Fortunately, many languages, including Python, come with libraries for manipulating XML files. In Python, these libraries are grouped in the `xml` package. You are welcome to explore this package and use it in your projects but we will not cover XML further in this course.

JSON

Another useful data format—not strictly a markup language—for exchanging data is JSON, which stands for JavaScript Object Notation. JSON was developed as a way to store and communicate data objects as attribute-value pairs in the JavaScript programming language. Since then libraries that read and write JSON files have been developed for a number of programming languages including Python and the format has been embraced by many programmers. In Python the `json` package provides an API for encoding and decoding objects as JSON strings. The following is an example JSON file for storing student grades akin to the XML example above.

```
[
  {
    "name": "Homer Simpson",
    "asgn1": "10.0",
    "asgn2": "50.0",
    "asgn3": "30.0",
    "exam": "45.0"
  },
  {
    "name": "Marge Simpson",
    "asgn1": "70.0",
    "asgn2": "65.0",
    "asgn3": "80.0",
    "exam": "75.0"
  }
]
```

3.6.3 Space Delimited and CSV Files

Comma-separated value (CSV) and space delimited files are very common for storing tabular data in a text file format. Many popular software packages, such as Microsoft Excel, are able to read and write CSV files. A example of a CSV file is shown below where we are storing the student grades for a course.

The "Save As..." dialogue in Excel allows you to save spreadsheets in *Comma Separated Value* format (with a .csv file extension) or *Tab Separated Value* format (with a .txt file extension).

```
Name,Assignment 1,Assignment 2,Assignment 3,Exam
Homer Simpson,10.0,50.0,30.0,45.0
Marge Simpson,70.0,65.0,80.0,75.0
Bart Simpson,15.0,90.0,40.0,55.0
Lisa Simpson,100.0,95.0,100.0,99.0
Maggie Simpson,0.0,10.0,5.0,20.0
Carl Carlson,60.0,70.0,70.0,75.0
Ned Flanders,65.0,80.0,75.0,70.0
Barney Gumble,15.0,45.0,40.0,50.0
Lenny Leonard,15.0,45.0,40.0,60.0
Otto Mann,90.0,85.0,80.0,85.0
Seymour Skinner,95.0,90.0,100.0,40.0
```

Now let's say we wish to calculate each student's final grade for the course. We would need to read the file, combine the marks for each assessment, and output the final grades. Fortunately, Python provides a module called `csv` for reading and writing CSV files that will simplify our task:

Python Code

```
1  import csv
2
3  # initialise list of final grades
4  final_grades = []
5
6  # read student file and compute final grades
7  fh = open("student_grades.csv", "r")
8  reader = csv.reader(fh)
9  next(reader) # skip the first row
10 for row in reader:
11     student_name = row[0]
12     student_grade = 0.25 * sum([float(i) for i in row[1:]])
13     final_grades.append([student_name, student_grade])
14
15 fh.close()
16
17 # write grades to a new file
18 fh = open("final_grades.csv", "w", newline="")
19 writer = csv.writer(fh)
20 writer.writerow(["Name", "Final Grade"])
21 for row in final_grades:
22     writer.writerow(row)
23
24 fh.close()
```

Note that we could just have stored the student grades in Python code files directly as the following example shows. Why might this be a bad idea?

Python Code

```
student_grades = [
  ["Homer Simpson", 10.0, 50.0, 30.0, 45.0],
  ["Marge Simpson", 70.0, 65.0, 80.0, 75.0],
  ["Bart Simpson", 15.0, 90.0, 40.0, 55.0]
]
```

JSON

As a further example we show code for computing the final grades for the case where the student data is stored in a JSON format (for example, if extracted from a database) as discussed in the section above. However, in the case of student grades a CSV file is arguably more convenient than a JSON file. The code is:

Python Code

```
import json

# initialise list of final grades
final_grades = []

# parse JSON file
with open('student_grades.json') as fh:
    students = json.load(fh)
    for student in students:
        student_name = student["name"]
        student_grade = 0.25 * (float(student["asgn1"]) +
            float(student["asgn2"]) +
            float(student["asgn3"]) +
            float(student["exam"]))
        final_grades.append([student_name, student_grade])

# print grades
for name, grade in final_grades:
    print("{}\t{}".format(name, grade))
```

3.6.4 Binary Files

Binary data, such as images and music, needs to be encoded in some way when stored in a computer. This tells the computer how to interpret the data. For example, an image should be interpreted as a table of RGB values, one of each pixel, and a music data should be interpreted as an audio waveform. Without an agreed upon encoding scheme there is no way for the computer to know how to display or play the data.

A **file format** defines how data in a file is encoded. Binary data is also encoded when it is sent across a communication channel, such as the Internet, and storing a data on a computer's disk drive can be modeled in the same way. Here the disk drive acts as the channel and the "transmission" is across time rather than space, i.e., at some later time the data will be read back from the disk and decoded. Figure 3.12 shows a generic model of this process.

Most binary file formats compress the data to reduce storage space on disk or transmission time when sending across a network. They also incorporate error checking and recovery mechanisms that allow the decoder to recover from (a

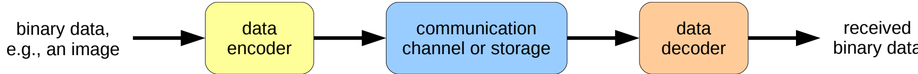


Figure 3.12: Illustration of a communications channel.

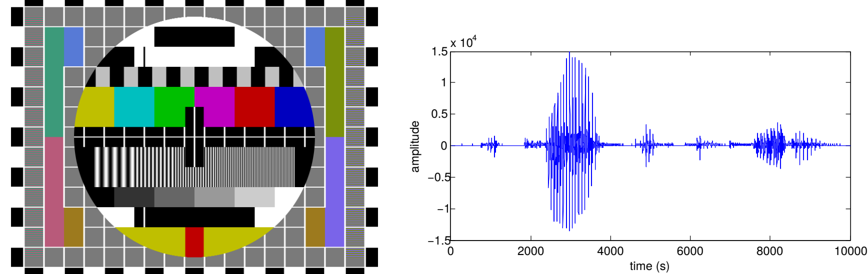


Figure 3.13: An image an audio waveform are just numbers stored in memory. The way these numbers are interpreted allows us to view images and play audio files.

small amount) of random corruption of the data that may occur during storage or transmission.

There are a multitude of schemes for representing different types of binary data and we'll only discuss a few formats here. Different formats are identified by the operating system through file extensions (e.g., "`.png`") and magic numbers (the first few bytes) within the file. However, both these mechanisms can be fooled. Most formats include *checksums* to detect errors in decoding, either caused by data corruption or incorrect determination of the file format.

Images and Video File Formats

Images and videos contain are large and contain a lot of redundant information. Consider, for example, the large uniform coloured patches in the image in Figure 3.13. One of the main things that an image file format tries to do is compress the amount of data needed to represent the image by encoding this redundancy in an efficient way.

Image an video file formats can be divided into *lossy* and *lossless*. Lossless formats compress the data in a way that allows exact recover of the original image when decoded. For example, long sequences of the same colour can be encoded efficiently using run-length coding where instead of storing the full sequence we store just two numbers, a value and a repetition count. The PNG image format uses this type of encoding (on top of a preprocessing step to further remove redundancy).

Lossy formats compress the data in a way that does not allow for exact recovery of the original image—information is lost. Usually the difference between the decoded image and original image is imperceptible to the human eye. Such formats often result in smaller files than lossless schemes. The JPEG format is a typical example.

There are various Python libraries for reading images. One such library is `matplotlib`, which we will encounter again later in the course for visualising data in the form of plots and charts. The following code shows an example of loading and displaying an image.

Python Code

```
import matplotlib.pyplot as plt
img = plt.imread("example_image.png")
print("image is {}-by-{}".format(len(img[0]), len(img)))
plt.imshow(img)
plt.show()
```

Once loaded the image is represented as a multi-dimensional array of RGB values. The red, green and blue values for a pixel at the r-th row and c-th column in the image can be extracted as

Python Code

```
red = img[r][c][0]
green = img[r][c][1]
blue = img[r][c][2]
```

Given this, the following shows a fun example of swapping the red and blue colour channels.

Python Code

```
import matplotlib.pyplot as plt
img = plt.imread("example_image.png")

# swap red and blue colour channels
for r in range(len(img)):
    for c in range(len(img[r])):
        img[r][c][0], img[r][c][2] = img[r][c][2], img[r][c][0]

plt.imshow(img)
plt.show()
```

Note that `matplotlib` provides many functions for manipulating image channels so you can swap two channels using less code than shown in the code snippet above (e.g., try `img[:, :, 0], img[:, :, 2] = img[:, :, 2], img[:, :, 0]`. Nevertheless the example does demonstrate some useful concepts in iterating over pixels.

Audio Formats

Like images audio signals contain a large amount of redundancy and file formats have been developed to exploit this redundancy to reduce the amount of data stored. MP3 and WAV are two very common formats.

The `wave` module in Python provides functionality for reading and writing WAV audio files. For other audio formats you will need to find and install additional libraries (e.g., `audiolab`).

3.6.5 Next Lecture

- String processing

3.7 Lecture 7: String Processing

Learning Outcomes

- Understand basic string manipulation such as slicing, concatentation, splitting, joining, and capitalisation.
- Write basic parsers for extracting values from, e.g., comma or whitespace separated lists.
- Be aware of different encodings for characters (ASCII, UTF-8, etc.) and know that, by default, all Python 3 strings are Unicode.

Overview

This lecture covers how to inspect and process strings using a variety of inbuilt Python methods and briefly discusses how strings are encoded and decoded.

3.7.1 String Indexing and Slicing

As we saw in Section 3.4.2, the data structure most commonly used to represent words or text in Python is a *string*. A string is just a sequence of characters which can be accessed like elements in a list. Characters are accessed by specifying the *index* of the character inside square brackets after the string. Note that just like lists, the first character in a string has index 0, not 1.

Characters in strings in Python 3 can be any Unicode symbol, including letters from alphabets of various languages and other symbols, e.g., 'á B ç' is a valid string.

The following example shows how characters from a string containing the lowercase English alphabet can be accessed. Note that negative numbers can be used as indices. In this case, the index represents the position of the character from the *end* of the string: -1 is the last character, -2 the second last, and so on. Using an index that is too large for the string results in an error.

Python Console

```
>>> alphabet = 'abcdefghijklmnopqrstuvwxyz'
>>> alphabet[0]
'a'
>>> alphabet[9]
'j'
>>> alphabet[-1]
'z'
>>> alphabet[-3]
'x'
>>> alphabet[30]
IndexError: string index out of range
>>> alphabet[-27]
IndexError: string index out of range
```

Just as with lists, strings can be sliced using the *string*[*start*:*end*] notation. In the following example, the slice notation is used to pull out the first four characters, the last four characters, and the middle three characters of the phrase 'Cats & Dogs'.

Python Console
```
>>> phrase = 'Cats & Dogs'
>>> phrase[:4]  # From start up to (and including) character 4
'Cats'
>>> phrase[4:]  # From just after character 4 to the end
' & Dogs'
>>> phrase[4:7] # From just after character 4 to character 7
' & '
```

You can even combine slicing with negative indexing, as in the next example.

Python Console
```
>>> phrase = 'Cats & Dogs'
>>> phrase[-4:]   # From 4th from end of string to end of string
'Dogs'
>>> phrase[:-4]   # From start up to just before 4th from end
'Cats & '
>>> phrase[-7:-4] # From 7th last to before 4th last
' & '
```

This mixing of slicing and negative indexing can be pretty confusing at first. The diagram shown in Figure 3.14 can help remember how indexing and slicing work with both positive and negative values. The key trick is that when you index a string the number count boxes from the start or end of the string, while in slicing the numbers count the gaps from the start or the end.

The easiest way to get used to these conventions is by playing around with some examples of your own in a Python console.

Slicing can also take an optional *step* parameter, e.g., `phrase[0:5:2]` takes every second character between gap 0 and gap 5. Try this with negative steps. What happens?

Indexing (boxes)
From start 0 1 2 3 4 5 6 7 8 9 10
C a t s & D o g s
From end -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

Slicing (gaps)
: 1 2 3 4 5 6 7 8 9 10 :
C a t s & D o g s
: -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 :

Figure 3.14: An index for a string (or list) counts the number of *boxes* from the start (positive index) or end (negative index) whereas in slicing the numbers count the number of *gaps* from the start or end with : denoting a boundary.

3.7.2 Converting Between Objects and Strings

[TODO: the string is not the object; difference between 42 and '42'; difference between (10, 5) and '(10, 5)']

3.7.3 String Methods

Testing String Properties

There are a number of standard functions on strings that allow you to test various properties. For example, whether a string is in uppercase or lowercase, or whether certain substrings are present.

A very basic property of a string (that also applies to lists) is its *length*, that is, the number of characters in a string. In Python, the length of a string can be obtained by using the `len` function.

Python Console

```
>>> alphabet = 'abcdefghijklmnopqrstuvwxyz'
>>> len(alphabet)
26
>>> phrase = 'Cats & Dogs'
>>> len(phrase)
11
>>> len(phrase[:4])
4
>>> len(phrase[-4:])
4
```

Characters in written English and other languages can be either *uppercase* (e.g., 'A', 'T') or *lowercase* (e.g., 'a', 't'). Characters can also be classified as *alphabetic* (e.g., 'a', 'b') or *numeric* (e.g., '6', '0') or *whitespace* (e.g., spaces, tab and newline characters). Unsurprisingly, there are a number of functions in Python that let you test whether the characters within a string have these properties. These include `islower` and `isupper` for testing upper and lower case, `islapha` and `isdigit` for testing whether characters are letters or numbers, and `isspace` for testing whether characters are whitespace. As the examples below show, these functions test whether *all* characters in a string are of a certain type.

Python Console

```
>>> 'a'.islower()
True
>>> 'A'.islower()
False
>>> '73'.isdigit()
True
>>> '3F'.isdigit()
False
>>> '  \tHi\n'.isspace()
False
>>> '  \t  \n'.isspace()
True
```

There are many more string methods for testing properties that have not been mentioned here. The table below gives a brief summary of those above and some others you may find useful.

Method	Description
s.`count(`c`)`	Count how many times string c appears in s.
s.`islower()`	True if all characters in s are lower case.
s.`isupper()`	True if all characters in s are upper case.
s.`isalpha()`	True if all characters in s are alphabetic (*e.g.*, 'a', 'Q', 'ñ').
s.`isdigit()`	True if all characters in s are digits (*e.g.*, '1', '2', etc.).
s.`isspace()`	True if all characters in s are whitespace (*i.e.*, space, tab, newline).
s.`find(`w`)`	Return the first index of string w in s or -1 if w is not in s.

Table 3.6: String methods for property testing.

We will now briefly look at how property testing, indexing and slicing can be used to process data. As we saw in Lecture 6, it is common for data to be stored in text files using what is known as "Comma-Separated Values" format (or CSV for short). The Python `csv` module should always be your first choice for working with these sorts of files, but it is instructive to consider how we might implement a very simple version of `csv.reader` using string operations.

Example 3.7.1. Let's suppose we had to separate the following single line from a CSV file into a list.

```
Homer Simpson,10.0,50.0,30.0,45.0
```

One approach to doing this is to use `find` to get the index of the first comma in the string, take the characters up to that index, and repeat the process on the characters after the comma.

Python Code

```
row = 'Homer Simpson,10.0,50.0,30.0,45.0' # Row to split
columns = [] # The list to put the columns into

# Loop through the row by repeatedly finding the next comma index
next_comma_index = row.find(',')
while next_comma_index >= 0:
    # Pull out the next column and append it to the list
    column = row[:next_comma_index]
    columns.append(column)

    # Focus on the remainder of the row (the +1 skips the comma)
    row = row[next_comma_index+1:]
    next_comma_index = row.find(',')

# Append the remain part of the row as the final column
columns.append(row)

print(columns)
```

Try to mentally execute the above code as it processes the string in `row` by keeping track the value in `row`, `columns`, and `next_comma_index`.

Processing Strings

As well as methods for testing string properties, there are a number of string methods that let you process a string by returning a new string with certain characters changed or removed. The process that was implemented in Example 3.7.1 using `find` and slicing is also available as a method called `split`:

Python Console

```
>>> row = 'Homer Simpson,10.0,50.0,30.0,45.0'
>>> row.split(',')
['Homer Simpson', '10.0', '50.0', '30.0', '45.0']
```

Table 3.7 describes some of the other string processing methods. For your own benefit, it is worthwhile browsing the documentation for other common

string functions available in Python 3: `https://docs.python.org/3.1/library/stdtypes.html#string-methods`. Try reading through this with a Python console open and playing with the various functions so as to get familiar with them.

Method	**Description**
s.`lower()`	Convert all characters in *s* to lower case.
s.`upper()`	Convert all characters in *s* to UPPER CASE.
s.`title()`	Convert all characters in *s* to Title Case.
s.`strip()`	Remove all surrounding whitespace from *s*.
s.`split(`*c*`)`	Split *s* into a list using the string *c* as a delimiter (or space if *c* is not given).

Table 3.7: String methods for property testing.

3.7.4 Characters and Unicode

We have been saying that strings are sequences of characters, but what are characters exactly? Thanks to the complexity of human languages and the way they are written there is not a simple answer to this. Fortunately, a lot of work has gone into encapsulating this complexity in Python 3 and most of the time you will not have to worry about how a character is represented. However, if you encounter strange symbols or bugs when processing text files it is useful to know a couple of things about character representations so you can search for and understand techniques for fixing these sorts of problems.

Characters in Python 3 are represented as *Unicode*. This is a standard that assigns every glyph that is used in any written human language a unique *codepoint*. Each codepoint is a number that is usually written as 'U+NNNN' where 'NNNN' is four hexadecimal digits. For example, the character 'A' in the English (and other) alphabet has the code point 'U+0041'. The character '⽇' from written Japanese or Chinese is assigned the codepoint 'U+2F47'. It is important to stress that these codepoints are *not* bytes or any other computer representation — they are just numbers. How these numbers are represented on your computer depends on the *encoding* used to represent the Unicode codepoints.

The built-in Python functions `ord` and `chr` convert to and from Unicode codepoints.

A Brief History of Character Encoding

Before diving into the way Unicode characters are encoded, it is useful to understand a small part of the history that led to the development of Unicode.

Due to memory constraints, early attempts at encoding characters tried to use as few bits as possible. Since there are 26 upper case, 26 lower case, and a handful of punctuation and other special or control characters, at least 7 bits (i.e., 128 different binary strings) per character were needed for encoding. Two separate standards for encoding characters at appeared in the 1960 to specify how to encode characters as bit strings: EBCDIC and ASCII. Several different encodings appeared for other languages, for example, KOI8 for the Russian (Cyrillic) alphabet and Latin1 for French, appeared later.

ASCII became the standard and was widely used for English text until the mid 2000s when Unicode became the dominant encoding. One big reason for the switch to Unicode was interoperability. Using ASCII was fine for English

but what if you needed to write a Russian phrasebook for English speakers? As the world wide web became increasingly popular and more and more text was represented digitally there was a need for a single character representation worked for all languages.

Work on what is now the Unicode standard began in 1988. Originally, it was designed to handle up to 16,384 characters — enough to encode all modern languages. However, in 1996 this was revised so that over a million characters could be specified. This enabled the inclusion of older characters such as those from Egyptian hieroglyphics and rare Chinese characters and leaves room to include other old languages and changes to existing ones that may be needed in the future.

The Unicode is not a static standard. New versions appear regularly with new characters. As recently as June 2015, the standard was updated to version 8.0, and then again to 9.0 in June 2016.

Encodings

Unicode is only an assignment of characters to codepoint. It is *not*, by itself, an encoding into bytes that can be stored in a computer's memory or disk. Indeed, there are several way codepoints can be translated to and from bytes. These are called Unicode *encodings*.

The most common encoding of Unicode is called *UTF-8* which is short for "Universal Coded Character Set + Transformation Format—8-bit". It is reportedly used for over 84% of all web pages. It is an encoding that neatly trades off space and flexibility by using 8 bits per character for a base set of 256 English and other characters but is able to specify that additional bytes are needed for other characters, such as Chinese. The *UTF-16* encoding is similar in that it can specify when extra bytes are needed but uses 2 bytes for a larger base set of over 16,000 characters. In contrast, the *UTF-32* encoding uses 4 bytes for every character which can represent over 4 billion different numbers and so does not need to be able to declare when extra bytes are needed.

For example, see the W3Techs surveys at `http://w3techs.com/`.

The following example show how the character 'A' is represented in UTF-8, UTF-16 and UTF-32. Note that the 'b' prefix in front of the output indicates that what follows it are bytes (e.g., `\\xc3` represents the bits `11000011`). The Python `encode` and `decode` methods are used to convert Unicode to and from bytes.

Python Console

```
1 >>> s = 'A'
2 >>> s.encode('utf-8')
3 b'A'
4 >>> s.encode('utf-16')
5 b'\xff\xfeA\x00'
6 >>> s.encode('utf-32')
7 b'\xff\xfe\x00\x00A\x00\x00\x00'
8 >>> b'\x40'.decode('utf-8')
9 '@'
```

One downside of UTF-32 is that, for English text files, the file size can be four times larger than necessary. Another downside is that UTF-16 and UTF-32 encodings are complicated by the fact that the order in which bytes are stored need to be taken into account when encoding or decoding each character. UTF-8

does not have this problem.

Once again, when you do basic string processing in Python 3 you will likely not have to worry about any of these technical details and things will "just work"... except in the rare instance when they don't. These notes are meant to help you get started in solving problems should you encounter weird characters or error messages involving string encodings.

Further Reading

A longer discussion of Unicode in Python can be found here: `https://docs.python.org/3/howto/unicode.html`.

If you plan on working with strings on a regular basis, you may want to read more on the details of Unicode and its encodings here: `http://www.joelonsoftware.com/articles/Unicode.html`.

3.7.5 Next Lecture

- Functions
- Advanced concepts
  - Recursion
  - Callbacks

3.8 Lecture 8: Functions

Learning Outcomes

- Understand that a functions encapsulate a task within a program and are used to help organise code and avoid repeated code.
- Understand the difference between a built-in function and a user-defined function.
- Be able to write small functions in Python.
- Be able to use member functions and recognise the difference between different ways to invoke a function (specifically "." (dot) notation for member functions).

Overview

In this lecture we introduce the concept of a function in a programming language. We discuss how functions are declared and invoked in Python, and that variables defined within a function have local scope. We also distinguish between general functions and member functions.

The concept of a **function** in a programming language is similar to the concept of a function in mathematics. In mathematics a function defines a mapping from some input space to some output space, i.e., it takes input arguments and returns output values. Similarly, in programming languages a function performs a computation on its inputs and (optionally) produces some output. We have already seen numerous examples of functions, e.g., the function `len` takes as input a list and returns as output the length of the list.

The are many reasons why functions are useful. The main reasons are to:

- abstract away low-level details and allow the programmer to focus on solving the problem at hand;
- improve code organisation, testing and maintenence; and
- avoid code repeation and facilitate better sharing and reuse.

3.8.1 Inputs and Outputs

At its simplest a function is a block of code that takes in inputs (also called *parameters*) and computes outputs (also called a *return value*). When we provide a specific input to a function parameter we call the input an *argument*. We will see later that in special situations a function can also modify its arguments. It is also possible for a function to not have a return value but to have other side effects (e.g., the `print` function). The general pattern for calling a function is

```
return_value = function_name(arguments)
```

A concrete example is the built-in function `sorted`, which takes as input a sequence and returns a sequence with the elements in sorted order as output.

Python Console

```
>>> a = [42, 10, 40, 5]
>>> b = sorted(a)
>>> b
[5, 10, 40, 42]
>>> a
[42, 10, 40, 5]
```

Objects (which we will discuss more in a later lecture), like lists, can have functions associated with them, and which operate on the object itself. Such functions are called *methods* (or sometimes member functions). Instead of providing the object as an input argument a method is invoked using the following pattern,

```
object_name.method_name(arguments)
```

Methods can also have return values, but their main purpose is to modify the object calling the method. (Methods also have access to private data held by the object and are aware of implementation details so can be made more efficient than regular functions in some cases).

It turns out that Python lists have their own `sort` method as the following code demonstrates.

Python Console

```
>>> a = [42, 10, 40, 5]
>>> a.sort()
>>> a
[5, 10, 40, 42]
```

Note that unlike the `sorted` function, the `sort` method reorders the elements of the original list.

Some functions also take optional parameters. If an argument is not given for the parameter then a default value is used. Parameters can also be explicitly named. Back to our built-in `sorted` function we show an example of both optional and named parameters.

Python Console

```
>>> a = [42, 10, 40, 5]
>>> b = sorted(a, reverse=True)
>>> b
[42, 40, 10, 5]
```

The `print` function is another example of a function with optional parameters that you may use often. The following code snippet shows a simple example of printing out a matrix.

Python Code

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for row in matrix:
    for entry in row:
        print(entry, end="\t")
    print()
```

3.8.2 Mental Execution Model

A good mental model of function execution is captured by the following steps:

- the expressions for each argument are evaluated and assigned to the corresponding function parameters;
- code within the function body is executed;
- the expression for the return value is evaluated and assigned to the output variable, if present;
- the program continues running at the statement immediately following the function call.

Once a function has been defined it can be called as many times as you like from anywhere in the code, each time following the same process as outlined above. This is one of the most powerful concepts in programming as it allows you to better organise your code at the same time as facilitating reuse.

Note that some functions may have state or result in side effects, meaning that they (may) produce different output even if called with the same input. In general this is to be avoided, but it can sometimes be useful (e.g., when generating psuedo-random numbers).

3.8.3 Defining a Function

The Python language itself defines many built-in functions. Others are provided from libraries and modules. These may suffice for simple scripts, but for any significant piece of code you will want to define functions of your own.

Remember our example of computing final course grades from data stored in a CSV file. Let's now convert those numerical grades to a letter grade. First, we will write a function to do the conversion. Note the indentation of the function body.

Python Code

```
def convert_to_letter_grade(grade):
    """Converts from a numerical grade to a letter grade."""

    if (grade >= 85.0): return "HD"
    elif (grade >= 75.0): return "D"
    elif (grade >= 65.0): return "CR"
    elif (grade >= 50.0): return "P"
    else: return "F"
```

Putting it all together.

Python Code

```
import csv

# open input and output files
fin = open("student_grades.csv", "r")
fout = open("final_grades.csv", "w", newline="")
reader = csv.reader(fin)
writer = cvs.writer(fout)

# deal with first row
next(reader)
writer.writerow(["Name", "Final Grade", "Final Letter"])

# for each student compute the final grade
for row in reader:
    student_name = row[0]
    student_grade = 0.25 * sum([float(i) for i in row[1:]])
    student_letter = convert_to_letter_grade(student_grade)
    writer.writerow([student_name, student_grade, student_letter])

# close input and output files
fin.close()
fout.close()
```

A more general-purpose version of the above could would replace Line 16 with the following to allow it to work on an arbitrary number of grades (not just four):

Python Code

```
student_grade = sum([float(i) for i in row[1:]]) / (len(row) - 1)
```

Recall back in Lecture 2 we said that a variable needs to be initialized before it can be used. The parameter list for a function declares the input variables for that function. They are initialised (with the input expression) each time the function is called.

Return Values and Mutable Parameters

The `return` keyword within a function provides an expression for evaluating the return value and instructs Python to exit the function and go back to where it was called from. A function can have multiple `return` statements, as we have seen, and each can have a different expression. The first `return` statement executed determines the value returned.[3]

When a mutable object (like a list) is passed as an argument to a function the contents of the object may be modified by the function. Typically this should be avoided as it introduces unexpected side effects, which can be difficult to debug. Consider the following example.

Integers and strings are immutable in Python.

[3] If a function ends without a `return` statement, e.g., by closing the indentation block, the function is assumed to return the special value `None`.

Python Code

```
def f(x):
    """Return the sum of elements in x."""
    sx = 0
    while (len(x) > 0):
        sx += x.pop()
    return sx

def g(x):
    """Return the sum of elements in x."""
    sx = 0
    for i in x:
        sx += i
    return sx

grades = [10.0, 50.0, 30.0, 45.0]

print(g(grades))  # prints 135.0
print(g(grades))  # prints 135.0
print(grades)

print(f(grades))  # prints 135.0
print(f(grades))  # prints 0.0
print(grades)
```

Optional Parameters

In addition to the mandatory parameters of a function (listed as a comma-delimited list of local variable names) can be optional parameters. These are specified in the parameter list as a variable name followed by an expression. If an argument is not provided for an optional parameter, the expression is used to provide a default value.

The following code extends our student grade example from above.

Python Code

```
def convert_to_letter_grade(grade, pass_fail=False):
    """Converts from a numerical grade to a letter grade.

    The optional parameter pass_fail can be set to True to
    return just a pass or fail letter grade. Otherwise the
    grades HD, D, CR, P and F are used."""

    if pass_fail:
       if (grade >= 50.0): return "Pass"
       return "Fail"

    if (grade >= 85.0): return "HD"
    elif (grade >= 75.0): return "D"
    elif (grade >= 65.0): return "CR"
    elif (grade >= 50.0): return "P"
    else: return "F"
```

Now the `convert_to_letter_grade` function can be invoked to produce either the standard letter grades (the default) or a pass/fail grade.

Python Code

```
convert_to_letter_grade(student_grade, True)  # Pass or Fail
convert_to_letter_grade(student_grade, False) # HD, D, CR, P, or F
convert_to_letter_grade(student_grade)        # HD, D, CR, P, or F
```

Named Arguments

Usually arguments will be passed to a function in the order in which the corresponding parameters are defined (known as *positional arguments*). However, there are situations where we may want to specify arguments out-of-order. Typically this occurs when there are multiple optional parameters and we only want to override a few of them. *Named arguments* provide a mechanism for such situations. The idea is best illustrated through an example.

Python Code

```
def bracket(text, pre="(", post=")"):
    return pre + text + post
```

Python Console

```
>>> print(bracket("hello"))
(hello)
>>> print(bracket("hello", post=")!"))
(hello)!
>>> print(bracket("hello", post=" :)", pre=""))
hello :)
>>> print(bracket("hello", "", "!"))
hello!
```

3.8.4 Variable Scope

The parameters and any variables defined within a function body fall within the function's namespace. That is, they have local scope and cannot be accessed outside of the function.

Python Code

```
def double(x):
    y = 2 * x
    return y

z = double(5)
print(z)
print(y)
print(x)
```

Whenever a local variable has the same name as a global variable, the local variable hides the global one. The `global` keyword can be used to override this behaviour. However, using global variables is considered bad programming practice and should be avoided if possible.

Python Code

```python
def scale(x):
    global a, y
    y = a * x
    return y

a = 2
z = scale(5)
print(z)
print(y)
print(x)
```

3.8.5 Documenting Functions

Use **docstrings** to explain how to *use* code; use **comments** to explain why and how code works.

In many programming languages a function's behaviour is documented by writing comments in the source code. Python provides a very elegant and helpful mechanism for documenting functions through a so-called *docstring* (or documentation string), which both acts as a comment and provides runtime access for introspection.A docstring is defined by including a string literal as the first statement in the function body. Since the documentation string is usually long and spans multiple lines a triple-quoted string is often used as we have already seen. The docstring can be accessed via the function's `__doc__` attribute. For example,

Python Code

```python
def triple(x):
    """Returns three times its argument."""
    return 3 * x

print(triple.__doc__)
```

Within the Python console we can also use the `help` function:

Python Console

```
>>> help(triple)
triple(x)
   Returns three times its argument.
```

As a minimum a docstring should provide a concise description of the functions purpose. Longer docstrings may include more details about the function including a description of the parameters, preconditions, return value, any side effects, and usage examples and references. It is convention for paragraphs to be separated by an empty line.

Always keep documentation and comments up-to-date. Incorrect or misleading comments are worse than no comments at all. So whenever you change the code make sure you change the documentation and comments too.

3.8.6 Advanced Concepts

Recursion

It should be fairly obvious that functions can call other functions. What is not so obvious is that functions can call themselves. This is known as *recursion*.

Generating the n-th Fibonacci number is perhaps the most classic (and poorly given) example.[4]

Python Code

```
def fib(n):
    """Return the n-th fibonacci number by recursion."""

    if (n == 0) or (n == 1):
        return 1

    return fib(n - 1) + fib(n - 2)

def fib2(n):
    """Return the n-th fibonacci number without recursion."""
    a, b = 1, 1

    for i in range(n):
        a, b = b, a + b

    return a

print(fib(10))
print(fib2(10))
```

Recursive functions always need one or more base cases that stop the recursion. Every recursive algorithm can be replaced by an (often more efficient) iterative one. However, an algorithm expressed by recursion can sometimes be simpler to implement. Examples include computing the greatest common divisor of two numbers or solving the famous Towers of Hanoi problem (but not computing Fibonacci numbers).

Callbacks

Callback functions used in event-driven programming is another advanced concept that will come up again when we discuss visualisation and animation. Briefly, a callback function is a function that gets invoked whenever some event occurs, such as a mouse-click or a timer expiring. The event is said to trigger the callback. A registration process tells the program which callback to invoke.

Callbacks are also called event handlers and are very common in GUIs and network programming.

Callbacks are sometimes be implemented as member functions within classes (more on classes in a later lecture). A good example is the `HTMLParser` class in the `html.parser` module. Here is some example code that pulls out the current temperature for capital cities from the Bureau of Meteorology website.

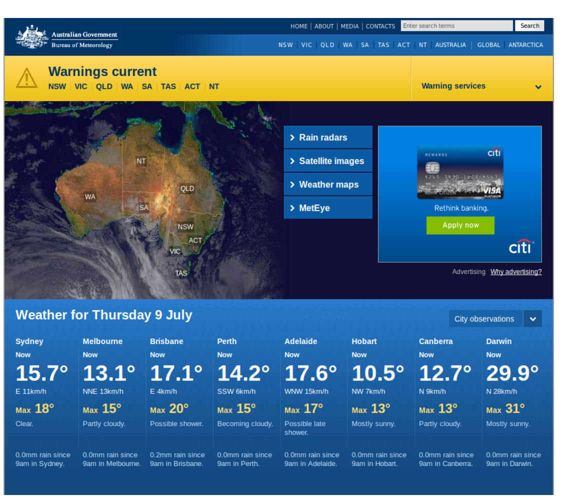
[4] A Fibonacci sequence is generated by by starting from 1 and 1, and then generating each successive number by summing the previous two in the sequence. The first few Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Python Code

```python
import urllib.request
from html.parser import HTMLParser

# query the BOM main page using urllib
URL = "http://www.bom.gov.au"

response = urllib.request.urlopen(URL)
html = str(response.read())
response.close()

# class to parse the website derived from HTMLParser
class BOMHTMLParser(HTMLParser):
    """Parser to extract capital city temperature from the
    main BOM webpage."""

    # member variables
    insideH3Tag = False
    insideNowTemp = False
    lastCityName = ""
    lastTemperature = ""

    def handle_starttag(self, tag, attrs):
        if (tag == "h3"):
            self.insideH3Tag = True
        if (tag == "p") and (len(attrs) > 0) and (attrs[0][1] == "now"):
            self.insideNowTemp = True

    def handle_data(self, data):
        if self.insideH3Tag:
            self.lastCityName = data
        if self.insideNowTemp:
            self.lastTemperature = data

    def handle_endtag(self, tag):
        if (tag == "h3"):
            self.insideH3Tag = False
        if (tag == "p") and self.insideNowTemp:
            self.insideNowTemp = False
            print("{}\t{}".format(self.lastTemperature, self.lastCityName))


# parse HTML to extract temperatures
parser = BOMHTMLParser()
parser.feed(html)
```

3.8.7 Next Lecture

- Binary and Boolean Logic
- Computer Architecture
- Basic Complexity Analysis




```
15.7 Sydney
13.1 Melbourne
17.1 Brisbane
14.2 Perth
17.6 Adelaide
10.5 Hobart
12.6 Canberra
29.9 Darwin
```

Figure 3.15: Bureau of Meteorology website for 9 July 2015 and correspodning output of our `HTMLParser` example code.

3.9 Lecture 9: Computer Architecture

Learning Outcomes

- Understand basic computer architecture and the main components of a computing system (CPU, memory, disk, I/O, operating system, network)
- Understand the distinction between volatile versus non-volatile storage.
- Understand that everything in a computer is stored and processed in binary.
- Appreciate the use of logic and arithmetic in computing systems, and that computer hardware operates at the level of machine instructions.
- Understand basic rules of Boolean algebra and their relationship to logical expressions.
- Understand the stored program concept.
- Understand that algorithms can be characterized in terms of running time and memory requirements.
- Understand that two algorithms can perform the same function but run in a very different amount of time.
- That this is independent of the specifics of how the algorithm is implemented (or what programming language it is implemented in).
- That running time and memory requirements often depend on the size of the input.
- Analyze the asymptotic running time of a simple algorithm in big-O notation.

Overview

This lecture combines two very different topics. First, it provides an overview of basic computer architecture and discusses how computer hardware interprets and then executes program instructions written in software. Second, it presents a brief tour of complexity theory and the analysis of algorithms.

3.9.1 Binary

Everything in a computer—all data and all programs—is stored, communicated, and processed as numbers. Your favourite song, the picture of your cat, the latest movie, and this document, to a computer are all just numbers. So how exactly does the computer know to take a sequence of numbers and play your favourite song? More on this later. For now let's talk about number systems.

Inside a computer numbers are represented in *binary*. So instead of writing the decimal number 42 as

$$42_{10} = 4 \times 10^1 + 2 \times 10^0$$

we write it as

$$\begin{aligned}101010_2 &= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 32_{10} + 8_{10} + 2_{10}\end{aligned}$$

where we have used subscripts to represent the *base*. The reason numbers are represented in binary is because it is easier for the computer hardware to work with two different states "1" and "0" (or "on" and "off") rather than ten different states (for each of the digits from 0 to 9). Each of the zeros and ones is called a *bit* (for binary digit). Bits are grouped together into fixed lengths usually of size 8, 32, or 64. An 8-bit sequence is often called a *byte*, while 32- and 64-bit sequences are called *words*. Continuing our example, 42 would be represented in a computer as the byte

$$00101010$$

where we interpret the rightmost bit (also called the least significant bit) as the units, the next bit as the twos, etc.

We are most used to the decimal system and so the primary format for entering and displaying numbers in (almost) all programming languages is as decimal numbers. However, binary representations and whether specific bits are 1 or 0 has a place in some applications.

Entering and reading numbers represented in binary can be cumbersome. Two alternatives that are often used are hexadecimal (base 16) and octal (base 8). The number 42 in hexadecimal and octal is

Hexadecimal uses symbols "0" to "9" and "A" to "F" for numbers 0 to 9 and 10 to 15, respectively.

$$2A_{16} \text{ and } 052_8$$

Note that each hexadecimal and octal symbol encodes four and three bits, respectively,

$$\underbrace{0010}_{2}\underbrace{1010}_{A} \text{ and } \underbrace{00}_{0}\underbrace{101}_{5}\underbrace{010}_{2}$$

So two hexadecimal symbols and three octal symbols form a byte.

In Python you can enter hexadecimal, octal and binary numbers by prepending `0x`, `0o`, and `0b`, respectively.

Python Console

```
>>> 0b0010101
42
>>> 0x2a
42
>>> 0x2A
42
>>> 0o052
42
```

The size of data in a computer is measured in terms of the number of bytes. For example, a one *megabyte* file (abbreviated 1MB) is a file that contains one million bytes.

Because $2^{10} = 1024$ is close enough to one thousand a *kilobyte* is often used to refer to 1024 bytes, keeping with the binary representation. Likewise, a *megabyte* is often defined as 2^{20}, or $1,048,576$, bytes.

Text

Older computer systems used to encode text (letters of the alphabet, digits, and common symbols) as a sequence of bytes where each byte represented one symbol. In modern systems multiple encoding system exist to support different languages and character sets. See Section 3.7 for details.

Arithmetic

Consider the addition of integers 42_{10} and 9_{10}. In decimal we first add the units 9 and 2 and find that we have a result of 1 unit and a *carry* of 1, which we add to 4 to give the answer 52_{10}. The same process of addition with carry can be performed in binary:

$$\begin{array}{ll} 00101010 & + \\ 00001001 & \\ \hline 00110011 & \end{array}$$

where the $1 + 1$ in the 2^3 column (fourth from the left) results in a 0 and carry of 1. Subtraction is performed in a similar way.

Thus far our discussion has been focused on positive integers. With an 8-bit byte we can represent integers in the range $[0, 255]$. If we try to add numbers that fall out of this range the operation results in overflow (or underflow)—a consequence of finite precision of the machine.

We can also encode negative integers using a representation known as two's complement.[5] Using two's complement integer addition and subtraction can be performed without modification.

When it comes to real numbers (i.e., fractional numbers) there are two common options. The first is a fixed-point representation, where a decimal (or more correctly binary) point is assumed at some fixed position within the binary representation. For example, if we set the binary point at the second bit position then

$$\begin{aligned} 00101010_2 &= 001010.10_2 \\ &= 2^3 + 2^1 + 2^{-1} \\ &= 10.5 \end{aligned}$$

The second option is known as *floating-point.* Here a number is represented by a sign bit, exponent (say, 8 bits) and mantissa, i.e., the fractional component following the decimal point (say, 23 bits). To decode a floating-point number we separate the binary word into sign, exponent and mantissa to express the number

$$0.(\text{mantissa}) \times 2^{(\text{exponent})}$$

In Python, the data type `float` is used for floating-point numbers.

[5] Essentially to represent a negative integer we start with a positive integer negate each bit in the binary representation and add one. For example, we get $-1_{10} = 1111111_2$ in two's complement by taking $1 = 00000001_2$ negating the bits to get 11111110_2 and adding 1. Now, consider adding 00000001_2 to 11111111_2. The answer is 0, confirming that 11111111_2 represents the number -1.

3.9.2 Boolean Algebra

The bits (1s and 0s) in a computer can be considered as representing logical values True and False. And since all operations inside computer hardware involve the manipulation of bits its operations can be formalised via Boolean algebra, i.e., a set of rules for operating on truth values.

The basic operations in Boolean algebra are *and* (conjunction), *or* (disjunction), and *not* (negation) defined by the following rules:

P	Q	$P \wedge Q$	$P \vee Q$	$\neg P$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

In computer hardware these are often implemented via the universal[6] *nand* (not and) or *nor* (not or) operations:

P	Q	P nand Q	P nor Q
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

De Morgan's Laws

Augustus De Moragn, a 19th-centuary mathematician came up with a set of useful pair of rules for transforming Boolean expressions. Formally, these are written as

$$\neg(P \wedge Q) \iff (\neg P) \vee (\neg Q)$$
$$\neg(P \vee Q) \iff (\neg P) \wedge (\neg Q)$$

Informally we say "the *not* of an *and* is the *or* or the *nots*" and "the *not* of an *or* is the *and* of the *nots*."

Consider the following Python example where we are iterating through a list of first and last names. Assume we wish to perform some operation for everyone who is not "Bart Simpson". We could use an `if` statement like the following:

Python Code
```
if not ((first_name == "Bart") and (last_name == "Simpson")):
   do_somthing()
```

or using De Morgan's laws we could also use

Python Code
```
if (first_name != "Bart") or (last_name != "Simpson"):
   do_somthing()
```

which is logically equivalent.

[6] All logical operations can be defined in terms of these. For example, $\mathbf{not}(P) = \mathbf{nand}(P, 1)$.

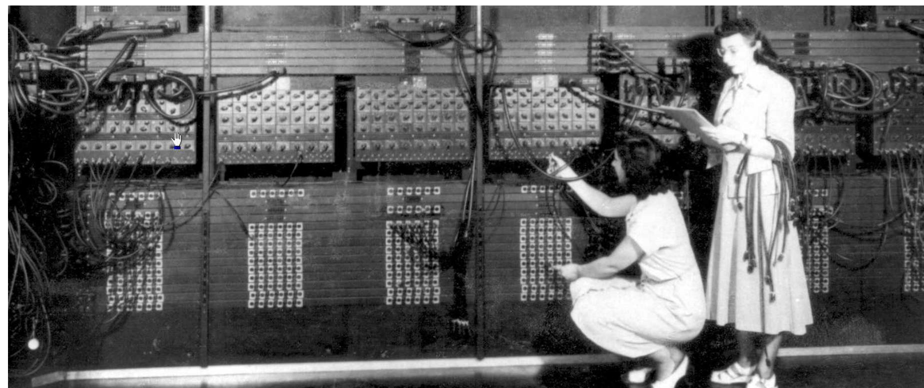


Figure 3.16: Technicians Gloria Ruth Gordon and Ester Gerston "programming" the ENIAC computer by plugging in wires.

3.9.3 Logic Circuits

Boolean and arithmetic operations inside a computer are performed using a combination of logic circuits and lookup tables. At the physical level the binary symbols 0 and 1 are encoded using different voltage levels within the electronic circuitry.

Logic gates take electrical inputs representing 1s and 0s and output an electrical signal representing 1 or 0 depending on the logical function implemented by the gate. Logic gates are wired together (inside integrated circuits) to create more complex operations, e.g., addition.

3.9.4 The Stored Program Concept

Early computers were hardwired to perform some fixed function. For example, ENIAC, one of the first electronic computers, was programmed by wiring up circuits. See Figure 3.16.

It wasn't until the 1940s that engineers started to think about the *stored program concept*—program instructions and data are stored in (random access) memory and the computer performs its function by executing the instructions.

The stored program concept allowed programs to be easily changed and promoted the development of general purpose hardware (CPUs), which can be customised for any application. This in turn gave rise to modern computer architectures (such as the von Neumann architecture) and the separation between hardware and software (further divided into operating system and application software). With this clear separation engineers were able to update hardware and software somewhat independently resulting in the exponential growth in computing power and capabilities that we have seen over the last 50 years.[7]

[7] The shrinking of transistors, the building blocks of logic circuits, has been a significant driver of faster and more powerful hardware, but by itself does not account of advances that have also been made on the software side independent of the hardware.

3.9.5 Basic Complexity Analysis

Suppose you have implemented a program to do some data analysis and run it on a small development set. Everything seems to be working fine. Now you run your program on the much larger full set of data. You leave the program running over night expecting to see some results in the morning. But when you wake up the program is still running. It's taking forever, much longer than you expected. What has gone wrong?

One explanation is that your code has implementation issues. There could be a bug that only gets triggered by some bad data, which causes it to run slowly or loop forever. Or your program could be running out of memory causing everything to slow down. But programming bugs may not be the only reason for the observed slow down.

Not all algorithms run at the same speed. Achieving the same result in two different ways can take dramatically different amounts of time. Sometimes this is due to the way an algorithm is implemented—an experienced programmer will write more efficient code than a novice—but often it is something fundamental to the algorithms themselves. This is especially true when comparing different algorithms (even when implemented by the same programmer).

A useful way to characterise an algorithm is via complexity analysis, which roughly measures the amount of time (or space) an algorithm takes to complete as a function of the input size. For example, when sorting a list of names into alphabetical order we would expect the sorting algorithm to run for longer on lists with more names than lists with less names. (We may also expect it to take longer if we give it the list in random order compared to if we give it the list in nearly sorted order. But in complexity analysis we only care about how an algorithm scales with the size of the input, not other characteristics of the input.) The big question then is how much longer?

Trying to characterise how much longer in terms of real running time (i.e., seconds) is a difficult problem because we would need to take into account the specific implementation, the specific machine that the code is running on, other applications that may be running at the same time, etc. A more useful measure is by how much the running time changes in a relative sense. For example, does doubling the length of the list of names provided as input also double the amount of time it takes to sort the list? Such an algorithm would be called linear and would be a very good algorithm indeed—the best known sorting algorithms grow slightly more than linear so doubling the length of the input would cause the algorithm to run for slightly more than twice as long.

Big-O Notation

We only consider "big-O" notation here, which measures the asymptotic, or limiting, upper bound on running time. Other characterisations of programs and analysis methods are covered in more advanced Computer Science courses on algorithms.

Let us consider a simple example of determining whether a name exists in an *already* sorted list of names, e.g., a guest list for a party. Assume that the names are stored in a variable called `guest_list` and the name we're looking for in `query`. The most basic solution is, perhaps, to search through the entire list of names looking for a match:

Python Code

```
name_found = False
for name in guest_list:
    if (name == query):
        name_found = True

if name_found:
    print("{} is a guest".format(query))
else:
    print("{} is not on the guest list".format(query))
```

We can now ask the question: for an arbitrary name, how long does the above piece of code take to run? As mentioned above, answering this question in terms of actual time (e.g., seconds) depends on a number of factors. However, we can answer the question in terms of the length of the list of names being scanned (i.e., the number of guests on the guest list). Mentally running through the program reveals that it compares the query name to every name in the list. So for a guest list with n names the code will do n comparisons, and we would say the algorithm has $O(n)$ running time.

Can we do better? Well, yes we can. Notice that our code keeps making comparisons even once it has found a name on the list. If the algorithm stops as soon as it finds a match, as implemented in the following code, it can save a bit of time on average.

Python Code

```
name_found = False
for name in guest_list:
    if (name == query):
        name_found = True
        break
```

Analysis of the modified code shows that in the worst case (where a name does not appear in the list or appears as the last entry in the list) it would perform n comparisons, but sometimes less. (The average number of comparisons will be $\frac{1}{2}n$ if the name is somewhere in the list and exactly n if not). In terms of big-O notation we say that the algorithm is still $O(n)$. The notation ignores constants and drops lower-order terms—so if an algorithm performs $3n^2-5n+10$ operations we would call it an $O(n^2)$ algorithm.

Our implementation so far has not taken into account the fact that the guest list is already sorted in alphabetical order. We can make use of this fact and implement a faster algorithm known as *binary search*.

Python Code

```
lower_bound = 0
upper_bound = len(guest_list) - 1
while (upper_bound > lower_bound):
    middle_index = int((lower_bound + upper_bound) / 2)
    if (query > guest_list[middle_index]):
        lower_bound = middle_index + 1
    else:
        upper_bound = middle_index

name_found = (query == guest_list[lower_bound])
```

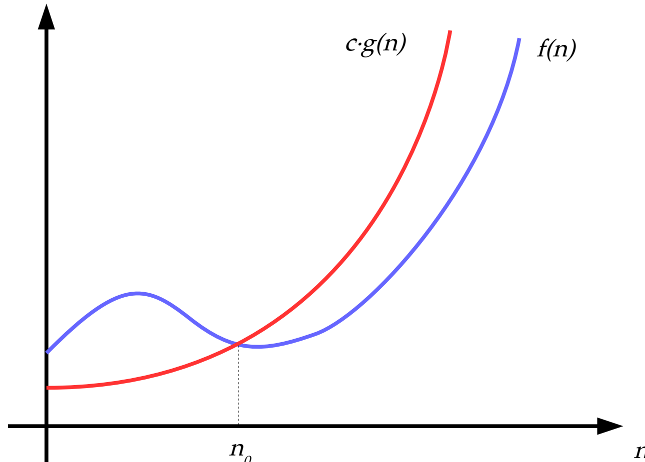


Figure 3.17: Illustration of big-O notation.

It can be shown that binary search runs in $O(\log n)$ running time for an input (guest list) of size n. Roughly speaking, this means that everytime we double the length of the list we see a linear increment in the running time. Or put another way, the time difference between searching a list of 100 names and 200 names is the same as the time difference between searching a list of 1000 names and 2000 names.

A more formal definition of big-O notation can be given as follows: *a function $f(n)$ is said to be $O(g(n))$ if there exists positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.* The definition is exemplified by the plot in Figure 3.17. Note that big-O notation defines an upper bound so every $O(n)$ algorithm is also an $O(n^2)$ algorithm. In general, we are interested in the tightest upper bound on running time.

In this lecture we have covered the very basics of complexity analysis. A more rigorous coverage of complexity theory and asymptotic analysis is studied in courses on algorithms and computation theory. We conclude with a brief discussion of one of the main unsolved problem in computer science.

Does P = NP?

One of the most famous questions in theoretical computer science concerns the equivalence of different complexity classes, that is, different categories for how fast certain problems can be solved (or how much memory they use). A problem is said to be in class P if there exists a polynomial time algorithm that solves the problem. Roughly speaking this means that there is a polynomial function $f(n)$ that bounds the number of seconds that a program for solving the problem will take to run for any size input n. Another class of problems, known as class NP, are those that have algorithms that can verify a given solution in polynomial time. For example, given the question "what is the square root of 1081600?" its easy to verify that 2 is not the solution and that 1040 is, even if finding the solution in the first place may be difficult. To verify that 1040 is a solution we simply multiply it by itself and compare to 1081600. The big theoretical question is then, "is every problem in NP also in P?" or more simply "does

$P = NP$?" For an entertaining discussion of complexity analysis and $P = NP$ see the following YouTube video:

`https://www.youtube.com/watch?v=YX40hbAHx3s`

3.9.6 Next Lecture

- Advanced Data Structures: queues, stacks, sets, dictionaries.
- Trees and Graphs

3.10 Lecture 10: Data Structures II

Learning Outcomes

- Understand the concept of a container for holding data.
- Define the operations and behaviours of standard containers: queues, stacks, sets, dictionaries.
- Understand how to build and parse trees and graphs.

Overview
This lecture introduces the concept of a container for storing data of a particular type. The abstract data types queues, stacks, sets, and disctionaries are defined, and examples given on when these data types can be used. The lecture also covers trees and graphs.

3.10.1 Containers

Containers in a programming language are objects that are used to hold collections of other objects. We have already seen some simple containers. For example, a list of names is a *list* container that stores *string* objects. Containers can even be used to hold other containers, for example, a list of lists.

A typical program will insert objects into a container, not necessarily all at once. The program will then *iterate* over the object in the container and perform some opertion on them. Different containers behave in different ways (for example, the order in which data is stored and rules for how data can be accessed) and are optimised for different uses. Often the right choice of container can make the solution to a problem simpler and more efficient.

Queues

Queues are used to process data in the same order in which the data is received (or inserted into a queue). For exmaple, a network device driver may insert network packets into a queue for processing by the operating system. This ensures that data (say a large file) is processed in the same order in which is was sent. Queues are sometimes calles FIFOs for "First In, First Out".

This is in stark contract to a list or tuple which allows access to any element at any time—a property known as *random access*.

The two most important queue operations are *enqueue* and *dequeue*, which add and remove items from a queue, respectively. These operations are illustrated in Figure 3.18. Other operations include creating (initialising) the queue, clearing the queue, and checking whether a queue is empty.

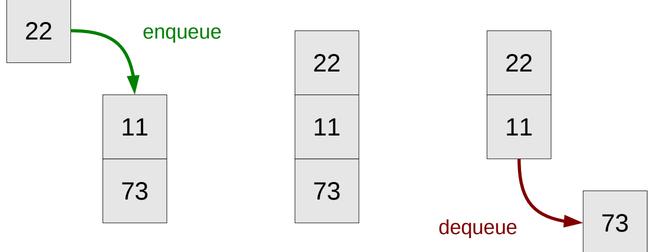


Figure 3.18: Illustration of pushing data onto and pulling data off a queue.

In Python the deque data type implements a so-called double-ended queue. It can be used to implement both queues and stacks (which we discuss next). The following code shows example usage.

Python Console
```
>>> from collections import deque
>>> q = deque([11, 73])  # initialise (alt. enqueue twice)
>>> q.appendleft(22)     # enqueue (alt. append)
>>> q
deque([22, 11, 73])
>>> q.pop()              # dequeue (alt. popright)
73
```

Note that a `list` object can be implement the same behaviour as a `deque` but is not optimised for frequent append and pop operations.

A variant of the queue data type is a *priority queue*, which associates a score, or priority, with each item. The item with highest priority remaining in the queue is dequeued first irrelevant of when it was added.

Stacks

In contrast to a queue, a *stack* is a data structure that implements a LIFO policy for “Last In, First Out”. Think of a pile of books or a stack of plates. Items can only be added to the top of the stack and items can only be removed from the top. These actions correspond to *push* and *pop* operations, respectively, as illustrated in Figure 3.19.

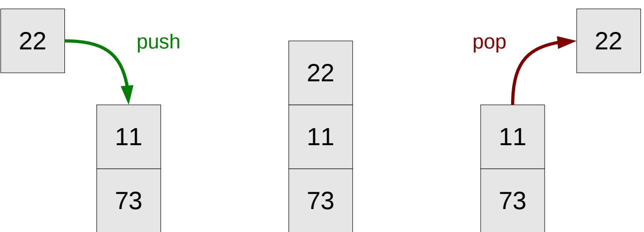


Figure 3.19: Illustration of pushing data onto and popping data off a stack.

Both the Python `deque` and `list` data types can implement a stack. We show an example using a `list`:

Python Console
```
>>> stack = [73, 11]
>>> stack.append(22)        # push
>>> stack
[73, 11, 22]
>>> stack.pop()             # pop
22
```

Most implementations of stacks and queues have a maximum size. If you try to push an item once the maximum size is reached then an exception will be raised. Likewise, if you attempt to pop an item from an empty stack (or queue) an exception will be raised. In general you should check that a stack is not empty before popping the top of the stack.

Sets

Sets are containers that are used to hold unique items, i.e., duplicate elements are not allowed in a set. Moreover, the elements of a set are unordered unlike elements of a list, but that does not stop us from being able to iterate over them. We will see later that sets are very useful for detecting duplicates in lists.

The important operations on a set are insertion, deletion, membership test and iteration. Figures 3.20 and 3.21 illustrate examples of inserting a new and existing element into a set, respectively. The following code snippet shows examples of the aforementioned set operations.

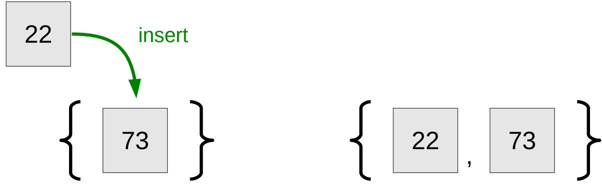


Figure 3.20: Illustration of inserting a new element into a set.

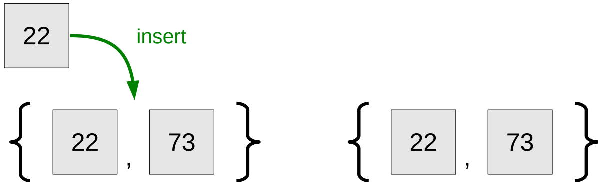


Figure 3.21: Illustration of inserting an existing element into a set.

Python Code

```
>>> s = set([73])     # s = set(); s.add(73)
>>> s.add(22)         # add an element to the set
>>> s
{73, 22}
>>> s.add(22)         # add a duplicate element
>>> s
{73, 22}
>>> 22 in s           # membership test
True
>>> 43 in s           # another membership test
False
>>> for item in s:    # iteration printing elements squared
...   print(item * item)
...
5184
484
>>> s.remove(22)      # set deletion
>>> s
{73}
```

Variants of the set container include the *multiset*, which allows duplicate elements to be added to the set, and the Python `Counter` object, which is useful for counting the number of times each item is seen. As we will see, a `Counter` can be easily implemented using a dictionary.

Dictionaries

A *dictionary* is a data structure that stores values referenced by a key. That is, we provide the dictionary with a key, such as an integer or a string, and the dictionary returns a value associated with that key. Thus a dictionary can be conceptually thought of as a list of *key-value pairs* with the important proviso that keys must be unique. Dictionaries are also called *maps*, *associative arrays* or *hash tables*.

Dictionaries are optimised to perform lookups on keys. Keys must be immutable objects, such as numbers, strings and tuples. Lists and sets cannot be used as keys, for example. But values can be anything (and we will show an example of a set value later). What makes key lookups fast is the way in which the key-value pairs are stored. Keys are *hashed* to produce a fixed length hash value (say, an integer)—for example, a string can be hashed by adding up the encodings of all its characters and ignoring integer overflow. Once hashed a key can be addressed in (near) constant time. The only thing we need to be careful about are *collisions*, i.e., two different keys which produce the same hash. Good hash functions are designed to avoid collisions but collisions are inevitible. Dictionaries work by first hashing a key to determine a bucket for the key and then searching for the exact key within the bucket.

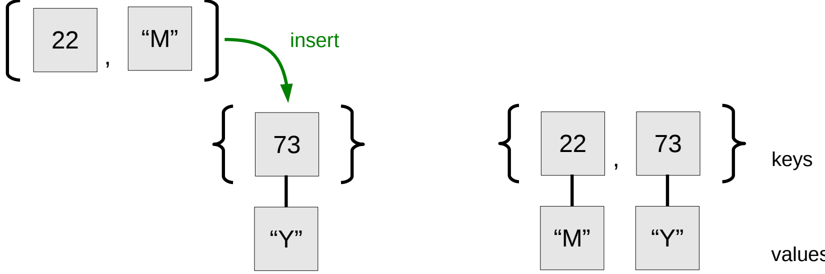


Figure 3.22: Illustration of inserting a new key-value pair into a dictionary.

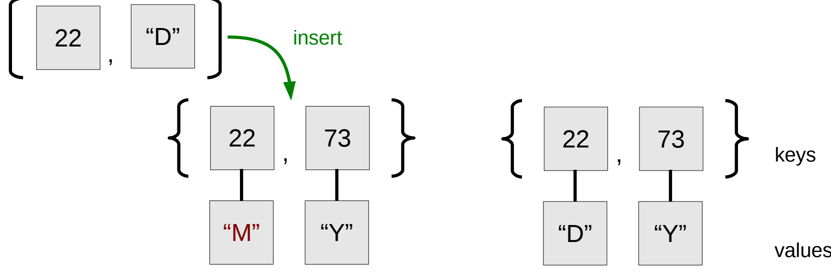


Figure 3.23: Illustration of changing the value associated with a given key.

Important operations on a dictionary include setting and retrieving the value for a given key, checking if a key exists in the dictionary (membership), and iterating over items in the dictionary. These operations are demonstrated in the code below.

A **dictionary** can be constructed using braces `{}` or `dict()`.

Python Console

```
>>> d = {73: "Y"}
>>> d
{73: 'Y'}
>>> d[22] = "M"
>>> d
{73: 'Y', 22: 'M'}
>>> d[22] = "D"
>>> d
{73: 'Y', 22: 'D'}
>>> len(d)           # number of keys in the dictionary
2
>>> 22 in d          # (key) membership test
True
>>> "D" in d
False
```

Usually we will want to iterate over keys stored in a dictionary and perform some function on their corresponding values. The following code snippet gives two examples where we simply print out key-value pairs. In what situation would we prefer one example iteration method over the other?

Python Console

```
>>> for k in d:
...   print("key {} has value {}".format(k, d[k]))
...
key 73 has value Y
key 22 has value M
>>>
>>> for k, v in d.items():
...   print("key {} has value {}".format(k, v))
...
key 73 has value Y
key 22 has value M
```

A more elaborate use of dictionaries is given below.

Example 3.10.1. Let us put some of our newly learned data structures to work in solving an interesting problem. Given a file containing a number of phrases, e.g., tweets, we would like to find phrases that are anagrams of each other. We assume that each line in the file is a separate phrase.

Our first step is to define a function to construct an anagram signature for a given phrase. We use the code developed in Lecture 4.

Python Code

```
import string

def anagram_signature(text):
    """Converts a string into a tuple of letter counts."""
    lc = ''.join(filter(str.isalpha, text)).lower()
    return tuple([lc.count(i) for i in string.ascii_lowercase])
```

Now we are ready to parse the file in order to find anagrams. We will write the code as a command line script, which will allow us to provide a filename as

a command line argument. (More about command line scripts in Lecture 3.25). As we read in each line we will compute its anagram signature and use this as a dictionary key. The value stored in the dictionary will be a set of phrases with matching key (i.e., anagram signature).

Python Code

```
import sys

# check command line arguments
if len(sys.argv) != 2:
    print("USAGE: python " + sys.argv[0] + " <filename>")
    exit()

# open file and hash signatures into dictionary
anagram_sets = dict()
for line in open(sys.argv[1]):
    line = line.rstrip('\n')        # remove line ending
    sig = anagram_signature(line)   # create signature
    if sig not in anagram_sets:     # if new signature add to dict
        anagram_sets[sig] = set()
    anagram_sets[sig].add(line)     # add to set with same signature
```

It is now a simple matter of iterating over dictionary keys and pulling out the set of anagrams associated with that key. Since the keys are anagram signatures all phrases associated with the key will are anagrams of each other. We only print the phrases if there are more than one in the set.

Python Code

```
# find all anagram sets of size greater than one
for sig in anagram_sets.keys():
    if len(anagram_sets[sig]) > 1:
        print("-" * 80)
        for line in anagram_sets[sig]:
            print(line)
```

3.10.2 Graphs and Trees

Graphs (and trees and forests) are pervasive in computer science. They are used in representing and solving many classical problems including search, game play, navigation and route planning, social networks, etc. Formally, a graph is defined as a tuple $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ where $\mathcal{V}$ is a set of nodes (or vertices) and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of edges (or arcs) between nodes. A graph with no cycles is called a tree.

Graphs can have many other properties including edge directedness, weights, cyclicity, etc.

A very common operation that is performed on a graph or a tree is to be able to traverse the nodes. For example, in playing games such as chess or tic-tac-toe, a game tree could be constructed where each node represents the game states and edges represent valid moves from one game state to another (e.g., moving a piece in chess) as illustrated in Figure 3.24. In this case traversing the game tree amounts to simulating a game being played.

There are no built-in graph or tree data structures in Python, but there are plenty of libraries (e.g., `networkx` which we saw in Lecture 1). However, for simple graphs, like game trees, it is quite simple to implement your own tree

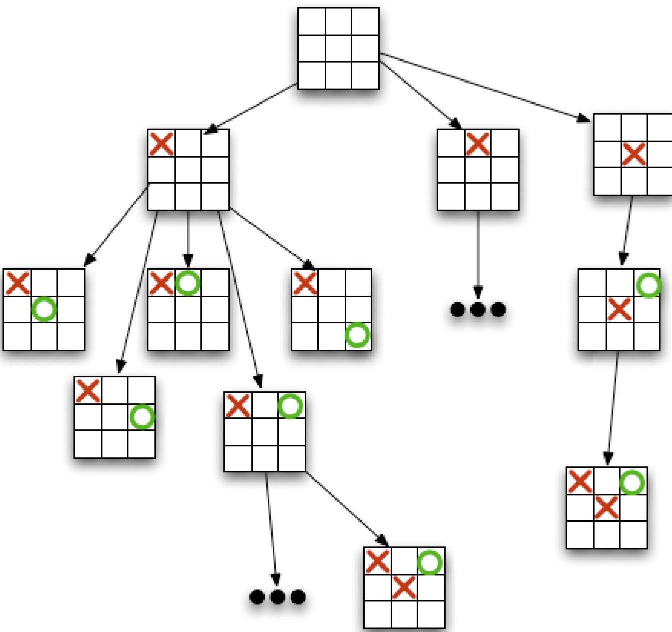


Figure 3.24: Illustration of a partial tic-tac-toe game tree. Image courtesy of `http://scienceblogs.com/goodmath/2007/09/16/games-and-graphs-searching-for/`.

data structure—you just need to keep track of parents and children for each node. Moreover, often you do not need to represent the tree explicitly at all as we demonstrate in the following search problem.[8]

Example 3.10.2. A *polyomino* is a geometric shape composed of joining one or more squares, connected orthogonally. For example, the popular Tetris game involve fitting falling tetrominoes (shapes composed of four squares). Some examples are shown in Figure 3.25.

[8]Warning: the programming in the polyominoes example is rather advanced. You are not expected to be able to write code like this from scratch at this point. However, you should be able to follow the logic of what is being done.

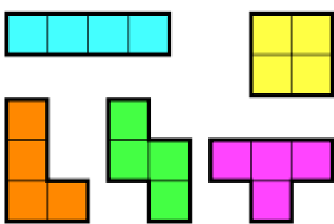


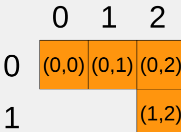
Figure 3.25: Example tetrominoes.

An amusing puzzle involves packing a pre-defined set of polyominoes into a fixed grid (allowing only rotation and translation of the polyominoes pieces) without any overlap. This is a puzzle in which computers excel. In this case study we develop code for searching over polyomino placement in order to fit ten tetrominoes (those in Figure 3.25 and their mirror images) into a 5-by-8 grid. The code we develop is illustrative; it can be made much more efficient but that is beyond the scope of this case study.

Our first step is to define the problem. We specify the size of the grid by defining variables for number of rows and columns.

Python Code
```
ROWS = 5
COLS = 8
```

Next we define the tetromino pieces. Each piece is defined as a 4-tuple of (x, y) pairs indicating the shape of the piece. The first (x, y) pair is the reference cell and always $(0, 0)$. For example, an L-shaped piece would be defined by the tuple $((0, 0), (0, 1), (0, 2), (1, 2))$ to represent



The code defining the pieces is:

Python Code
```
PIECES = [
    ((0, 0), (0, 1), (0, 2), (0, 3)),          # straight
    ((0, 0), (0, 1), (1, 1), (1, 0)),          # square
    ((0, 0), (0, 1), (0, 2), (1, 1)),          # tee
    ((0, 0), (0, 1), (0, 2), (1, 2)),          # ell
    ((0, 0), (0, 1), (-1, 1), (-1, 2)),        # ess
    ((0, 0), (0, 1), (0, 2), (0, 3)),          # straight
    ((0, 0), (0, 1), (1, 1), (1, 0)),          # square
    ((0, 0), (0, 1), (0, 2), (1, 1)),          # tee
    ((0, 0), (0, 1), (0, 2), (-1, 2)),         # jay
    ((0, 0), (0, 1), (1, 1), (1, 2))           # zed
]
```

When placing pieces we are allowed to rotate them (by increments of 90 degrees), so let's write a function that will rotate a given piece.

Python Code
```
def rotate(piece):
    """Rotate a piece by 90 degrees anti-clockwise."""
    rotated_piece = [(p[1], -p[0]) for p in list(piece)]
    return tuple(rotated_piece)
```

Now that we have rotation defined we can generate all possible shapes for packing into our grid. We know that after rotating by 90 degrees four times we are back to the starting shape of a piece. However, some pieces are symmetrical and require less rotations to get back to the start. So in addition to defining

the pieces we will specify the number of 90-degree rotations a piece can be turned before matching its original shape. This is not strictly necessary but will allow us to speed up the search by not having to consider rotations that result in a shape already attempted. For the pieces defined in PIECES we have the corresponding number of rotations:

Python Code

```
ROTATIONS = [2, 1, 4, 4, 2, 2, 1, 4, 4, 2]
```

In considering the representation of the board a two-dimensional array (or list of lists) seems most appropriate where each cell in the array indicates the identity of the piece occupying that location on the board. An empty board can then be initialised as

Python Code

```
board = [[None for y in range(ROWS)] for x in range(COLS)]
```

Our search strategy will be to incrementally place pieces on the board to occupy free locations. If we cannot place a piece we will backtrack (i.e., remove pieces) and try a different location and orientation. We can place pieces by writing a function that takes the specification for the (rotated) piece and a reference cell on the board. The function will then check whether the piece fits on the board and does not overlap with any existing piece. If so we mark the cells on the board occupied by the new piece with the index (or tag) for that piece and return the new board. Otherwise the function will return an indication that the proposed placement is not allowed—in our implementation we use the None value to indicate that placement failed. The function looks as follows:

Python Code

```
def place_piece(board, ref_point, piece, tag):
    """Add a new piece to the board at the given reference point
    and return the new board. Returns None if the piece could not
    be added at the given location."""

    # Check that reference location is free.
    if (board[ref_point[0]][ref_point[1]] is not None):
        return None

    # We need to make a deep copy of the board, otherwise we will
    # be modifying the input.
    new_board = deepcopy(board)
    for p in piece:
        x = ref_point[0] + p[0]
        y = ref_point[1] + p[1]
        if (0 <= x < len(board)) and (0 <= y < len(board[0])) and
            (board[x][y] is None):
            new_board[x][y] = tag
        else:
            return None

    return new_board
```

At this point we can informally test our implementation to make sure our

code works as expected.

Python Console

```
>>> board = [[None for y in range(ROWS)] for x in range(COLS)]
>>> place_piece(board, (0, 0), PIECES[0], 0)  # place a piece
[[0, 0, 0, 0, None], [None, None, None, None, None], ...
>>> place_piece(board, (3, 4), PIECES[0], 0)  # illegal position
>>>
```

Having the board display in a more intuitive way would be helpful. Let's write a `display_board` function:

Python Code

```
def display_board(board):
    """Displays the given board configuration."""

    for x in range(len(board)):
        print(" ".join(str(board[x])))
```

We are almost ready to start our search. Our search can be thought of as building a tree. We start with an empty board as the *root node*. We then attempt to place the first piece in our list at all possible locations and orientations on the board. Successful placements become the children of the root node. We then proceed by attempting to place the next piece in the list on each of the new boards. We call this process *expanding* a node in the tree. Included in each node is the identifier of the next piece to place. Instead of explicitly storing the tree our code will maintain a list of nodes not yet expanded. This list will be known as the *frontier*, and as we expand nodes their children get added to the frontier. If we are able to expand nodes all the way to the bottom of the tree (i.e., place the last piece) we are done and the corresponding *leaf* will contain our solution.

Here is the code for expanding a node:

Python Code

```
def add_piece_to_search(frontier, board, tag):
    """Expands a node by adding all possible moves for a piece with
    the given tag to the frontier."""

    for x in range(COLS):
        for y in range(ROWS):
            p = PIECES[tag]
            for rotations in range(ROTATIONS[tag]):
                new_board = place_piece(board, (x, y), p, tag)
                if (new_board is not None):
                    frontier.append([new_board, tag + 1])
                p = rotate(p)
```

We can now write the code to perform the search. As stated we will start with an empty board and add the first piece to all possible locations. This becomes our frontier. Then, while the frontier is not empty (i.e., there are unexpanded nodes) and we have not yet found a solution, we take a node off the frontier and expand it. Note that we have a choice on the order in which we select a node from the frontier to expand. We will adopt a *depth-first-search*

policy, which expands the last node added to the frontier at each iteration (i.e., it treats the frontier as a stack).

Python Code

```
1     # initialise stack of expanded nodes
2     frontier = []
3
4     # try first piece (expand root node) on empty board
5     board = [[None for y in range(ROWS)] for x in range(COLS)]
6     add_piece_to_search(frontier, board, 0)
7
8     # iteratively remove nodes from the stack and expand them
9     nodes_popped = 0
10    while (len(frontier) > 0):
11        board, tag = frontier.pop()
12        nodes_popped += 1
13
14        # check if we've added all the pieces
15        if (tag >= len(PIECES)):
16            display_board(board)
17            exit()
18
19        if (nodes_popped % 10 == 0):
20            print("..." + str(nodes_popped), end="\r")
21        add_piece_to_search(frontier, board, tag)
```

As soon as we are able to successfully place all pieces the code prints the solution and exits. We could modify the code to not exit and show all possible solutions. If the frontier becomes empty and we have not been able to place the last piece then the puzzle is impossible.

Here is the solution found for the 5-by-8 puzzle. A solution also exists for a 4-by-10 grid. How would you modify the code to find the solution to this puzzle?

```
[ 6 ,  6 ,  8 ,  8 ,  8 ]
[ 6 ,  6 ,  8 ,  7 ,  5 ]
[ 9 ,  9 ,  7 ,  7 ,  5 ]
[ 4 ,  9 ,  9 ,  7 ,  5 ]
[ 4 ,  4 ,  3 ,  3 ,  5 ]
[ 2 ,  4 ,  3 ,  1 ,  1 ]
[ 2 ,  2 ,  3 ,  1 ,  1 ]
[ 2 ,  0 ,  0 ,  0 ,  0 ]
```

3.11 Lecture 11: Objects and Classes

Learning Outcomes

- Understand what object-oriented programming means and how it can be used to encapsulate design concepts in your code.
- Become familiar with how Python uses classes and objects in many of its libraries.
- Understand the terms *class*, *object*, *constructors*, *fields*, and *methods* and be able to identify them in a Python program.
- Write code that constructs objects for existing classes to that can be interacted with through their fields and methods.
- Write a simple class in Python.

Overview

This lecture gives a very brief overview of object oriented programming. We cover how classes are implemented in Python and discuss how to use objects within programs.

Python supports an approach to designing software called "object-oriented (OO) programming". Although this is a widely used and important approach to building large systems, we will focus on how to *understand and use* code that has be developed in an OO style and only briefly discuss how you might write your own code in this style.

3.11.1 Classes and Instances

In object-oriented programming, data is typically organised into objects that have predefined properties and ways of accessing and modifying those properties.

While this type of organisation can be done with combination of data structures such as dictionaries and lists, there are several advantages to "hiding" the details of these structures and only allowing them to be accessed through the methods (functions) that the class defines.

Object-orientation can be a useful way to think about programming as we are predisposed to think about the world in terms of categories and sub-categories. For example, we can consider the class of *animals* as the collection of things that have (amongst other things) an age. Figure 3.26 shows how specific instances of animals can be thought of as instances of the class of animals and how their various properties differ.

3.11.2 Classes and Objects in Python

In Python these properties are called *fields* and the functions used to manipulate or inspect an object are called *methods*. In other programming languages these might be called *member variables* and *member functions*, respectively. A collection of fields and methods is what defines a *class*. Instances of a class are *constructed* from this class definition.

It is sometimes said that "everything in Python is an object". What this means is that integers, strings, lists, even functions, are all members of different classes and all those classes are subclasses of the top-level "object" class.

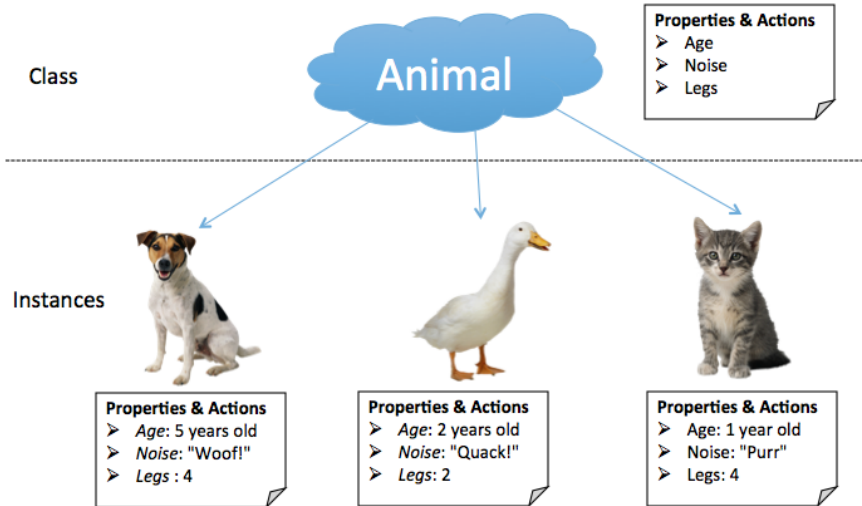


Figure 3.26: Specific animals such as a 5 year old dog can be thought of as "instances" of the "class" of animals.

At a terminal or in the PyCharm console, you can determine the class of anything using the `type` function.

Python Console
```
>>> type('Some words')
<class 'str'>

>>> type(17)
<class 'int'>

>>> type([1,2,3])
<class 'list'>
```

You would have already used several methods belonging to the string and list classes, such as `sort` and `append` on lists and `islower`, `split`, and `format` on strings. The slightly unusual thing about these very common types is that their constructors are "built-in" to the Python language: lists can be build with the `[1,2]` syntax and strings by surrounding text with single- or double-quotes.

Constructors, Methods, and Fields

Normally, if `MyClass` is a class, its constructor is called like so: `x = MyClass()` or, if parameters are used, like so: `x = MyClass(param1, param2)`. The instance of `MyClass` assigned to the variable `x` will then have all the methods and fields declared in the definition of `MyClass`. As we saw in Lecture 3, the methods belonging to `x` can be called like function by using the "dot" notation: `x.somemethod()`. Fields are access using the same "dot" convention but without the parentheses: `x.somefield`.

By convention, class names are in "CamelCase".

We will see several more examples of constructors, methods, and fields, as well as how to define them, in the remainder of this lecture.

3.11.3 Why Use Classes?

As mentioned earlier, classes enable a programmer to hide implementation details and present a coherent interface for working with a particular "shape" of data. Being able to abstract away parts of a system that are not important to a programmer outside that system is one of the central programming tools when developing, managing, and reasoning about large software systems.

Case Study: Calendar Dates

To make this point more concrete, let us consider how we might handle working with calendar dates in software.

The rules for the number of days in each month (especially in February!) are fairly complicated and is the type of thing that should be "solved" (i.e., programmed and tested) once and then not worried about any more. This is exactly what the `datetime` library in Python does.

Python Console

```
>>> from datetime import date, timedelta
>>> d = date(2015,1,12)
>>> d + timedelta(15 * 7)
datetime.date(2015, 4, 27)   # April 27th, 2015
```

You can see here that we are able to construct a `date` object `d` representing the 12th of January, 2015, and then add 15 weeks (15×7 days) to work out the result (April 27th, 2015).

Most languages come with some kind of calander or date and time library as standard and so you should never need to work with dates "by hand". However, it is instructive to consider just how much complexity is hidden in this class.

We can consider at least three different approaches to working with date calculations like the one above if we were not allowed to use `datetime`.

In computing lore, this date is common as a reference point and is called the "UNIX epoch".

1. *Represent each date as the number of days since January 1st 1970*: One advantage of this representation is that doing calendar arithmetic is easy — you just add or subtract the number of days from the date you have. The downside is that convert to or from this representation is hard. For example, how would you work out what date 10,000 days from Janurary 1st, 1970 corresponded to?

2. *Use different variables for year, month, and day*: The updside of this representation is that it does not require any data structures or data conversion. The downside is that it is up to you to keep the values in those variables in sync (e.g., if `month = 4` (April) and you add one to `day = 30` then `month` needs to be incremented to 5 (and `day` should become 1).

3. *Use a dictionary with keys 'year', 'month', and 'day'*: This makes it possible to keep the variables from the previous approach together and would make it possible to write functions like `addDays(date, days)` that takes and returns this dictionaries. However, there is no guarantee with dictionaries that the values associated with the keys represent valid dates to begin with.

Classes provide several advatnges over all the above approches. Constructors can be used to guarantee that only valid dates are built (e.g., by raising an error if someone tries to create "the 30th of February"). Multiple ways of constructing dates can be provided. The method on the class can then hide the details of calculations such as adding and subtracting weeks or days. Methods that convert between different representation can also be provided (e.g., `d.daysSinceEpoch()` could return the number of days from 1 Jan. 1970 to `d`).

The nice thing about the existence of libraries like `datetime` in Python is that, since someone has already thought about and made these trade-offs, you can just make use of the resulting class in your own programs without having to worry about all the messy details. All you have to do is read the documentation for the class to understand how to create and work with its instances.

3.11.4 Writing Your Own Classes

To write your own class in Python, you need to define the constructor for its instances and its methods. The basic template is shown below for a class called `ClassName`.

Python Code

```
# The Class syntax
class ClassName:
        # Class variables
        cvar = some_value

        # Constructor
        def __init__(self, arg1, arg2):
                # Setting an instance variable
                self.ivar = arg1 + arg2

        # A method
        def method1(self, arg1):
                return self.ivar / cvar
```

The first thing to note is that all of the components of the class are indented so that they sit "under" the class declaration `class ClassName`. This is yet another situation where spacing is important in Python.

On line 4, we see what is called a *class field*, which is a variable that is shared between all instances of a class. This is in contrast to *instance fields* which always appear within a class prefixed by `self` (more on this below).

A class's *constructor* (line 7 above) is always called `__init__` and, like other methods (such as `method`) in this class, are defined in the same way functions are defined elsewhere in Python. Although the constructor method is called `__init__`, you never call this method by this name outside the class itself. If you want to create an instance of this class and assign it to `x` you would write `x = ClassName(1,2)`. This has the effect of calling `__init__(self, 1, 2)` and creating a new instance of `ClassName` with the *field* `ivar` set to 3 (i.e., `arg1 + arg2`).

The double-underscore convention is used to indicate methods that play special roles within Python.

The key thing to note with class methods is that the first argument to every method is `self` which can be followed by any other arguments to the method. This `self` variable is used within the definition of the method to refer to the instance the method was called on.

More concretely, suppose we created the instance `x = ClassName(1,2)` as above and then invoked `x.method1(7)`. What Python does is call the function `method1(self, 7)` where `self` is set to the same instance that `x` refers to. This is a little confusing at first but you do eventually get used to mentally substituting a call like `x.method()` into `method(x)` as the execution passes into the class. The following example will hopefully make this clearer.

Example 3.11.1. In this example we want to be able to incrementally summarise collections of numbers by their mean and variance. Recall that the mean of a collection of n numbers $x_1, x_2, \ldots, x_n$ is defined to be

$$\bar{x} := \frac{1}{n}\sum_{i=1}^{n} x_i$$

and the (biased) variance for that sample is

$$\hat{\sigma}^2 := \frac{1}{n}\sum_{i=1}^{n} (x - \bar{x})^2 .$$

For example, if $n = 3$ and the numbers are $1, 2, 3$ then the mean will be $\frac{1}{3}(1 + 2 + 3) = 2$ and the variance will be $\frac{1}{3}\left((1-2)^2 + (2-2)^2 + (3-2)^2\right) = \frac{2}{3}$.

Given a list of numbers it would be easy to write functions that calculate these statistics. However, we would like some way of keeping track of statistics as more and more numbers are added. The follow class lets us do exactly that by keeping all numbers it has been presented with in a list and then calculating the statistics on the entire list whenever they are needed.

Python Code

```
1  class Summary:
2      """
3      A summary incrementally keeps track of key statistics such as
4      the number, mean, and variance of a collection of numbers.
5      """
6      # A class variable that keeps track of the number of instances
7      num_summaries = 0
8
9      def __init__(self):
10         """Create a empty summary with a unique ID."""
11         self.numbers = []
12
13         Summary.num_summaries += 1
14         self.id      = Summary.num_summaries
15
16     def add(self, x):
17         """
18         Add x to this summary and update the statistics.
19         :param x: The number to add.
20         """
21         self.numbers.append(x)
22
23     def addAll(self, xs):
24         """
25         Add a list of numbers to this summary.
26         :param xs: A list of numbers.
27         """
28         for x in xs:
29             self.add(x)
30
31     def statistics(self):
32         """
33         Returns the count, mean, and (biased) sample variance of the
34         numbers added to this summary.
35         :return: The triple (count, mean, variance).
36         """
37         n = len(self.numbers)
38         mean = sum(self.numbers) / float(n)
39         var = sum([(x-mean)**2 for x in self.numbers]) / float(n)
40         return (n, mean, var)
```

This class can now be used as follows. Notice that new points can be added to a summary using `addAll` or `add` and that the ID of each summary is incremented whenever a new summary is created.

Python Console

```
>>> s = Summary()
>>> s.addAll([1,2,3])
>>> "Summary #{}: {}".format(s.id, s.statistics())
'Summary #1: (3, 2.0, 0.6666666666666666)'
>>> s.add(4)
>>> "Summary #{}: {}".format(s.id, s.statistics())
'Summary #1: (4, 2.5, 1.25)'
>>> s2 = Summary()
>>> s2.addAll([1,1,1,1,1])
>>> "Summary #{}: {}".format(s2.id, s2.statistics())
'Summary #2: (5, 1.0, 0.0)'
```

Let's walk through what the above console commands do.

When `s = Summary()` is called the method `__init__(self)` inside the `Summary` class is called and the `numbers` field is initialised to an empty list in line 12. The class field `Summary.num_summaries` is incremented by one on line 14 and then its value is stored in the instance field `id` in line 15. This means the value of `s.id` is 1 after the constructor finishes.

When `s.addAll([1,2,3])` is called the method `addAll(self,[1,2,3])` on line 24 is called within the `Summary` class and `self` is assigned to the same object as `s`. You can see that this method loops through the values in `xs` and calls the method `self.add(x)` for each. This, in turn on line 22, appends the value of each `x` to the list that is referred to by the instance field `numbers`.

Finally, when `s.statistics()` is called, you can see on lines 38–40 that the statistics for the list of numbers in `numbers` is computed and returned as a triple.

The above example shows how a particular functionality can be encapsulated in a class. All a user of this class needs to know is how to create instances with the constructor and what the methods for adding new numbers and calculating statistics do.

In the next example, we will see how we can leave the *interface* to this class exactly as it is but improve the way the various methods are implemented.

Example 3.11.2. The implementation of summaries in Example 3.11.2 is a good first attempt but it has several drawbacks. The major among these is that if we were to use it for a large number of samples we would quick run into problems with memory and the time it takes to compute and recompute statistics.

In this second implementation of the same class, we rely on the fact that the mean and variance can be computed incrementally. The mean because it is just a sum and the variance because its definition can be manipulated to prove that

You might want to try to prove this fact yourself.

$$\hat{\sigma}^2 := \frac{1}{n}\sum_{i=1}^{n} x_i^2 - [\bar{x}]^2 .$$

This means that we can compute the mean and variance if we keep track of n (the number of samples), the sum of the values (for the mean), and the sum of the squares of the values. The following code does exactly this.

Python Code

```python
class Summary:
    """
    A summary incrementally keeps track of key statistics such as
    the number, mean, and variance of a collection of numbers.
    (This version requires constant space and time).
    """

    # A class variable that keeps track of the number of instances
    num_summaries = 0

    def __init__(self):
        """Create an empty summary."""
        self.count = 0
        # Keep track of sum and sum of squares.
        self.sum         = 0.0
        self.sum_squares = 0.0

        Summary.num_summaries += 1
        self.id          = Summary.num_summaries

    def add(self, x):
        """
        Add x to this summary and update the statistics.
        :param x: The number to add.
        """
        self.count += 1
        self.sum   += x
        self.sum_squares += x**2

    def addAll(self, xs):
        """
        Add a list of numbers to this summary.
        :param xs: A list of numbers.
        """
        for x in xs:
            self.add(x)

    def statistics(self):
        """
        Returns the count, mean, and (biased) sample variance of the
        numbers added to this summary.
        :return: The triple (count, mean, variance).
        """
        n    = self.count
        mean = self.sum / float(n)
        var  = self.sum_squares / float(n) - mean**2
        return (n, mean, var)
```

What the previous two examples show is that classes are a practical way to “hide” implementation details from developers who use your class. All they need to understand is how the *interface* to a class works—that is, the constructor, fields, and methods. Of course, the implementation details (such as the time

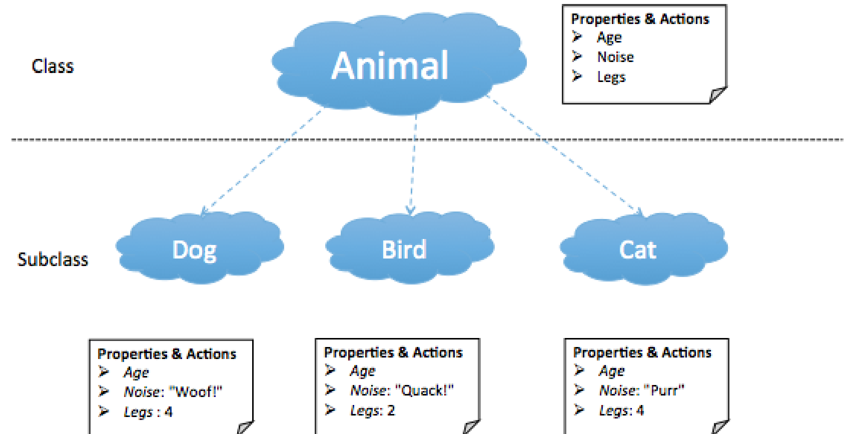


Figure 3.27: The classes "Dog", "Bird", and "Cat" can be thought of as *subclasses* of the class "Animal".

and space complexity of the methods) matter as well but it is sufficient to put these concerns in the documentation of a class so anyone using it is aware of them without having to read through the code.

3.11.5 Subclasses and Inheritance

Another way objects can be used to structure your code using classes is through *inheritance*. This is when one class (the "subclass") derives or inherits some of its properties from another class (the "superclass" or "parent class"). An analogy that might be useful here is shown in Figure 3.27. Dogs, cats, and birds are all subclasses of animals that inherit properties like age, noise, and legs from the parent class Animals. Note, however, that the subclasses can set properties that are common to all the instances in the subclass (e.g., all dogs make the noise "Woof!").

In Python, subclasses are created by adding the name of the superclass in parentheses in the subclass's definition. For example, if we wanted the class `Dog` to inherit from the class `Animal` we would write:

Python Code
```
class Dog(Animal):
        # Class definitions go here...
```

In the following example, we show how the summary class we built earlier in this lecture can be extended to add a `name` property to each summary. The thing to notice here is how little extra code is need to add this feature. All of the definition of the parent class is available in the subclass "for free".

Example 3.11.3. Suppose we wanted to add names to the instances of Summary that we create. One way to achieve this would be to edit the definition of the `Summary` class to add an extra `name` field that can be set in the constructor.

However, if the `Summary` class was defined in another library that we did not want to modify, we can still easily add extra features such as a name by taking advantage of *subclassing* in Python.

The following code shows how we can write a new `NamedSummary` class that *extends* the `Summary` class. To do so, all we need to do is put `Summary` in parentheses in the first, class definition line, and then provide a constructor that takes in a `name` argument and stores it.

Python Code

```python
class NamedSummary(Summary):
    """Extension of the Summary class that adds a name to each
    summary."""

    def __init__(self, name):
        """Create a new summary with the given name."""
        super().__init__()  # Initialise fields in Summary
        self.name = name    # Store the given name
```

There is one critical step in the definition of this constructor. To ensure that all of the fields in the original `Summary` class are correctly initialised, the `__init__()` constructor for the `Summary` class must be called from within the `NamedSummary` constructor. This is done by using the built-in `super()` method in line 7.

3.11.6 Classes and Variable Namespaces

We saw in Lecture 4 that variables have a scope, that is, are accessible in some parts of your code and not others. Moreover, namespaces are used to avoid collisions (two different variables having the same name). This can get very confusing when dealing with classes because of the various namespaces (class, instance, function, etc). Here we demonstrate a piece of code with multiple different variables all called `name`.

Python Code

```python
class ANewClass:
    """A simple class to demonstrate variable scoping rules."""

    # define a class variable (accessible from all classes)
    name = "class variable"

    def __init__(self, name):
        # define an instance variable and assign it the given name
        self.name = name
```

The first variable is the *class variable* `name` defined on Line 5. This variable is common to all instances of class `ANewClass`. The second variable is the *constructor argument* `name`, which only exists within function `__init__()`. The third variable is the *instance variable* `self.name`, which only exists in instances of this class. In fact, each instance of the class will have its own copy of `self.name` as we show in the following code.

Python Code

```python
# create a global variable and two instances of ANewClass
name = "global variable"
instance_one = ANewClass("name for first instance")
instance_two = ANewClass("name for second instance")

# print out the contents of each "name" variable
print(name)
print(ANewClass.name)
print(instance_one.name)
print(instance_two.name)
```

The code above also includes a fourth variable `name` with *global* scope. This variable is accessible everywhere unless masked by a local variable with the same name.

The confusion that this example creates can be alleviated by giving the variables different names, as shown in the following code. Within class methods the instance variables are still referenced using the `self.` prefix, but outside the class they are accessed using the variable for the given instance.

Python Code

```python
class ANewClass:
    """A simple class to demonstrate variable scoping rules."""

    # define a class variable (accessible from all classes)
    class_name = "class variable"

    def __init__(self, arg_name):
        # define an instance variable and assign it the given name
        self.instance_name = arg_name
```

Python Code

```python
# create a global variable and two instances of ANewClass
global_name = "global variable"
instance_one = ANewClass("name for first instance")
instance_two = ANewClass("name for second instance")

# print out the contents of each "name" variable
print(global_name)
print(ANewClass.class_name)
print(instance_one.instance_name)
print(instance_two.instance_name)
```

3.11.7 Beyond Simple Classes

There are many more nuances to consider when writing classes, using inheritence, and following general OO programming principles. These are all covered in more advanced programming courses.

3.12 Lecture 12: Tools and Practices

Learning Outcomes

- Appreciate the value of being able to "sandbox" code and data, i.e., creating independent, isolated copies that can be discarded so as to allow for "fearless" exploration.
- Understand how distributed revision control systems (such as git) work and enable several concurrent developers to contribute to the same code.
- Be aware of issue tracking systems (such as GitLab's) and how to use these effectively to report problems with or make suggestions to improve other people's code.
- Understand how to write and run unit tests using Python's testing framework and how tests can be used to isolate errors and support making changes to code.

Overview

In this lecture we cover standard software development tools and practices. We discuss sandboxing, revision/version control, issue tracking and testing. We assume that students have been exposed to all of these topics earlier but will give the topics a more formal treatment here.

3.12.1 Revision Control with Git

In Lecture 3, we saw how to manage getting code from GitLab using PyCharm. You saw how to *fork* a repository on the GitLab server to your own account, *clone* it to your own computer, edit the code and *commit* the changes, and finally how to *push* those changes back to your GitLab account.

In this lecture we will go into some more detail about these operations so that you understand more about what is going on when you work with revision control systems like git. Having seen some of the concepts behind distributed revision control systems such as git should make it easier for you to find help should you run into trouble when using these systems. A later lecture will cover even more advanced topics such as branching and merging.

Repositories, Working Directories, and Staging

When you first saw how to work with repositories, a distinction was made between "local" repositories and "remote" repositories. This distinction is still valid but the code and edits you make on your local machine can actually move between several places as you work on it. Figure 3.28 shows three conceptual "places" that your code can move between:

- **Working Directory**: This is a directory on your computer where you can open files, edit them, and save them.

- **Local Repository**: This is where all the history of the changes to your code is recorded on the same machine as the working directory.

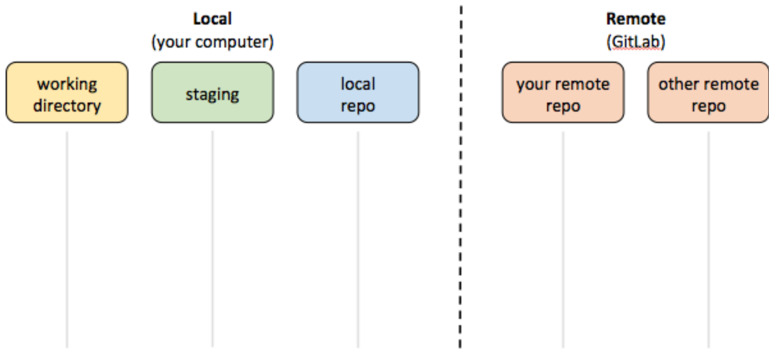


Figure 3.28: Operation in git move code changes between various places. Some are "local" (i.e., on your computer) while others are "remote" (i.e., on a server).

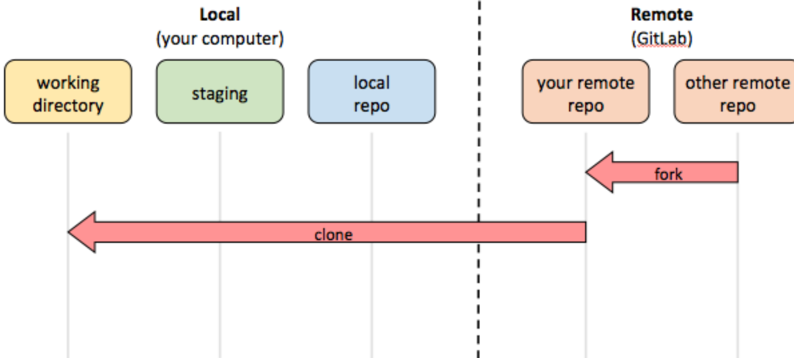


Figure 3.29: Forking copies a remote repository to a new remote repository. Cloning copies a remote repository to a local repository.

- **Staging Area**: This is an "in between" space where you can organise how you commit the changes you have made to the files in the working directory before saving them in the local repository.

We will clarify how these different places are used when we discuss the various operations for working with your code below.

Forking and Cloning

Forking and cloning are both ways to copy the contents of a repository (i.e., its files and commit history) from one repository to a new one. The main difference between the two operations is where the new repository is created. Figure 3.29 shows the difference diagrammatically.

Forking is an action that happens on a server such as GitLab, GitHub, or BitBucket and is typically done through a web interface. It is used to duplicate the contents and commit history of a repository in one person's account on the server and make a new copy of that repository in the account of the person who requested the fork.

After you have forked a repository it is like any other repository on the server

that is under your control. You can clone it (see below) and make changes to it without affecting the repository that it was forked from. This is useful for when you want to work on someone else's code without pushing your changes back to the original repository.

There is one difference: you can make "pull requests" to the repository you forked. More on this in a later lecture.

Cloning also copies the entire contents and history of a repository but the difference is that it copies it to a repository on your laptop or desktop (i.e., it makes a "local" copy). Cloning also copies all the latest versions of the files in the repository into your working directory so you can begin editing them.

Example 3.12.1. Suppose Jane has put a project of hers up on GitLab and has called it "Data Analysis". If you wanted to build upon that code you could first *fork* it, thereby making a new project in your GitLab account with all the files and history from Jane's version of the project. You could then use PyCharm (or other tools) to *clone* the repository you just created using the fork to your laptop. Once you do this, you will have a copy of all the files from the "Data Analysis" project on your laptop and you can begin editing them.

Add, Commit, and Push

Once you have some code in your working directory you can start editing it. Every time you save a change, git keeps track of the differences between the files in the working directory and those in the local repository. Once you have finished editing and want to save the changes you made into the local repository you must first "stage" the changes you want to commit. The reason for this is sometimes during an editing session you may make several different changes to a variety of files. For example, you may have fixed a bug in one function and added a new class to build a new feature for another part of your code. Staging allows you to record these changes separately. That is, you can save the bug fix edits separately to the edits that added the new class. This makes it easier for both you and other who might look at your repository to see how the code has changed over time.

Once the changes you wish to save are organised in the staging area, you can record them to the local repository using the *commit* command. The files that had their changes committed are no longer marked as modified in your IDE or on the command line. Once all outstanding modifications are added and committed the repository and working directory are said to be "up to date". Figure 3.30 shows these operations as a diagram.

Command line interfaces will be covered in a later lecture.

Note that in PyCharm, the add and commit steps are handled together by the user interface. When you select files in PyCharm to commit you are effectively adding them. Once you add a commit message and hit the "Commit" button these are written to the local repository.

Often you will want your changes to be reflected in the remote repository. This has several advantages. First, it provides a backup should something untoward happen to your machine. Second, it makes your code accessible to other developers (or from other machines). To do this you need to "Push" your changes to the remote respository. PyCharm provides a convenient "Commit and Push" option which both commits changes to the local repository and then

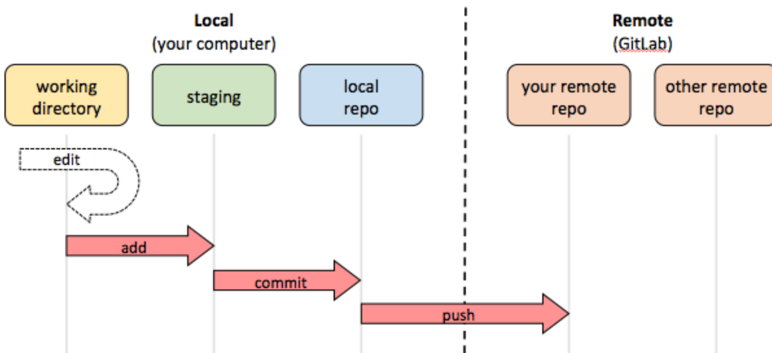


Figure 3.30: After you make edits, the *add* operation stages your changes. The *commit* operation saves these to your local repository. Performing a *push* sends the changes to the remote repository.

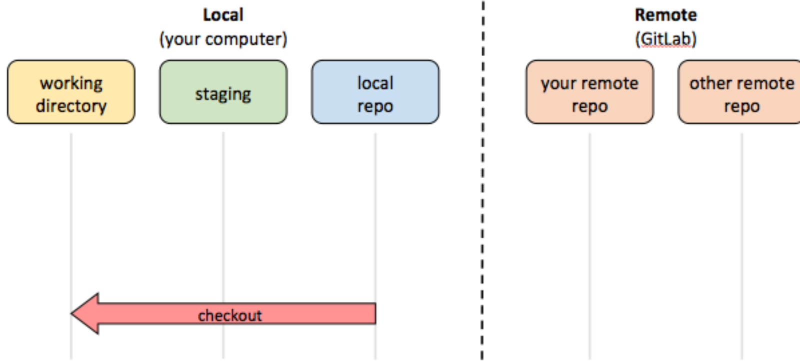


Figure 3.31: Performing a *checkout* copies files from the local repository to your working directory.

pushes those changes to the remote repository in one operation.

Checkout

The final operation we will cover in this lecture is the *checkout* command. This operation copies files from the local repository to the working directory. This can be useful if you have made changes to your files since your last commit that you do not want to keep. Being able to make changes without destroying the original versions of the files is a useful way to implement "sandboxing".

Checking out a version of your code from the local repository is shown in Figure 3.31

3.12.2 "Sandboxing"

Sandboxing is a general term used to describe a way of marking off an area to "play" in. When applied to programming, this means making a copy of your code so that you can make changes to it without worrying about loosing existing, working code. The idea here is to be able to play with new ideas,

experiment with your code and data, and generally try things out. Once you have finished experimenting, you can "move out" of the sandbox and return to the code exactly as it was before you started experimenting.

The simplest way to make a sandbox is to simply use your operating system to make a copy of the directory that holds all your code. You can then work on the files in the new directory without modifying the originals. When you want to go back to the original files, just pull up the original directory.

Copying directories or files is a perfectly good way to sandbox. However, version control systems and their support in IDEs provide several advantages. These include being able to go back and forth "through time" to different versions of your code, and the ability to easily compare the current state of your files with older versions.

Sandboxing in PyCharm

The simplest way to practice sandboxing in PyCharm is to simply add and commit the current state of your code. Once this is done, the local repository has a copy of the code that you can go back to at any time in the future. Any changes you make to the code after it has been committed with be displayed in the "Version Control" tab (use *VCS* → *Show Changes View* in the menu).

In later lectures we will see how *branches* let you keep multiple versions of your code history.

If you wish to go back to the original version of your code, you can use the *revert* command within PyCharm to do so. This command simply executes the git checkout operation described in the previous section. An easy way to run the revert command is right-click on a file in the "Version Control" panel and choose "Revert..." (you can also right-click on a file in the "Project" view on the left of the IDE and select "Git → Revert...").

Once you have selected "Revert..." you will be shown a list of modified files with the one you selected checked. When you press the "Revert" button the selected files with go back to the state they were in at the most recent commit.

IMPORTANT: Reverting files will permanently discard your most recent changes! If you wish to keep these changes you should add and commit them to the local repository.

3.12.3 Issue Tracking

Much of software development is about communication. Programmers regularly interact with each other through social media, blogging, technical forums, email, etc. When developers have problems with specific projects or collections of code a form of communication usually takes place through *issue trackers* and many (public and private) issue trackers are integrated into code management tools such as GitLab, GitHub, JIRA, and Mozilla's BugZilla.

There are two main types of issue when it comes to software projects. A *feature request* is made by a user of a software project when he or she wants the project's code to behave differently or in some new way. Typically, these changes are fairly incremental, for example, adding an extra button or field to a web page's sign up form. The more common type of issue is a *bug report.* These are made when a project's code is broken in some way or not behaving as a user expects. Both types of issue are essentially a request for the owner or owners of the project to modify the code. Owners of projects can also raise issues, which

can be a useful way to organise development (and document and remember to fix bugs that they themselves find).

Several different types of issue tracker exist, but at their core they all offer the ability for users or developers to:

- **Open** an issue on a particular project.
- **Discuss** an open issue.
- **Close or resolve** an issue.

Issue Etiquette

When reporting a bug or requesting a feature the most important thing to keep in mind is that you are asking someone to give up their time to help you. Most developers are happy to receive reports and requests as it helps them to improve their code for themselves and for others. However, for most projects you will find on the web, these developers work on their projects for free so try to do the following:

- **Be specific**: Describe what OS, language version, and libraries you are using as part of your bug report or feature request. Where possible, describe the steps you took to manifest the bug or exactly what you would like your new feature to do.
- **Be courteous**: Many developers create and support their code in their free time without pay so please treat them politely and with respect. If you are reporting a bug, don't get angry at the developer — bugs happen and if you get a developer offside, the chances are he or she will not want to fix your problem. Finally, Don't forget to say "thank you"!
- **Be helpful**: Try to help the developer as much as possible. Download the code in question and try to fix it or add the feature yourself. Failing that, try to write some test or example code for the issue you have.

3.12.4 Unit Testing

Unit tests are small pieces of code that are run to check whether other parts of your code are working as expected. So as not to introduce bugs into your tests, the aim is to keep the unit tests as small as possible. Usually, this involves applying the function to some known inputs and checking whether the output of that function is correct for those inputs. For example, if you had a function `double(x)` that doubled the value of `x` then your unit tests might check that `double(2) == 4`, that `double(0) = 0`, and that `double(-1) = -2`. In this case the implementation of `double` is so simple you would not bother writing a test, but for more complicated functions they become an incredibly useful tool for writing — and maintaining — good code.

The Python `unittest` Library

You can write a simple collection of tests just like the example above where `==` was used to test the output of `double` on known inputs. However, Python

comes with a simple unit testing library called `unittest` that makes it easy to organise and run your tests.

The `unittest` library provides a class called `TestCase` along with a number of methods to help write and run your tests. To create a collection of test, you start by extending the `TestCase` class and then defining methods on that class with tests in them, like so:

Python Code

```
import unittest
class MyTests(unittest.TestCase):

        def test_double(self):
                self.assertEqual(double(2), 4)
                self.assertEqual(double(0), 0)
                self.assertEqual(double(-1), -2)
```

The above code runs the same test cases as described in the earlier example. The main advantage of writing tests this way is that they can be run and checked more easily. For example, in PyCharm, you can run all the tests in a collection (or *suite*) by right-clicking on the class name `MyTests` and choosing the "Run Unittests" command. You can also run individual tests within the class by sleecting the name of the test (e.g., `test_double`) and choosing the "Run Unittest" command from the menu.

In the above example we only used the `assertEqual` method. This takes in two arguments and tests whether they are equal. There are a number of other methods in the `TestCase` class that you can use to write tests, including:

- `assertNotEqual(a,b)`: Test whether `a` and `b` are not equal.
- `assertTrue(expr)` and `assertFalse(expr)`: Test whether the expression evaluates to `True` or `False`, respectively.
- `assertIsNone(x)`, `assertIsNotNone(x)`: Test whether `x` is/is not the value `None`.
- `assertIn(x,xs)`, `assertNotIn(x, xs)`: Test whether `x` is/is not in the collection `xs`.

The details of how to use `unittest` and its other methods can be found here: `https://docs.python.org/3/library/unittest.html`

3.13 Lecture 13: Reading Source Code

Learning Outcomes

- Appreciate the importance of being able to read code effectively.
- Know how to find a good starting point when reading a codebase.
- Know how to read code actively by writing small examples and adding debugging statements in unfamiliar code.
- Understand how IDEs can help in navigating a large codebase.
- Understand what programmers expect to see when reading code and how to make your code readble.

Overview

This lecture covers the important skill of reading and understanding source code written by someone else. We give some hints on how to go about exploring a large codebase, and how you should write your own code to make it easy for someone else to navigate.

3.13.1 The Importance of Reading Code

Like other many other disciplines, spending time understanding good examples of your craft is a great way to improve. The craft of programming has been continually evolving ever since early programs were punched into cards: machine code was replaced by assembler then FORTRAN, C, and later languages like Python. The direction of this change has been almost invariably towards making programming languages easier to understand by humans. As Abelson and Sussman note in the beginning of their classic book *The Structure and Interpretation of Computer Programs*, "Programs must be written for people to read, and only incidentally for machines to execute."

A paper from the 2006 International Conference on Software Engineering titled *Maintaining Mental Models: A Study of Developer Work Habits* studied the way in which 157 Microsoft employees approach programming. It showed that professional programmers spend as least as much time trying to understand code as editing, writing, or designing it. Of the time spent understanding code, over 40% is spent simply reading code.

Although some of the characters are the same, reading code is very different process to reading a novel or textbook since code is usually not written to be read from start to finish. Instead, different parts of a program are spread across files and groups according to the data they work on or whether they do similar things. This means reading code is not going to be a linear process, and so it is import to know how to find a way into code that you are planning to read.

Don't panic! You should not expect to understand a piece of code in a single read through it. It takes time, thought, and usually several passes through some code before it begins to make sense. This is especially true in large codebases where the role of one piece of code may not start to make sense until you have read several other pieces.

3.13.2 Strategies for Reading Code

There are many ways to read code effectively and how you approach reading code will often depend on what you are hoping to achieve. If you are trying to use a library, you made only need to understand and tweak a couple of examples snippets of code and ignore the rest. If you are planning on making a significant contribution to a large, existing codebase then you will need to carefully understand how various pieces of code fit together.

What follows are some general tips you may want to use to help you approach reading code. Ultimately though, you should figure out what works best for you.

Where to start?

Most software projects and libraries have some kind of presence of the web. This can range from a repository on GitHub or BitBucket with a short README file to a dedicated web site with many learning resources.

If you are planning to use or modify some code, it is worthwhile quickly searching the web to see whether there are some examples or tutorials that can help you find a good starting point. Occasionally, a project may have a "quickstart" guide that walks you through downloading and installing the software, testing that it works, and then presenting some code that makes use of its main features.

If the code you want to read does not have a tutorial or quickstart guide, have a look for any other documentation for the code such as an API Reference. These typically explain how to use functions or classes in the codebase and can sometimes include examples that you can use.

If the project only presents the code without any introductory material then you should try to find where the "top" of the codebase is. This is sometimes the file that you must import to use the library or the script you must run to make the software work. From this starting point you can try to understand what classes and functions are called first and then track down the files they reside in.

Mental Modeling

Once you have found a piece of code that you want to understand, you will need to read it and its documentation carefully to understand what it does and how it is to be used. A good first step here is to make a note of what *type* of input the function of class constructor requires. Numbers? Strings? Lists of objects? Instances of certain classes? Figuring this out may require reading other parts of the codebase to figure out how to correctly build a collection of arguments to the function you want to use.

For example, if you want to use a plotting function but set the colours yourself, you may see that one of the arguments to the plotting function is a `ColorMap`. In order to build a `ColorMap` you will have to track down its code or documentation to find out how it is constructed.

Once you have figured out what inputs are required, you can mentally step through what you expect the code you are reading will do with those inputs. For some pieces of code, this will require some effort and cross-referencing of documentation of functions and methods across several files.

Active Reading

If you are having trouble understanding what a piece of code is supposed to do by mentally executing it, you can try actually executing it!

Open a new file and try to build a small, simple use case of the code you want to use by setting up the required inputs and then calling the function. You will very likely get some error messages the first time you try this but don't despair. Try to understand what the error messages are saying (searching the web for the text of an error message can help here) and then fix it. Common errors here include: not importing a required library, not initialising an instance properly, or using the wrong type of input.

If you are still struggling to understand a piece of code after trying it like this then you may want to try modifying the code your are reading by adding print statements or other debugging methods to try to understand how the code is being executed.

Ask Questions

So you've read through the code you want to understand, written some simple use cases for it, run it, added print statements, and you still don't know why things are not working? Then ask for help!

Even asking the question to yourself out loud can help. This is sometimes called "rubber ducking" as some programmers find explaining their problems to a rubber duck or some other toy can help understand a problem better.

One important tip before asking for help is to have a very clear idea of what the problem is you want to solve, what you've tried, and why you think it is not working. It is worth trying to step back and write down answers to these questions before you look for help. Sometimes the process of clarifying the problem in this way will make the solution apparent.

Once you have a clear idea of what you are stuck on, there are many people you can ask: fellow students, tutors, lecturers, friends, etc. However, the quickest approach may be to use a search engine with a query that includes aspects of your problem (e.g., "python plotting set own colours"). Don't forget to include the name of the language and/or the name of the library you are using as this can help narrow down the results of the search. We will discuss other strategies for getting help in a later lecture.

3.13.3 Writing Readable Code

This flip-side of understanding how to read code is ensuring that the code you write is easy to understand. While this is particularly important when you are working with other people, it is still good practice even if you think only you will ever look at your code.

"Any code of your own that you haven't looked at for six or more months might as well have been written by someone else."
— Eagleson's law

Documentation and Comments

Code documentation and code comments are two distinct ways of telling a reader about your code. In a nutshell, *documentation* should focus on what your code does at the level of modules, classes, methods, and functions, whereas *comments* explain how or why certain parts of your code are implemented the way they are.

Whenever you are writing code that you intend to look at later, it is good practice to add at least the following documentation to your code:

- **README.txt**: This file should sit at the top level of your repository and briefly explain what your code is for and how to configure and run it (if applicable).

- **File descriptions**: At the top of each of your source code files, add a short description of what the code in the file is for. This is also good place to add your name and possibly license information and contact details. For example, the Python code below shows the start of the NetworkX `graph.py` class (see `https://github.com/networkx/`).

- **Class and function docstrings**: These are the triple-quoted strings that appear after `class` and `def` declarations. These docstrings should describe what the associated class or function does, the arguments and parameters it takes in, and what it returns.

Python Code

```
"""Base class for undirected graphs.

The Graph class allows any hashable object as a node
and can associate key/value attribute pairs with each undirected edge.

Self-loops are allowed but multiple edges are not (see MultiGraph).

For directed graphs see DiGraph and MultiDiGraph.
"""
#    Copyright (C) 2004-2015 by
#    Aric Hagberg <hagberg@lanl.gov>
#    Dan Schult <dschult@colgate.edu>
#    Pieter Swart <swart@lanl.gov>
#    All rights reserved.
#    BSD license.
```

In constrast to documentation which, ideally, is comprehensive and appears after every class and function, comments in code are best used sparingly and concisely. The aim of good code comments is to act as guideposts and warnings about how your implemented your solutions. For example, if setting up a plot requires several lines of code to configure the colours, label the axes, add the data, etc. Then a short comment like `# Setting up a plot of the data` above those lines is appropriate and lets the reader know he or she can skip over that block if it is not considered important. An example of comments as warnings would be to flag a bit of code that seems strange at first glance but is actually correct. The comment `# Add 1 to index to skip first element` would be appropriate just before a loop to let the reader know that the usual indexing from 0 is not used and why.

You should avoid adding comments that just mirror what the code is telling you anyway. For example, if you have some code like `x += 1`, a comment above it like `# Increment x` is completely redundant since it is clear from the code what is happening.

As Steve McConnell puts it: "Good code is its own best documentation. As you're about to add a comment, ask yourself, 'How can I improve the code so that this comment isn't needed?' Improve the code and then document it to make it even clearer."

Coding with Style

Finally, if other people are to read your code, it is just as important to "polish" your code as it is to fix up drafts of essays. As with essays and other forms of writing, whoever reads your code will find it much easier if you use a consistent style. For example, using single-quotes for strings, appropiately spacing and indenting code, and having a convention for naming variables and functions.

Be sure to use meaningful names for your variables, classes, and functions. Good choices of names will make it much easier to understand the intent of your code. For example, calling a variable `username` will make it much easier to guess that the variable is a string than a variable like `un`.

3.14 Lecture 14: Libraries and APIs

Learning Outcomes

- Appreciate that collections of code can be organised into libraries that present an Application Programming Interface (API).
- Understand how Python uses modules and packages to organise collections of code into libraries.
- Know where to find API documentation and how it is typically presented.
- Be able to organise Python code into modules and packages.

Overview

This lecture covers organisation of code into software libraries. In Python this is done through packages and modules. We discuss Application Programming Interfaces (APIs), which defines the interface to a library and abstracts away the implementation.

As you start writing larger and larger programs it becomes important to be well organised in terms of how you structure your code. You will also want to use code from other programmers or distribute code yourself. For these reasons (code organisation and sharing) libraries are very important.

3.14.1 Modules and Packages

In Python libraries are called **modules**. A module contains a set of Python statements and function definitions in a single file. There are hundreds of built-in modules that come with Python (e.g., the `math` and `string` modules, which we have already seen many times). Modules can be grouped together into **packages**. A large number of useful packages come bundled with the Anaconda Python distribution and an even larger number are available over the web through sites like GitHub and BitBucket.

Packages are organised in directories containing a set of files (modules) and the special `__init__.py` file.

Modules

To use the code in a module you need to import it into your code. For example, to use the `math` module include the following command at the top of your Python program

Python Code

```
import math
```

You can now start using functions from that module,

Python Console

```
>>> import math
>>> math.sqrt(2.0)
1.4142135623730951
```

Notice how we had to prepend "`math.`" to the square-root function in the above example. This is because Python uses **namespaces** to avoid problems (i.e., collisions) when multiple modules define the same function or variable.

Prepending the function name (or variable name) with the module name allows Python to determine exactly which function (or variable) you mean.

It is often convenient to rename a module (or more specifically rename the namespace for a module) when importing it to save typing as the following example shows.

Python Console

```
>>> import math as m
>>> m.sqrt(2.0)
1.4142135623730951
```

However, make sure you pick meaningful names and follow conventions to avoid confusion in larger programs.

You can even import functions into the global namespace as in,

Python Console

```
>>> from math import sqrt
>>> sqrt(2.0)
1.4142135623730951
```

but be very careful that you do not re-define existing functions with the same name.

Packages

Packages are collections of modules and/or other packages. Large libraries, such as the plotting library `matplotlib`, do many different things (e.g., 2D plotting, 3D plotting, animation, image manipulation) and so have a large number of related modules. Packages provide a way to organise these modules so they can be shared and used as a single library.

If a module called `mymodule` is part of a package `mypackage` then within Python the module would be referred to by `mypackage.mymodule`. In particular, if you wanted to import and use functions from `mymodule` you would do so like this:

Python Code

```
import mypackage.mymodule
mypackage.mymodule.some_function()
```

Just like with unpackaged modules, you can assign a shorter name to a packaged module when you import it by using the `as` command:

Python Code

```
import mypackage.mymodule as mypm
mypm.some_function()
```

This type of renaming of unweildy names is very common. You will often see statements like `import matplotlib.pyplot as plt` whenever the `pyplot` module from the `matplotlib` package is used.

Python Module Index

_ | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | z

_	
__future__	Future statement definitions
__main__	The environment where the top-level script is run.
_dummy_thread	Drop-in replacement for the _thread module.
_thread	Low-level threading API.
a	
abc	Abstract base classes according to PEP 3119.
aifc	Read and write audio files in AIFF or AIFC format.
argparse	Command-line option and argument parsing library.
array	Space efficient arrays of uniformly typed numeric values.

Figure 3.32: A screenshot of the Python 3 module index.

3.14.2 Application Programming Interfaces (APIs)

As stressed in earlier lectures, one of the most powerful tools a programmer has is abstraction. That is, the ability to hide away aspects of a subproblem that are not relevant to solving a larger problem. For example, when you want to read in a CSV file to get access to some data, you do not care *how* your data file is loaded and parsed. You only care about what format the data will be in after it has been loaded by the library. In this case, the process of reading and parsing a CSV file has been abstracted away and all you really care about is the *interface* to the functions that let you read in the file you want.

Modules and packages, such as the `csv` module, promote abstraction by allowing collections of code that solve a particular problem to be organised and presented as a library. The collection of classes and functions within a library is sometimes called an *Application Programming Interface* or *API* since when you are writing code for a particular application and want to use a library, all you care about is its interface.

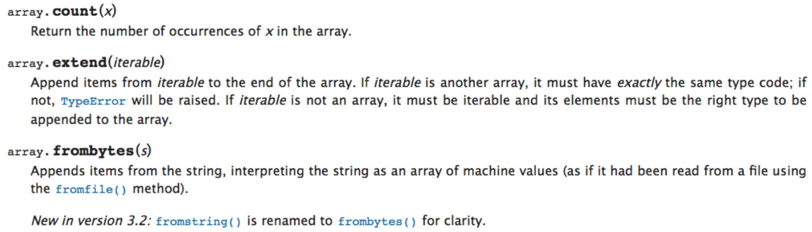
3.14.3 Module and Package Documentation

A crucial part of a software library is its documentation, since this is one of the places where a prospective user will go to understand its API.

There is a fairly common style to Python modules and packages. Both the in-built libraries and third-party libraries typically have web pages that list all the modules, classes, and functions within a package, along with a general overview of what the package is for, and sometime with examples of how it is to be used.

A list of Python 3 modules that come with its standard distribution can be found at this address: `https://docs.python.org/3/py-modindex.html`. Figure 3.32 shows a snippet of this page with a few of the 100 or more modules that come standard. Each of the links from this page goes to the API reference documentation for a specific module.

For example, the page for the `array` module documents a number of functions for efficiently creating and working with large arrays of numbers and char-



array.**count**(*x*)
Return the number of occurrences of *x* in the array.

array.**extend**(*iterable*)
Append items from *iterable* to the end of the array. If *iterable* is another array, it must have *exactly* the same type code; if not, `TypeError` will be raised. If *iterable* is not an array, it must be iterable and its elements must be the right type to be appended to the array.

array.**frombytes**(*s*)
Appends items from the string, interpreting the string as an array of machine values (as if it had been read from a file using the `fromfile()` method).

New in version 3.2: `fromstring()` is renamed to `frombytes()` for clarity.

Figure 3.33: A portion of the API documentation for the array module.

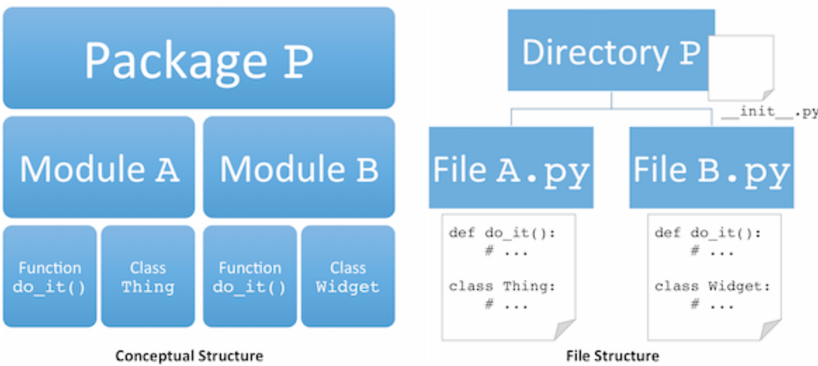


Figure 3.34: Packages are directories and modules are files in Python.

acters in Python. Figure 3.33 shows the documentation for a few of the functions in this module. This style of documentation is typically automatically generated from the docstrings in the code that are associated with each of the classes and functions within a module.

3.14.4 Making Your Own Modules and Packages

Compared to some other languages, Python makes it very easy to organise your code into modules and packages. In a nutshell, modules are just files with Python code in them and packages are directories containing modules or other packages. Figure 3.34 shows how the hierarchical structure of packages and modules is mapped onto the file system.

The only catch is that package directories must always include a file named `__init__.py`. This file can be empty but can also contain Python code for organising, renaming, and presenting collections of modules. We won't go into the details of that configuration here as we will not need it in this course. However, if you are interested, the Python documentation on modules and packages provides a good introduction: `https://docs.python.org/3/tutorial/modules.html`.

The following Example shows how code from a single file can be split into a file containing "application" code — that is, code that is to be run directly — versus library code which is put into a package.

Example 3.14.1. In this example we show how a very simple program can be split into application code and library code that lives in a package.

The starting point is the following Python program which represents a numerical table as a list of lists of numbers and has a few functions for displaying such a table and summarising its rows.

Python Code

```
def format_table(table):
    """
    Convert a table into an aligned, formatted string.
    :param table: A list of list of numbers
    :return: A string representation of the table
    """
    result = ""
    for row in table:
        row = [str(entry) for entry in row]
        result += '\t'.join(row) + '\n'

    return result

def sum_rows(table):
    """
    Sum the rows of the given table.
    :param table: A list of lists of numbers.
    :return: A list with the sums of the table's rows
    """
    if not table: return None

    sums = []
    for row in table:
        sums.append(sum(row))

    return sums

#-------
# Try out the functions on an example table.
table = [
    [0,0,0,0,0],
    [1,2,3,4,5],
    [2,4,6,8,10],
    [3,6,9,12,15]
]

print(format_table(table))
print(sum_rows(table))
```

You could imagine the two functions written above being part of a larger set of functions for working with tables represented in this way. There could be several different functions for formatting tables and some extra functions for summarising them, say by column, or for taking averages, etc.

For the purposes of this example, we will now split this single program file into three parts. The first, `main.py`, will set up the `table` variable and runs the tests at the end of the code above. The other two will be modules: `format.py`, containing the `format_table` function; and `summary.py`, containing the `sum_rows` function. These two modules will live in a package called `tables` which will contain an empty `__init__.py` file.

The `format.py` file will be put in the `tables` directory:

Python Code
```python
# Contents of tables/format.py
def format_table(table):
    """
    Convert a table into an aligned, formatted string.
    :param table: A list of list of numbers
    :return: A string representation of the table
    """
    result = ""
    for row in table:
        row = [str(entry) for entry in row]
        result += '\t'.join(row) + '\n'

    return result
```

The file `summary.py` in the `tables` directory:

Python Code
```python
# Contents of tables/summary.py
def sum_rows(table):
    """
    Sum the rows of the given table.
    :param table: A list of lists of numbers.
    :return: A list with the sums of the table's rows
    """
    if not table: return None

    sums = []
    for row in table:
        sums.append(sum(row))

    return sums
```

Finally, here is the contents of the resulting `main.py` file:

Python Code
```python
# main.py
import tables.format
import tables.summary

# Try out the functions on an example table.
table = [
    [0,0,0,0,0],
    [1,2,3,4,5],
    [2,4,6,8,10],
    [3,6,9,12,15]
]

print(tables.format.format_table(table))
print(tables.summary.sum_rows(table))
```

Notice that the only difference between the above code and the original, single file example is the addition of the `import` statements at the top and the qualified reference of the functions (e.g., `tables.format.format_table` instead of just `format_table`).

3.15 Lecture 15: Searching for Help

Learning Outcomes

- Recognise that help can be found from various places including code comments, inline and API documentation, and technical web forums.
- Appreciate that, while the Internet is a great source of information, the quality of that information varies.
- Develop skills for searching for help effectively.

Overview

There are many places you can turn to for help when developing code. This lecture covers some of the key skills needed in searching for, and interpretting, help from reading API documentation to visiting online technical forums to searching the web.

Software development is a complicate process and efficiently searching for help is a very important skill that you will continue to use even as an experienced programmer. We categorise getting help into four buckets:

Donald Rumsfeld, US Secretary of Defense under President George W. Bush used these categories to excuse the lack of US intelligence regarding the existence of weapons of mass destruction during the 2003 Iraq war.

- **Known Knowns.** You don't need any help.
- **Unknown Knowns.** You've forgotten sonthing and need a reminder.
- **Known Unknowns.** You are aware that a solution exists but know nothing about it.
- **Unknown Unknowns.** You are not even aware that help is out there.

3.15.1 Before You Look for Help

The famous physicist Richard Feynman when asked how he solves problems was quoted as saying, "First, I write down the problem. Then I think very hard. Finally, I write down the solution." When you get stuck on a problem there is a natural temptation to search for help immediately or to try random things in the hope that something will work. Resist these temptations.

Insetad, take 30 seconds to look away from the computer screen and describe to yourself what it is that you are stuck on. The better you can articulate the problem the easier it will be to find a solution. Next write down a plan for what you will try next. The plan does not have to be elaborate—just a few key points will do. This will allow you to approach problem solving in a systematic and structured way. And it just might help you find the solution without looking for external help.

This is sometimes called **rubber duck debugging**. See the associated Wikipedia article for the story.

3.15.2 Inline and API Documentation

Inline documentation can be accessed via the PyCharm IDE and is a excellent quick reference to remind you of the exact name of a function or its calling convention (i.e., argument order). Just start typing and PyCharm will help

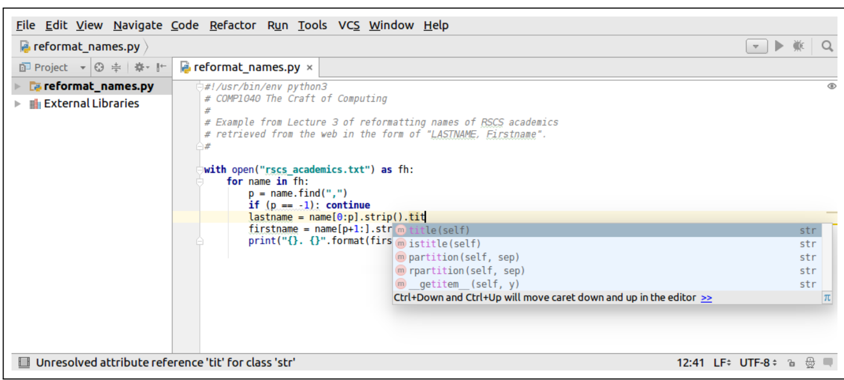


Figure 3.35: PyCharm's inline help provides suggestions as you type.

complete your code by offering suggestions (see Figure 3.35). Hit Enter to accept the current suggestion; just keep typing or hit Esc to cancel inline help.

PyCharm also lets you jump to external API documentation by pressing Shift-F1. This will open a webpage with the official API documentation for the function currently highlighted.

API documentation is a very good source of help and often includes examples. For new libraries try find a `README` file or project home page to get an overview of what the library does. Many libraries come with tutorials or a quick-start section that will get you up to speed on the libraries main features. You should also skim the API reference documentation, which may help jog your memory when you're looking to solve a problem later.

For built-in libraries use the official documentation at `https://docs.python.org/3/`. Be sure to check out the "Quick Search" feature.

3.15.3 Searching for Help

Searching the Web Effectively

You should very familair with using search engines (e.g., Google) to find information on the web. When it comes to programming there are a few tips that will narrow down your search and help you find answers more quickly. They include:

- Add the name of the language you are using (e.g., "Python") to your search. Note, however, that many programming languages have very generic names (e.g., "R") so try appending "language" (e.g., "R language").
- If you're looking for example code to try adding terms "example", "tutorial" or "quickstart" to your search string.
- If you're getting an error message that you do not understand then try searching for the generic part of the error message (i.e., remove line numbers, specific variable names, etc.).

Consider the following buggy code snippet where we attempt to change the second character of a string `my_string`.

Python Code

```python
my_string = 'abc'
my_string[1] = 'Z'
```

An attempt to run the code returns an error message with something like:

```
Traceback (most recent call last):
  File "...", line 2, in <module>
    my_string[1] = 'Z'
TypeError: 'str' object does not support item assignment
```

Searching the Internet for the generic error "`'str' object does not support item assignment`" returns some very useful hits including a *stackoverflow* page, which provides answers to user posted questions (more on this later). In this case the question is related to the same error message and the answers explain that strings in Python are immutable so cannot be changed in-place.

Python Resources

There are many beginner-friendly resources available for Python. These include:

- **Python.org**
  - Python Programming FAQ:
    `https://docs.python.org/3/faq/programming.html`
  - Python Tutorial:
    `https://docs.python.org/3/tutorial/`
  - Asking for help Wiki:
    `https://wiki.python.org/moin/Asking%20for%20Help`
- **CodeAcademy**
  - Similar to CodeBench
    `https://www.codecademy.com/en/tracks/python`
- **Coursera**
  - Introductory online course
    `https://www.coursera.org/course/pythonlearn`

3.15.4 Asking for Help Online

Suppose that you have tried the inline help, reviewed the library API documentation and scanned the FAQ, but you still cannot find an answer to your problem. Fortunately there are a number of places on the web where you can seek expert advice on a problem. These fall into two main catagories—newsgroups and technical Q&A sites.

Discussion Groups

Discussion groups or newsgroups are forums for discussion on a particular topic. Most are accessible via dedicated newsreaders, email subscription, or Google Groups. Email or Google Groups are probably the easiest.

There are a number of newsgroups dedicated to Python programming. You can find a list of active newsgroups here:

`https://www.python.org/community/lists/`

The main ones are:

- comp.lang.python: `https://groups.google.com/forum/#!forum/comp.lang.python`
- Python tutor: `https://code.activestate.com/lists/python-tutor/`

Question & Answer Sites

There are also several technical Q&A sites around the web. The biggest is StackOverflow, which recently passed 10 million questions. Here experts (i.e., other users) will browse the site and answer questions. More helpful questions propagate to the top of the list through a user voting scheme (with reputation management).

You can see all questions on StackOverflow tagged with the "python" keyword here:

`http://stackoverflow.com/tags/python`

However, rather than just browsing questions it is often easier to search for the specific query you are interested in. Put "[python]" in the query string to limit results to those relevant to Python programming. Searching directly from Google will also often return StackOverflow hits.

Not all questions and answers are created equally. Here are some suggestions for improving your mileage when reading answers and asking questions.

Reading Answers

- Check the date of the answer (and language version)
- Answers are ranked by users in terms of helpfulness
- User who asked question can label answer as correct
- Check the "Related" section to see if there are better answers

Asking Questions

- Search the site before posting (duplicates are frowned upon)
- Make your question concrete. Include snippets of code
- Include language version, libraries, what you've tried, etc.

3.15.5 Helping Others

Helping other people is often a good way to help yourself and promotes a community culture that is beneficial to everyone. Often being forced to explain something to someone else can help you to understand it better or from a different perspective. This is exactly why we encourage collaboration in this course, and many discussion groups (including StackOverflow and Python tutor) encourage community participation. Apply the same due diligence and care to answering questions as you would to asking them.

3.16 Lecture 16: Advanced Revision Control

Learning Outcomes

- Understand how revision control systems track the history of a codebase as a series of changes or versions.
- Know how to revert to an earlier revision of a codebase using git.
- Define a merge conflict and understand how conflicts are resolved.
- Observe and example of a branch and merge and understand the difference between a branch and a fork.
- Appreciate the link between issue tracking and revision control.

Overview

In this lecture we discuss more advanced features of revision control systems—mostly `git`—such as branching and merging. These are particularly useful for managing larger projects and for working in groups.

In Lectures 3 and 12 we introduced the idea of revision control and showed how to access Git via the PyCharm IDE.[9] A summary of the different Git commands and places is shown in Figure 3.36. In this lecture we explore more sophisticated uses of revision control systems, most importantly dealing with code conflicts when multiple people are collaborating on the same piece of software.

3.16.1 Revision History and Rollbacks

Software repositories store the entire commit history for a project. That is, every modification (and *commit*) you make to the code is tracked by the repository. Instead of storing a new copy of the code each time you make a commit, the repository records the changes from the previous commit. Each point in the history is called a **revision** and has a unique identifier (or hash), e.g., `68e6124f3e7...` Figure 3.37 illustrates this idea. Note that commits are recorded in the local repository—they only appear in the remote repository when they are *pushed*.[10] The special identifier `HEAD` refers to the current (i.e., latest) point in the revision history.

In PyCharm there are many ways to view the revision history for a single file or for the whole project. The two most accessible ways are via the `VCS|Git|Show History...` menu option or the revision control panel displayed with `View|Tool Windows|Version Control` or pressing Alt-9. You can also view the revision history (assuming that you have *pushed* to the remote repository) via the GitLab web-interface.

Sometimes you will want to look at a previous revision of the code in more detail, or even revert permanently. This is called rolling back the code. In git you do this using the *checkout* command and providing the unique identifier for the

[9] Git can also be accessed from the command line to give you fine control over various operations. However, for now we will stick with using Git in an integrated fashion from PyCharm.

[10] Technically, there can be multiple remote repositories and you can push code to any of there. Moreover, your own local repository can act as a remote repository for another developer on the project.

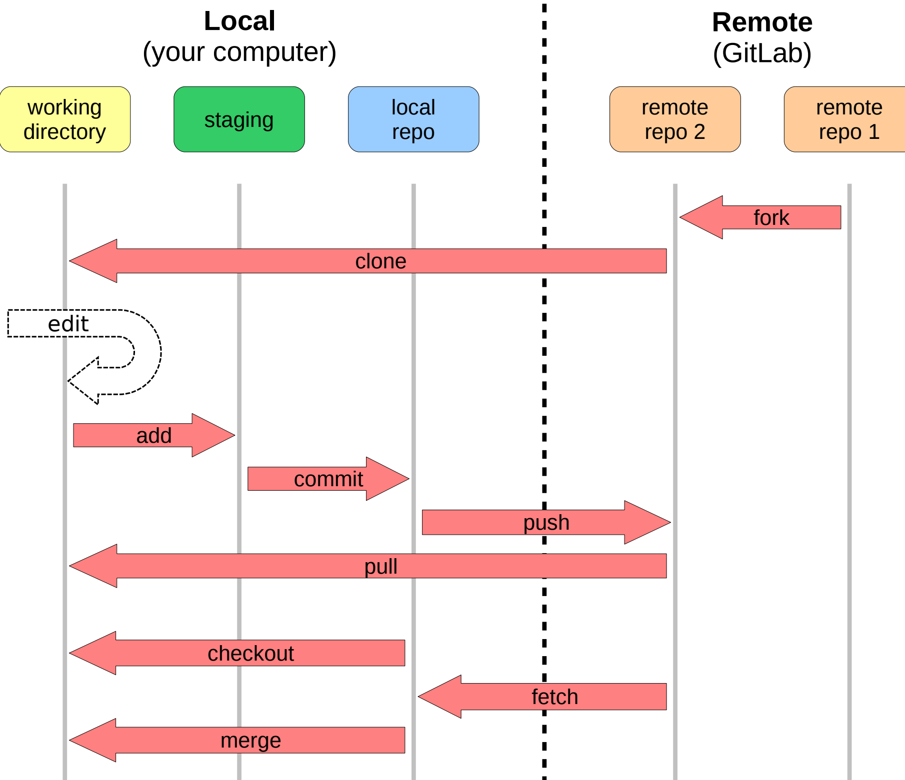


Figure 3.36: Overview of Git places and commands.

Figure 3.37: Illustration of Git revision history.

Figure 3.38: Example **diff** between two snippets of code.

commit that you wish to rollback to. In PyCharm the `VCS|Git|Branches...` menu item provides rollback functionality. Alternatively you can right-click on the revision you want from the Git log and choose `Checkout Revision`. Finally, the menu item `VCS|Git|Revert...` will discard all changes in the working directory and revert your code back to the most recent commit.

On the command line type `git checkout -- <filename>`.

Comparing Revisions

The smallest unit of change in a commit is a line. A list of differences in the lines of two revisions is called a **diff**. When comparing two revisions in PyCharm you are shown whether lines have been *added*, *deleted*, or *modified*. Understanding diffs is key to being able to handle working collaboratively with revision control systems. Figure 3.38 shows a very small example of the difference between two code snippets. The diff visualises where code has been added and modified between the two revisions.

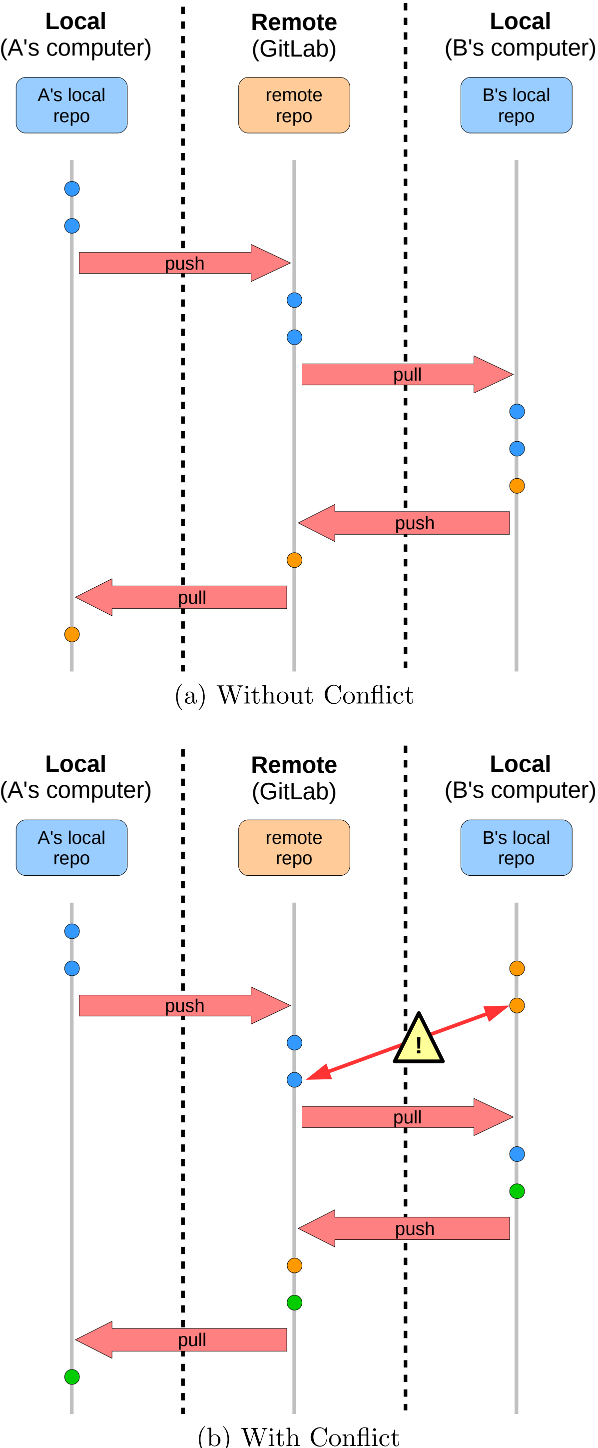
3.16.2 Code Conflicts

When multiple programmers are working on the same project, and editing the same codebase, there will be a need to synchronise the codebase. This is often done via a remote repository that is shared between all project members. Commits from each programmer's local repository get *pushed* to the remote repository. Other programmers can then *pull* those changes to synchronise their local copy of the codebase. In the simple case (shown in Figure 3.39(a)) there is no overlap in the commits and git simply updates to the latest version on each *pull*.

However, when two or more programmers edit the same code concurrently then a **conflict** can occur. An example of this scenario is shown in Figure 3.39(b). In these situations one of two things can happen. First, the commits may be able to be automatically merged (e.g., when commits are for different files of have no changed lines in common). Second, the some manual work is needed to resolve the conflicts. Below we step through an example where manual merging is required.

- Create a project in GitLab (say, `revctrldemo`) by clicking the "New Project" button in GitLab. Add a description and keep it private. Click "Create Project".
- Clone a first local copy of the project. Start PyCharm and choose to "Check out from Version Control". Select "Git". Copy and paste the URL from GitLab and choose the destination directory. Click "Clone" and say "Yes" to opening the cloned project.
- Add a new file to the project (`File|New...`). Call it `main.py`.
  - Click "OK" to add the new file (only) to Git.
- Write some code and commit.
  - Right-click on filename, click `Git|Commit File...`, enter comment and choose "Commit and Push."
- Close the project and clone a copy to a different directory (simulating another developer cloning the project on their local machine).
- Change the code, commit and push, and close the project.
- Open the first copy of the project and make some changes *without* pulling any updates from the remote repository.
- Try commiting and pushing the latest changes. We have a conflict!
- Click "Merge..." Resolve the conflict and push changes with `Git|Push...` (the merge is automatically commited locally by PyCharm). Notice that two commits are pushed. Viewing the "Network" in GitLab shows the different edits and the merge.

The first developer writes and commits initial version of code:



(a) Without Conflict

(b) With Conflict

Figure 3.39: Illustration of shared Git repository with and without code conflicts.

Python Code

```
# declare set of fruit that each child likes
Ariella = {'apples', 'oranges', 'pears'}
Bronte = {'apples', 'bananas', 'oranges'}
Hana = {'apples', 'mangoes', 'oranges'}

# find all fruit liked by all children
for fruit in Ariella:
    if fruit in Bronte and fruit in Hana:
        print(fruit)
```

The second developer clones the repository, modifies the code, and pushes back her changes:

Python Code

```
# declare set of fruit that each child likes
Ariella = {'apples', 'oranges', 'pears'}
Bronte = {'apples', 'bananas', 'oranges'}
Hana = {'apples', 'mangoes', 'oranges'}

# find all fruit liked by all children
liked = set()
for fruit in Ariella:
    if fruit in Bronte and fruit in Hana:
        liked.add(fruit)

# print in alphabetical order
print('\n'.join(sorted(liked)))
```

The first developer then modifies his code without checking whether the remote repository has been updated. He tries to push his changes and finds that there is a conflict. PyCharm provides support for resolving the conflict by displaying a diff between the conflicting files as shown in Figure 3.40. Once resolved the repository ends up with an additional commit which merges the changes.

Python Code

```
# declare set of fruit that each child likes
Ariella = {'apples', 'oranges', 'pears'}
Bronte = {'apples', 'bananas', 'oranges'}
Hana = {'apples', 'mangoes', 'oranges'}

# find all fruit liked by all children
liked = Ariella & Bronte & Hana
for fruit in liked:
    print(fruit)
```

Branches and Merges

Sometimes fixing a bug or implementing a new feature is a considerable project in its own right and cannot be done in a single sitting with a handful of commits. Moreover, development of other aspects of the project can't be paused while work on the bug or new feature is complete. This is where *branches* and *merges* can help in supporting multiple concurrent development streams. Branches are simply different streams of development stored within the one repository. The

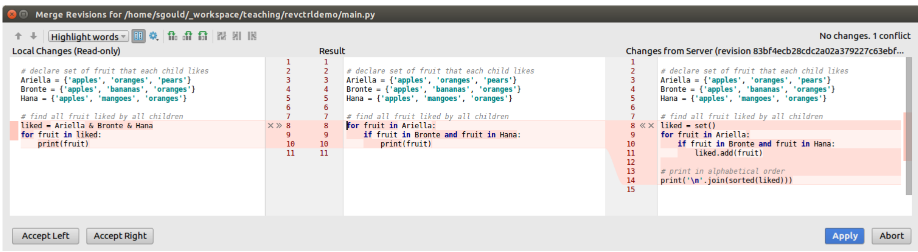


Figure 3.40: Merging in PyCharm after finding a code conflict.

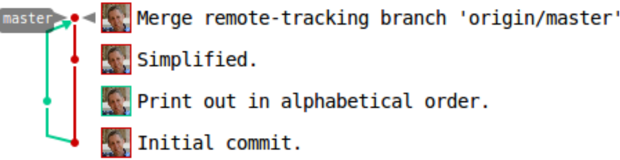


Figure 3.41: Network of repository commits showing merge in GitLab.

main development branch is often called the "master" and this is assumed to be where the latest and most stable version of the code resides.

Creating a new Branch. New branches can be created very easily from an existing branch and will "contain" all the commit history of the branch from which it was created. Usually branches are created off the master branch. In PyCharm select `VCS|Git|Branches...` and choose `New Branch`. You can also create a new branch using the GitLab web interface on the remote repository and then pull it to the local repository (`VCS > Update Project...`). To start working on the new branch you will need to switch to it.

On the command line use `git checkout -b <branchname>`.

Switching between Branches. Sometimes during development you will want to switch to a different branch in your respository. This may be to do some work on another branch or to compare/copy code between two branches. Switching branches is done if Git via a *checkout*. In PyCharm click on menu item `VCS|Git|Branches...` and choose the branch you wish to switch to. Note that if you have uncommitted changes in the branch that you are switching from (i.e., your current branch) then these changes will either be lost (reverted) or stashed (see below). Changing branches updates the files in your working directory.

On the command line use `git checkout <branchname>`.

Merging. At some point you will want to merge the changes you made in your development branch with the master branch. You may also want to merge changes that other developers have been pushing into the master branch with your development branch. This can be done using the same *merge* operation as when merging changed code pulled from a remote repository. The only difference is that you are merging code between two branches in the same local repository. In PyCharm, simply choose `VCS|Git|Merge Changes...`

Merges also come up in the context of merging code between forks. This is quite often used in open-source software development, where the fork can be viewed as a "detatched" branch (which no longer lives in the same repository and is subject to different visibility, etc). Such a merge is sometimes called a *merge request* or *pull request* and needs to be authorised by a member of the

repository where the merge will occur. This type of merge, while important in real-world software development, is beyond the scope of this course.

Stashing. Git and other revision control systems allow a temporary branch to be created to "stash" any changes in the working directory. The command `git stash` will store any uncommited changes. A subsequent command git stash pop will merge changes back into the working directory.

Last word. Sometimes merges and conflicts become too complicated. When this happens you can always checkout the latest version of the code (or an earlier version if you like) in a new directory and start over.

3.16.3 Revision Control and Issue Tracking

One of the key benefits of systems like GitLab is that they integrate revision control and issue tracking. You can make use of this while developing by adding special phrases to your commit messages that get interpretted by the system. For example adding "Fixed #N" or "Closes #N" to a commit message will automatically close Issue #N in the GitLab project when those commits are pushed. Moreover, the issue tracker will identify which commit resulted in the issue being closed.

3.17 Lecture 17: Visualising Data

Learning Outcomes

- Understand the importance of visualising data through plots and graphs (of nodes and edges) for both conveying information and debugging.
- Appreciate different types of plots and when each is appropriate for the information being conveyed.
- Basics of the `matplotlib` and `networkx` Python libraries and how to find help for using these libraries.

Overview

This lecture covers some basic visualisation techniques. We present the `matplotlib` and associated Python library. We also revisit the `networkx` library for drawing graphs.

Perhaps the most common form of data visualisation is through plots such as line graphs, bar graphs, and pie charts.[11] But there are many more way of visualising data (including non-numerical data) and Python has a plethora of packages for this task, some of which we will cover over the next two lectures.

`Tkinter` and `PIL` were once the de facto standard for platform independent graphics in Python, but now there many much more sophisticated libraries.

3.17.1 Basic Plotting with `matplotlib`

Let us start with plotting numerical data. For this we will make heavy use of the `numpy` and `matplotlib.pyplot` libraries, which both come bundled with the Anaconda Python distribution.[12]

Python Code

```
import numpy as np
import matplotlib.pyplot as plt
```

The most basic plot we can draw is a line plot. For example, if we wanted to plot the parabola $y = x^2 - 3x + 2$ over the range $-5 \leq x \leq 5$, we could do so as follows:

Python Code

```
x = np.linspace(-5.0, 5.0, num=21)
y = x ** 2 - 3.0 * x + 2.0
plt.plot(x, y)
plt.show()
```

The resulting plot is shown in Figure 3.42. While the plot of the parabola looks continuous it is actually made up of many small line segments. Examining the code we see that Line 1 defines an array of values for x starting at negative five and going up by increments of 0.5 to positive five. Here the 0.5 comes from the fact that we specified 21 evenly spaced points (inclusive of the end points). A `numpy` array containing the corresponding values of y is computed in Line 2. This line of code is `numpy` shorthand for the following:

[11] Warning: in this lecture we will use the term "graph" to mean both a plot of a function and a set of nodes and edges. The meaning should be clear from the context.

[12] Matplotlib has good documentation and numerous examples at `http://matplotlib.org`.

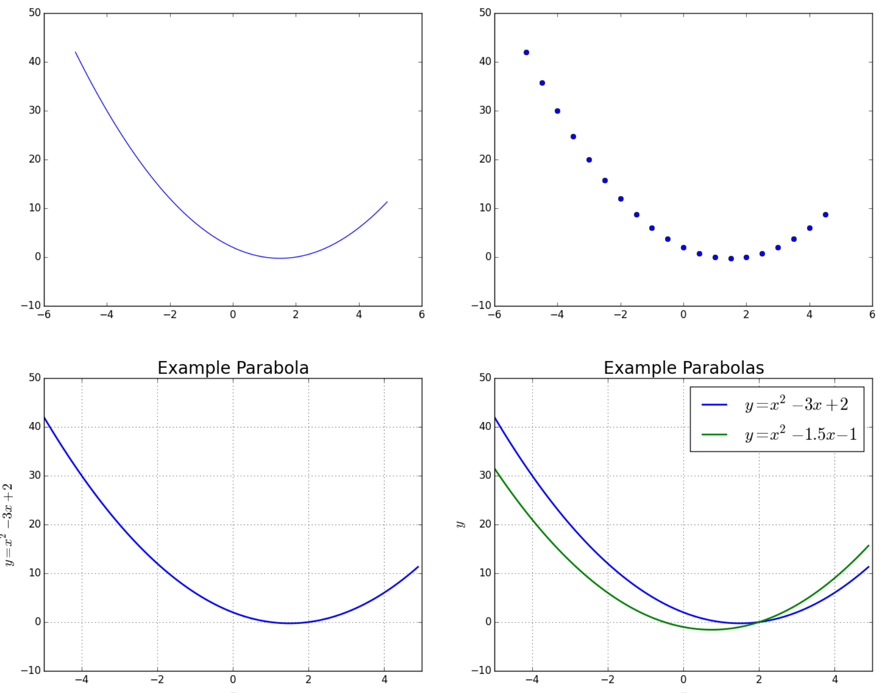


Figure 3.42: Four plots. The first with default formatting, the second showing markers, the third with user-specified formatting, and the fourth showing two plots on the same axes.

Python Code

```
y = np.empty(len(x))  # could also use np.shape(x)
for i in range(len(x)):
    y[i] = x[i] ** 2 - 3.0 * x[i] + 2.0
```

When the graph is plotted `matplotlib` draws straight line segments between consecutive pairs of points.

We can see the sampled points more clearly if we plot *markers* instead of line segments as done by the following code. The corresponding plot is also displayed in Figure 3.42.

Python Code

```
plt.plot(x, y, 'bo')
plt.show()
```

A Quick Word on `numpy`

`numpy` is a very popular package for numerical computing in Python. Quoting the `numpy` website[13], it "is the fundamental package for scientific computing with Python." The main feature of `numpy` that we will be using is its efficient

[13] http://www.numpy.org/

storage and manipulation of multi-dimensional arrays. However, numpy is much more powerful and can be built upon to do a vast range of scientific computing, especially based on linear algebra.

The imporant properties of a multi-dimensional numpy array (apart from the data that it contains) are:

- The number of dimensions (ndim), one for a vector, two for a matrix, etc.
- The size of each dimension (shape). For example an n-by-m matrix, that is, a matrix with n rows and m columns, has shape defined by the tuple (n, m).
- The type of data stored (dtype), which is most often integer or (64-bit) floating-point.

Creating an array in numpy can be done in various ways. Here we demonstrate a few options:

Python Code

```
import numpy as np

# create an array of 21 linearly spaced points between -5.0 and 5.0
x = np.linspace(-5.0, 5.0, 21)

# create an array from -5.0 to (under) 5.0 in steps of 0.5
y = np.arange(-5.0, 5.0, 0.5)

# create an empty array with 20 elements
z = np.empty(20)
```

Once created we can perform basic elementwise arithmetic on arrays, apply functions to each element of an array, reshape an array, index individual elements, and take slices of the array. Consult the numpy tutorial and online documentation for more information on array creation and manipulation.

3.17.2 Plot Formatting and Labeling

The plots we've looked at so far are pretty basic—the axes are not labeled, there is no title, etc. While this is fine for playing around with data and debugging it would not be very useful in a report or on a webpage. The tick button on the Matplotlib plot window (see Figure 3.43) allows you to manually format a graph, but it is often much, much better to do the formatting programmatically. This will allow you to regenerate the exact same graph later without having to go through the trouble of formatting by hand. It will also allow you to generate plots with a consistent look and feel, which gives reports a much more professional appearance.

Matplotlib provides a number of mechanisms for changing the format and style of a graph. Basic functionality is demonstrated in the code below, which changes the linewidth (lw) of the parabola, adds labels to the x- and y-axes, adds a title, and makes the x-axis limits tight. The resulting plot is shown in Figure 3.42.

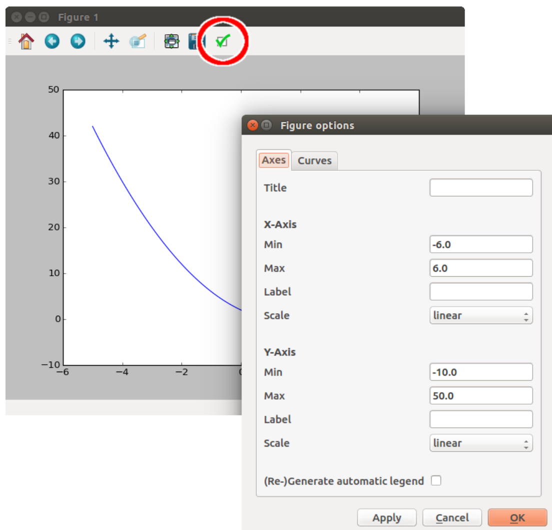


Figure 3.43: Manually formatting a Matplotlib graph.

Python Code

```
plt.plot(x, y, lw=2)
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$y = x^2 - 3x + 2$', fontsize=16)
plt.title('Example Parabola', fontsize=20)
plt.xlim([-5.0, 5.0])
plt.grid()
plt.show()
```

We are not limited to showing a single plot. Multiple plots can be drawn on the same axes. In such cases you will want to add a *legend* to the graph to delineate the plots. The following code provides an example and the resulting plot is, again, shown in Figure 3.42.

Python Code

```
y2 = x ** 2 - 1.5 * x - 1.0
plt.plot(x, y, lw=2)
plt.plot(x, y2, lw=2) # alternatively, plt.plot(x, y, x, y2)
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$y$', fontsize=16)
plt.title('Example Parabolas', fontsize=20)
plt.xlim([-5.0, 5.0])
plt.grid()
plt.legend(['$y = x^2 - 3x + 2$', '$y = x^2 - 1.5x - 1$'])
plt.show()
```

Note how the `legend` function and `xlabel` and `ylabel` functions in the previous example are able to interpret LaTeX-style mathematics. LaTeX is a document typesetting (markup) language that is popular in mathematical disciplines for writing technical articles and reports. It provides a very convenient way of describing mathematical equations, which are enclosed between $ signs. LaTeX is beyond the scope of this course but for those who are interested an excellent introduction can be found at `https://tobi.oetiker.ch/lshort/lshort.pdf`.

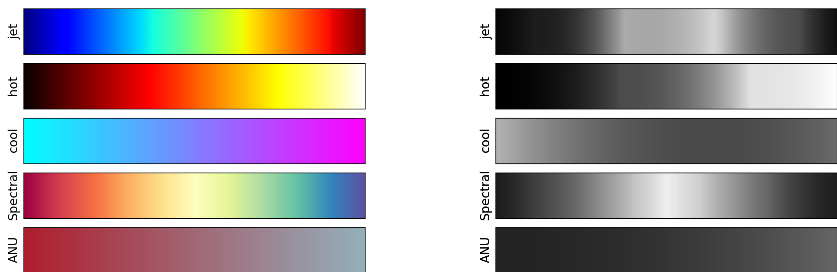


Figure 3.44: Some popular colour maps and a user-defined ANU colour map. Shown are the colour map in full colour and greyscale (as would be displayed if printed in black and white). The "hot" and "cool" colour maps exhibit a better intensity (greyscale) gradation.

Colour Maps

Often when we display multiple plots on a set of axes we will want them drawn in different colours (and, sometimes, styles). This is done automatically for us by `matplotlib`. Moreover, when drawing a 3D surface plot (such as a topographical landscape), contour plot, or heatmap we will want colours to denote different "height" values. The mapping of plot index or surface height to colour is determined by a colour map.

`matplotlib` includes a variety of colour maps that you can select from using the `cm.get_cmap` method. The following code demonstrates the use of this function.

Python Code
```
import numpy as np
import matplotlib

index = np.arange(...)

cmap = matplotlib.cm.get_cmap('Spectral')
cnorm = matplotlib.colors.Normalize(0, len(index) - 1)
colours = cmap(cnorm(index))
```

You can even define your own colour map. Here is some code that creates a "ANU" colour map. Some common colour maps along with the ANU colour map are shown in Figure 3.44.

Python Code
```
import matplotlib

# create ANU colour map
start_colour = "#af1e2d"  # red
end_colour = "#94b0bc"    # grey
cmap = matplotlib.colors.LinearSegmentedColormap.from_list("ANU",
     [start_colour, end_colour])
matplotlib.cm.register_cmap(name="ANU", cmap=cmap)
```

The *jet* colour map is the default. It is used in `matplotlib` and a large

number of other plotting software. However, it has the disturbing property that intensity does not correspond to increasing value as can be seen in the greyscale versions of the colour maps in Figure 3.44. The *hot* and *cool* colour maps are much better in this respect. When choosing a colour map you may want to take this into consideration—for example, what happens if your graph is printed in black and white? See the discussion at `http://matplotlib.org/users/colormaps.html` for more information on choosing colour maps.

3.17.3 Exploring Different Plot Types

There are many different options for visualising numerical data. Make sure you choose a plot type appropriate for the data you want to show.

Line Plot A line plot displays the relationship between two variables as a series of points (on the xy-plane) connected by line segments. Line plots are very useful for showing trends in time series data where the x-axis depicts time and the y-axis represents the quantity being measured/simulated.

Scatter Plot Scatter plots shows paired numerical data using the x-axis (horizontal) and y-axis (vertical). The graph is most useful for exploring the relationship between quantities, for example, age versus height where each point on the plot represents an age-height measurement for a given individual. Often a scatter plot will include a regression line showing the statistical relationship (or correlation) between the variables. They are also useful for discovering clusters in your data. For example, the age-versus-height relationship may differ for males and females.

Bar Graph Bar graphs are used to compare a numerical quantity across a discrete set of categories. Rectangular bars are drawn with length proportional to the quantity being represented. Bars can be drawn horizontally or vertically and multiple bars representing different aspects of a category can be grouped together or stacked one on top of the other. Bar graphs can be made fancy by replacing the bars with pictures, or icons, relating to the items being plotted. This is known as a *picograph*.

Histogram A histogram is a variant of a bar chart used to display a probability distribution of data. Note, however, that a bar chart is used for categorical data whereas a can be used for continous data as well. Here each bar of the histogram represents a range of values, known as *bins*. The height of a bar denotes the frequency (or count) of items appearing in the corresponding bin. A related graph that has become popular for showing word or tag frequencies is the *word cloud*.

For word clouds check out the `wordcloud` Python package by Andreas Mueller.

Pie Chart A pie chart shows the relative proportion of categorical data as slices in a pie. Pie charts use useful for illustrating the qualitative relationship between categories (e.g., country populations or wealth distribution), but are not very good for making precise comparisons between categories. Bar charts are much better in this respect. Sometimes one or more slices of the pie will be offset to highlight the importance of the corresponding category. This is known as an *exploded pie chart*.

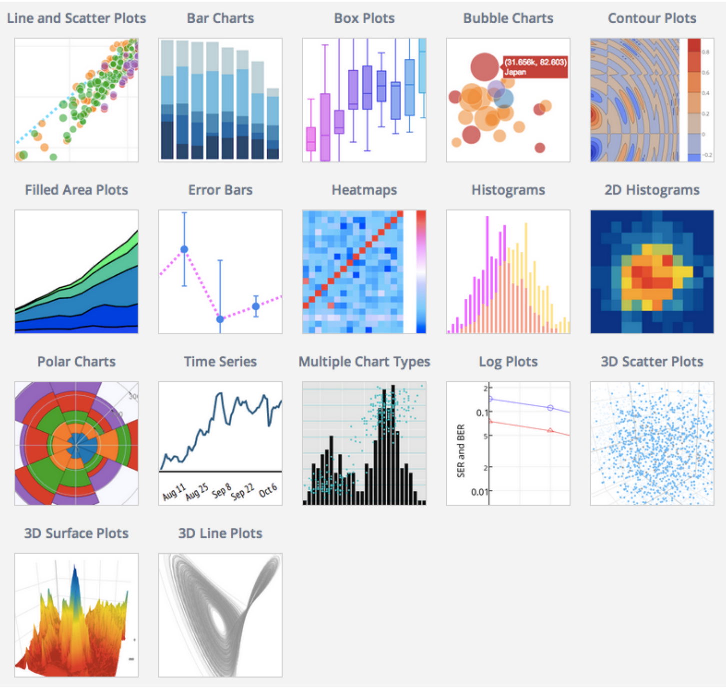
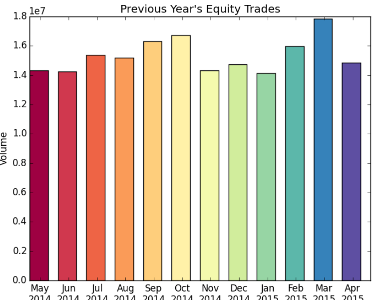


Figure 3.45: Basic plot types in the third-party `plotly` package. Many of the same plot types are available in other Python libraries too.

Heat Map A heat map is a graphical representation of spatial data (or data arranged in a matrix) where a colour scale is used to denote numerical values. For example, weather information such as rainfall or barometric pressure is often displayed as a heat map. In biology, results of assays (e.g., DNA gene expression) are often displayed as heat maps.

Surface Plot A surface plot depicts a function defined over two continuous variables. It is a 3D representation where the x and y axes correspond to the two variables, and the z axis corresponds to the function value. Typically x and y are spatial coordinates and z is some natural quantity measured at each location (e.g., height, temperature, etc.). Variants of the surface plot include a heat map, contour plot, or 3D mesh plot.

Example 3.17.1. Consider plotting monthly stock market trading volume from the ASX. We can extract the monthly volume from the ASX website using the `HTMLParser` class we've seen earlier. The following code will then plot the volume data as a bar chart:



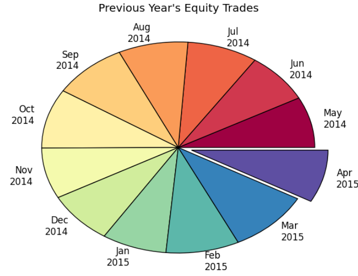


Figure 3.46: Charts showing the trading volume in equity stocks at the ASX from May, 2014 to April, 2015. A bar chart is more appropriate for this type of data than a pie chart, which fails to convey the variation across the months.

Python Code

```
import numpy as np
import matplotlib.pyplot as plt

def plot_bar(volume, months, colours):
    width = 0.65
    index = np.arange(len(volume));

    plt.bar(index, volume, width, color=colours)

    plt.ylabel("Volume")
    plt.title("Previous Year's Equity Trades")
    plt.xticks(index + 0.5 * width, months)
    plt.show()
```

Or as a pie chart:

Python Code

```
import numpy as np
import matplotlib.pyplot as plt

def plot_pie(volume, months, colours):
    explode = [0 for i in range(len(volume))]
    explode[-1] = 0.1
    plt.pie(volume, labels=months, explode=explode, colors=colours)
    plt.title("Previous Year's Equity Trades")
    plt.show()
```

The corresponding bar and pie charts are shown in Figure 3.46. Note that for this type of data the information conveyed by the bar chart is much more useful.

Axis Scale

Sometimes plotting data on a linear scale or starting the scale from zero is not appropriate for the data at hand. The `xlim` and `ylim` methods allow you to change the x- and y-axis limits. You can also control whether the scale is linear or logarithmic with the `Axes.set_xscale` and `Axes.set_yscale` methods as the following code demonstrates.

Python Code
```
plt.plot(x, y + 1.0)
plt.gca().set_yscale('log')     # gca() gets the current axes
plt.xlim([-5.0, 5.0])
plt.grid()
plt.show()
```

3.17.4 Layout and Multiple Subplots

`matplotlib` can draw multiple sets of axes or *subplots* in the same window as demonstrated by the following code. This is useful for visualising related plots that, perhaps, cannot be drawn on the same set of axes.

Python Code
```
plt.subplot(1, 2, 1)
plt.plot(x, y, lw=2)
plt.xlabel('$x$', fontsize=16); plt.ylabel('$y$', fontsize=16)
plt.title('$y = x^2 - 3x + 2$', fontsize=20)
plt.xlim([-5.0, 5.0])
plt.grid()

plt.subplot(1, 2, 2)
plt.plot(x, y2, 'g', lw=2)
plt.xlabel('$x$', fontsize=16)
plt.title('$y = x^2 - 1.5x - 1$', fontsize=20)
plt.xlim([-5.0, 5.0])
plt.grid()

plt.show()
```

You can also draw plots in multiple different windows, known as *figures*. The `figure()` function will generate a new figure. In the above code snippet replace the `subplot` lines with `plt.figure()`. You can also provide a figure identifier (number) to instruct `matplotlib` that the succeeding drawing commands are to be done in that window. The function `gcf()` gets a *handle* to the current figure. Likewise, `gca()` gets a handle to the current axes.

3.17.5 Some Guidelines for Including Plots in Reports

Graphs and charts are an incredibly powerful way of communicating information to your readers. But the value of a graph is deminished if your readers cannot understand how the data is being displayed. Here are a few tips for producing informative graphs that make it easy for your readers to understand what you are showing.

- Use a chart type that is appropriate for the data you are showing.

- Give your chart a title and label all axes. Include a legend if appropriate.
- Make sure all fonts are legible (when printed).
- Use different colours and styles for different plots on the same axes. Make sure the colour scheme works well for printed reports (especially black and white printing—becoming less important these days).
- Be consistent with your colours and styles across figures.
- Give the chart a figure number and caption and refer to it in the text of your report.
- If comparing different methods against a performance metric, indicate the direction of the metric (e.g., "lower error is better" or "higher *ACME* score is better"). You can do this in the caption of the figure.
- Generally, you should not include figures inline. Rather display at the top or bottom of a page, on a page by itself, or in an appendix.
- **Always** write a script to generate and format the plot from raw data. This allows you to easily update the plot when the underlying data changes.
- Don't over do the beautification of your chart, conveying information is more important than a pretty picture.

There are some very nice Python libraries that produce very impressive plots. These are not included in the Anaconda distribution but can be installed on most systems as separate packages. See for example,

- the Seaborn library `http://stanford.edu/~mwaskom/software/seaborn/` for some beautiful statistical plots;
- `plotly` for a wide range of different graph types.

3.17.6 `NetworkX` for Visualising Relationships

The mathematical subject of **graph theory** is concerned with the study of graphs.

Graphs are used in computer science (and mathematics) to represent pairwise relationships between a set of discrete objects. The objects in a graph are denoted by *nodes* (or vertices) and the relationships denoted by *edges* (or arcs) between the nodes. We have already seen some examples throughout this course such as the movie-actor graph in Lecture 1, where we used nodes to represent actors and edges between actors appearing in the same movie together.

The `networkx` library provides functionality for defining and drawing graphs. It also includes some useful graph algorithms such as finding the shortest path between two nodes.

The following code shows a small example of defining and visualising a graph. For more information see the online `networkx` documentation at `https://networkx.github.io/`.

Python Code

```
import networkx as nx
import matplotlib.pyplot as plt

# create a graph
g = nx.Graph()

# add some nodes
names = ["A", "B", "C", "D"]
for n in names:
    g.add_node(n)

# add some edges
g.add_edge("A", "C")
g.add_edge("A", "D")
g.add_edge("B", "C")
g.add_edge("B", "D")

# visualise
nx.draw(g, with_labels=True, font_size=24)
plt.show()
```

3.17.7 Next Lecture

- Data exploration
- Animation
- Saving figures and videos
- Interaction

3.18 Lecture 18: Visualising Data II

Learning Outcomes

- Appreciate more sophisticated visualisations types and animation.
- Be able to use plotting for exploration of numerical datasets.
- Understand the use of callback functions for handling events.
- Know how to save images and videos for incorporation into documents and web pages.

Overview

In this lecture we discuss more advanced visualisation techniques including animation and interaction via event callback functions. We also introduce the `pandas` Python library for manipulating data sets.

3.18.1 Saving Figures and Loading Images

Consider again the nicely formatted parabola plot from the previous lecture,

Python Code

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5.0, 5.0, num=21)
y = x ** 2 - 3.0 * x + 2.0

plt.plot(x, y, lw=2)
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$y = x^2 - 3x + 2$', fontsize=16)
plt.title('Example Parabola', fontsize=20)
plt.xlim([-5.0, 5.0])
plt.grid()
plt.show()
```

We can save the plot from the `matplotlib` window by clicking on the disk icon. Alternatively, if we have a large number of plots to save we can use the `savefig` function,

Python Code

```
plt.savefig("parabola_plot.png")
```

Make sure you put this statement before the `plt.show()` or omit showing the image altogether. The `savefig` function includes a number of optional parameters for controlling how the figure is saved.

You can load and view previous saved figures or other arbitrary images using the `imread` function. We saw an example of this in Lecture 6 on data formats. The following code loads and redisplays the parabola plot.

Python Code

```
import matplotlib.pyplot as plt
img = plt.imread("parabola_plot.png")
plt.imshow(img); plt.show()
```

Note, however, that since we saved the plot as an image we can no longer apply formatting to things like the axes, etc. Python is now treating the plot as a bitmap image and has lost information about how it was constructed.

3.18.2 Data Exploration and the pandas Library

The pandas library[14] is an easy-to-use data analysis library that very nicely handles reading and indexing of tabular data. The following code snippet shows how to load a CSV file using the pandas library. Here the *wine quality* data file was downloaded from the UCI Machine Learning repository.[15]

Python Code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# load the wine quality dataset
wine = pd.read_csv('winequality-red.csv', sep=';')

# print out the different wine attributes (csv header line)
for attribute in wine:
    print(attribute)
```

The code produces the following output:

```
fixed acidity
volatile acidity
citric acid
residual sugar
chlorides
free sulfur dioxide
total sulfur dioxide
density
pH
sulphates
alcohol
quality
```

We could have also printed the entire dataset with:

Python Code

```
print(wine)
```

Access to individual columns, i.e., slicing, for plotting or analysis can be done using attribute names. For example, we can compute the average pH level across the entire dataset with

Python Code

```
# print average 'pH' across the entire dataset
print("mean pH is {}".format(np.mean(wine['pH'])))
```

[14] http://pandas.pydata.org
[15] https://archive.ics.uci.edu/ml/datasets/Wine+Quality

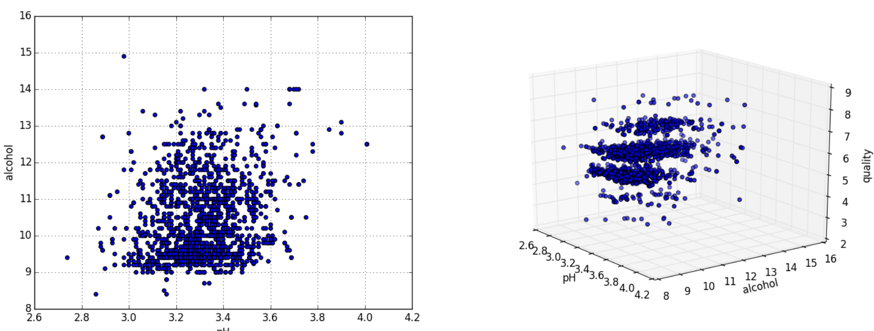


Figure 3.47: 2D and 3D scatter plots from `pandas` data.

where we have used `wine['pH']` to extract the pH column from the dataset and then applied the `mean` function from `numpy` to compute the average.

The `pandas` library also integrates well with `matplotlib` for plotting data. For example, the following code will produce a 2D scatter plot of *pH* versus *alcohol* for all points in the dataset.

Python Code
```
# plot pH against alcohol content
wine.plot(kind='scatter', x='pH', y='alcohol')
plt.show()
```

Likewise, we can produce 3D scatter plots.

Python Code
```
# plot pH and alcohol content against quality
from mpl_toolkits.mplot3d import Axes3D

ax = plt.figure().gca(projection='3d')
ax.scatter(wine['pH'], wine['alcohol'], wine['quality'])
ax.set_xlabel('pH')
ax.set_ylabel('alcohol')
ax.set_zlabel('quality')
plt.show()
```

The resulting scatter plots are shown in Figure 3.47.

Visualising more than three dimensions of data is difficult. One convenient diagram is the *scatter matrix*, which is a square array of plots, each panel in the grid plotting one attribute against another. The diagonal of the array of plots shows a histogram distribution for the attribute corresponding to that row (and column). Scatter matrices can be generated in a single line of code using `pandas`:

Python Code
```
# multiple plots
pd.tools.plotting.scatter_matrix(wine)
plt.show()
```

The corresponding plots are shown in Figure 3.48.

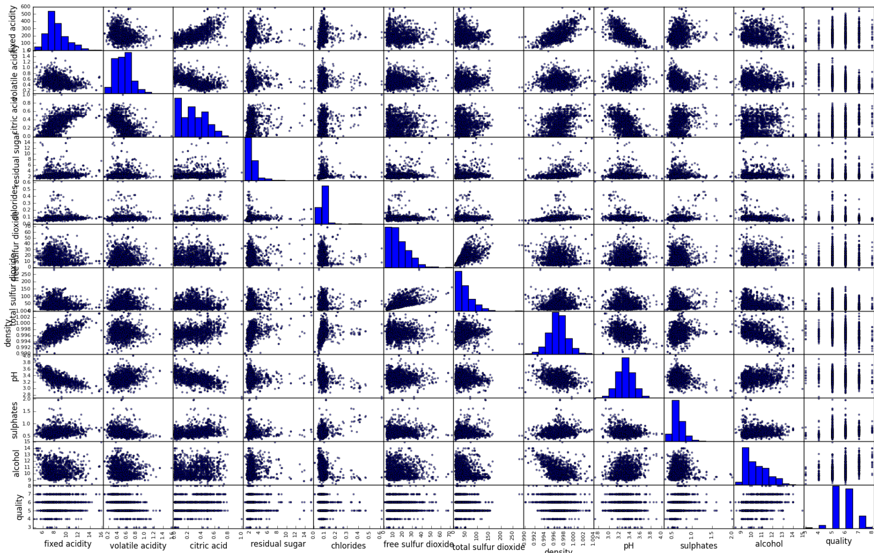


Figure 3.48: Scatter matrix for the wine dataset.

3.18.3 Animation

An animation is the process of creating perceived motion by rapidly displaying a sequence of (static) images, called *frames*. Typically images need to be shown between at 24 to 60 frames per second for motion to be perceived.

Simple animations can be generated using `matplotlib` by repeatedly redrawing on a figure window. This model for animation is supported by two programming mechanisms: (i) `plt.draw()`, which instructs `matplotlib` to immediately redraw the current figure window, and (ii) `plt.pause()`, which pauses execution for a given number of seconds thus controlling the frame rate (and giving the operating system enough time to refresh the window). Example animation code is shown below.

Python Code

```python
import math
import numpy as np
import matplotlib.pyplot as plt

plt.figure()                # intialize the figure
plt.ion()                   # turn on interactive mode

for theta in np.linspace(0.0, 2.0 * math.pi, 100):
    x = np.linspace(0.0, 4.0 * math.pi, 100)
    y = np.sin(x + theta) # compute the shifted sine wave
    plt.cla()             # clear the plot on the current axes
    plt.plot(x, y, 'b-')  # plot as a line graph
    plt.xlim(x[0], x[-1]) # set the range of the x-axis
    plt.title("$\\theta = {:0.3f}$".format(theta))
    plt.draw()            # tell Python to update the plot

    plt.pause(0.1)        # delay for 100ms before the next plot
```

An alternative animation model is via *callbacks*, which have the advantage

of separating the drawing logic from the frame rate control. Callbacks are an example of an event-driven programming model. In this instance, an *event handler*, i.e., the callback function, is registered with `matplotlib` to implement the animation updating. When the animation starts `matplotlib` sets a timer to go off at a regular interval. Each time the timer expires an event is triggered and `matplotlib` invokes the callback function.

`matplotlib` provides a very simple method of registering a callback function for animations using the `animation.FuncAnimation()` method. An example is shown in the code below.

Python Code

```python
import math
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

def animate(fnum):
    """Animation function for plotting a phase-shifted sine wave."""
    theta = 2.0 * math.pi * float(fnum) / 100.0
    x = np.linspace(0.0, 4.0 * math.pi, 100)
    y = np.sin(x + theta) # compute the shifted sine wave

    plt.cla()              # clear the plot on the current axes
    plt.plot(x, y, 'b-')  # plot as a line graph
    plt.xlim(x[0], x[-1]) # set the range of the x-axis
    plt.title("$\\theta = {:0.3f}$".format(theta))

    return p.gca()        # update the entire axes

# main function
f = plt.figure()           # intialize the figure
plt.ioff()                 # turn off interactive mode

# initialize the callback function
ani = animation.FuncAnimation(f, animate, interval=100,
    frames=100, repeat=False)
plt.show()
```

Passing Arguments into the Animation Function

Often a callback function will require more state information that just the frame number (passed as the first argument, `fnum`, to the animation function). It is possible to store state in global variables, which the callback will be able to access. However, a cleaner alternative is to pass state as an input to the callback explicitly. The following code shows how we can do this using the `fargs` optional parameter when registering the callback.

Python Code

```
import math
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

def animate(fnum, x, theta):
    """Animation function for plotting a phase-shifted sine wave."""
    y = np.sin(x + theta[0])     # compute the shifted sine wave
    theta[0] += 0.02 * math.pi  # update state

    plt.cla()                  # clear the plot on the current axes
    plt.plot(x, y, 'b-')       # plot as a line graph
    plt.xlim(x[0], x[-1])      # set the range of the x-axis
    plt.title("$\\theta = {:0.3f}$".format(theta[0]))

    return plt.gca()           # update the entire axes

# main function
f = plt.figure()               # intialize the figure
plt.ioff()                     # turn off interactive mode

# initialise state information
x = np.linspace(0.0, 4.0 * math.pi, 100)
theta = [0.0]                  # note: list

# register the callback function and show
ani = animation.FuncAnimation(f, animate, fargs=(x, theta),
    interval=100, frames=100, repeat=False)
plt.show()
```

In this example we pass two pieces of state information. The first is the variable `x`, which defines the domain for plotting. This variable does not change throughout the animation so having it pre-computed and passed in to `animate` saves some processing time. The second piece of state information is `theta`, the phase of the sine wave, which changes each iteration. Because we need to update `theta` within the callback we have implemented it as a list—if it were simply a float we could not modify it since floats are immutable (see Lecture 4).

An alternative to passing in lots of different arguments is to group the animation state information together into a class and only pass in an instance of the class. This has the added advantage of separating functionality between the class and the drawing function as shown in the code snippet below.

Python Code

```
class SineWaveAnimationState:
    """Holds state information for animating a sine wave."""
    def __init__(self):
        self.x = np.linspace(0.0, 4.0 * math.pi, 100)
        self.theta = 0.0

    def y(self):
        """Returns the shifted sine wave."""
        return np.sin(self.x + self.theta)

    def inc_theta(self, amount=0.02):
        """Increment theta."""
        self.theta += amount
```

The animation function can then be simplified to:

Python Code
```
def animate(fnum, state):
    """Animation function for plotting a phase-shifted sine wave."""
    plt.cla()                  # clear the plot on the current axes
    plt.plot(state.x, state.y(), 'b-')     # plot as a line graph
    plt.xlim(state.x[0], state.x[-1])      # set the x-axis limits
    plt.title("$\\theta = {:0.3f}$".format(state.theta))
    state.inc_theta()          # increment theta

    return plt.gca()           # update the entire axes
```

And the main code to start the animation:

Python Code
```
# main function
f = plt.figure()               # intialize the figure
plt.ioff()                     # turn off interactive mode

# initialise state information
state = SineWaveAnimationState()

# register the callback function and show
ani = animation.FuncAnimation(f, animate, fargs=(state,),
    interval=100, frames=100, repeat=False)
plt.show()
```

When specifying the `fargs` parameter don't forget that you must provide a tuple. A 1-tuple (tuple with only one element) is Python is declared with a comma following the first element as can be seen in the code above.

Creating Videos

A sequence of frames from an animation can be turned into a video and saved by assigning the `FuncAnimation` object to a variable and then invoking the `save` member function before calling `plt.show()` as illustrated in the following code:

Python Code
```
ani = animation.FuncAnimation(f, ...

ani.save("animation.mp4", writer="avconv", fps=30,
    extra_args=["-vcodec", "libx264"])
```

Note that the parameters may need to be adjusted depending on the operating system and set of video codecs installed. Moreover, the approach does not work out-of-the-box on many systems ann can be difficult to get working. A fail-safe (albeit less elegant) approach is to output each frame (using `savefig`) and combine them together into a video with another tool.

3.18.4 Interaction

The most simple form of user interaction involves waiting for the user press a key or mouse button. We have also already seen how to receive user input from the keyboard with the `input` function.

If you wish to show a sequence of plots/images and wait for the user to press a key between each one, then the easiest approach is to use `matplotlib`'s `waitforbuttonpress()` function. This function blocks execution of the program until the user presses a keyboard key or clicks a mouse button, returning `True` for the former and `False` for the latter.

More interesting things can be done if we know where the mouse cursor was when the mouse was clicked. For this we can use the `ginput()` function, which blocks until a given number of mouse clicks have been received. The function returns a list with the location of each mouse click. The following code snippet waits for five mouse clicks and then displays them on the plot.

Python Code

```
plt.figure()
plt.cla()
points = plt.ginput(5)
for i, (x, y) in enumerate(points):
    plt.plot(x, y, 'bo')
    plt.gca().text(x, y, "Point {}".format(i + 1))
plt.draw()
plt.show()
```

The interactivity covered in this lecture is quite rudimentary but sufficient for many needs. More advanced user experience (UX) design and graphical user interface (GUI) development, which is often built on an event-driven programming model with support for handling mouse movement, button clicks, etc., is beyond the scope of this course. Interested students can read about event handling in `matplotlib` at

`http://matplotlib.org/users/event_handling.html`

3.19 Lecture 19: Debugging Strategies

Learning Outcomes

- Recognise that software testing and debugging is an important part of the software development process (even for small programs).
- Be able to identify and fix common bugs that can occur in code.
- Understand the basic features of a debugger and how it can be used to track down bugs.
- Be aware of techniques for minimising the number of bugs in your code.

Overview

You've probably already discovered that almost all programs have bugs. Being able to find and resolve bugs is a key skill that separates really great programmers from everyone else. This lecture gives you some high level strategies for debugging code.

The term **bug** is attributed to Grace Hopper, a naval computer engineer, who used the term after finding that a moth had blown part of the computer circuitry she had been working on.

A disciplined approach to programming and good software design will help reduce errors in code. But no matter how careful or experienced you are bugs will still occur in your code. Edgar Dijkstra once (half) joked: *If debugging is the process of removing software bugs, then programming must be the process of putting them in.* Indeed, some studies estimate an industry "defect rate" of anywhere between 1 and 25 errors per 1,000 lines of code. These rates may sound small, but when you take into account that the codebase for the Windows operating system is around 50 million lines of code and Google's services are reported to consist of 2 billion lines of code, these rates suggest that there are 50,000 bugs in Windows and over 2 million in Google's services!

cf. *Code Complete: A Practical Handbook of Software Construction (2nd Edition)*, S. McConnell, Microsoft Press, 2004.

Some bugs can be found through ad hoc testing (i.e., just running your code), but a set of well designed tests is essential if you really want to have any confidence that your code is working correctly. Professional programmers often spend more time writing tests and fixing bugs than they do writing the intended software application itself.

The aim of this lecture is to highlight some techniques that can help you find, fix, and prevent bugs in your code.

3.19.1 The Craft of Debugging

The most difficult part of debugging is locating the source of the bug. Once found, fixing the bug is often easy. This lecture gives you some strategies for finding bugs. The important thing to remember is that debugging is an ongoing process—tedious at times—and an integral part of software development. Sometimes you will think you've fixed a bug only to find its ugly head sometime later. Software revision control and issue (bug) tracking tools can greatly help manage this process.

Debugging typically has several stages that we will expand on through this lecture:

- **Detection**: Determining there is a bug. This can be obvious (e.g., when error messages or nonsense output appears) or more subtle and tricky (e.g., a non-repeatable problem or plausible but incorrect output).
- **Isolation**: Finding roughly where in your code the bug is. This can involve extracting just the problem area from your existing code and running it with a simplified version of the input that manifested the bug.
- **Comprehension**: Understanding precisely why a certain piece of code is causing a problem. This requires a strong sense of what the intended behaviour of your code should be.
- **Correction**: Modifying your code to remove the bug. It is important to make the fix as small and localised as possible to avoid introducing new bugs.
- **Prevention**: Writing your code in such a way as to minimise bugs. This can involve writing test cases, programming defensively, and having other people check through your code.

3.19.2 Detecting and Isolating Bugs

Isolating Bugs

Like several problems in programming, getting rid of bugs is susceptible to the "divide and conquer" strategy. If there are many errors in your code it is much more effective to isolate each one and deal with them in turn.

A high-level plan for isolating bugs is the following:

1. **Work backwards**: Use error messages or incorrect output as starting points to figure out what has gone wrong. Look at the code at or just before these places to see whether there is an obvious bug.
2. **Simplify**: If it is not clear why your code is not working, you may want to simplify the code or the input to the code in such a way that the bug is still present. Consider opening up a new file with only the parts of the code you suspect your bug is in.
3. **Mentally execute**: Mentally walk through the suspect code explaining to yourself what each line should be doing. Add comments to help you, if necessary.
4. **Use Logging**: Add print statements or other forms of output to inspect the value of variables or expressions. Check these against what you expected to see in the previous step.

Error Messages

Finding the place to work backwards from is sometimes very easy, especially when the bug results in an error message. For example, consider the following piece of code in a file called `problem.py`:

Python Code

```
1 print('Uh oh!)
```

When run, this results in the error:

```
  File "problem.py", line 1
    print('Uh oh!)
                 ^
SyntaxError: EOL while scanning string literal
```

The first line here tells us which file the problem is in and which line the problem is on. The code from that line and file is then shown with a caret symbol ('^') highlighting where on the line the problem is. The final line says that the problem is a syntax error (i.e., the code was not well formed) and that it reached the end of the line (EOL) before it found the end of the string.

Be aware, however, that sometimes the error message will not point you to exactly where the problem is. Consider the following code:

Python Code
```python
print('Line with error'
x = 1
```

The error message received when this is run says the problem is on line 2 even though it is clear that there is a missing parenthesis on line 1:

```
  File "problem2.py", line 2
    x = 1
    ^
SyntaxError: invalid syntax
```

Error messages are therefore only a starting point when debugging your code. If the problem shown in the error message does have an obvious fix then you may have to hunt backward from where the error is reported to find the bug.

Using Version Control to Help Find Bugs

Sometimes you may have a working codebase that, due to changes made by yourself or others, suddenly stops working problem. In this case, it can be very helpful to look through the recent commit history in the version control system you are using. If the commits that were made after the bug appeared only change a small amount of code, these changes can be a great place to start looking for the new bug.

Also keep in mind that you can "go back in time" using the revision control system to find a point where the bug did not occur. We covered how to do this in Lecture 18.

3.19.3 Using Debuggers

Sometimes the interactions between various parts of a codebase can become so complex that mentally trying to determine what is being executed becomes very difficult. Fortunately, most languages and software development environments provide *debuggers* that allow a programmer to stop a piece of code executing, examine the state of variables, and step through the execution incrementally.

Figure 3.49 shows a snapshot of a debugging session in PyCharm. The code in the top panel has been run via the `Run|Debug...` command and has stopped

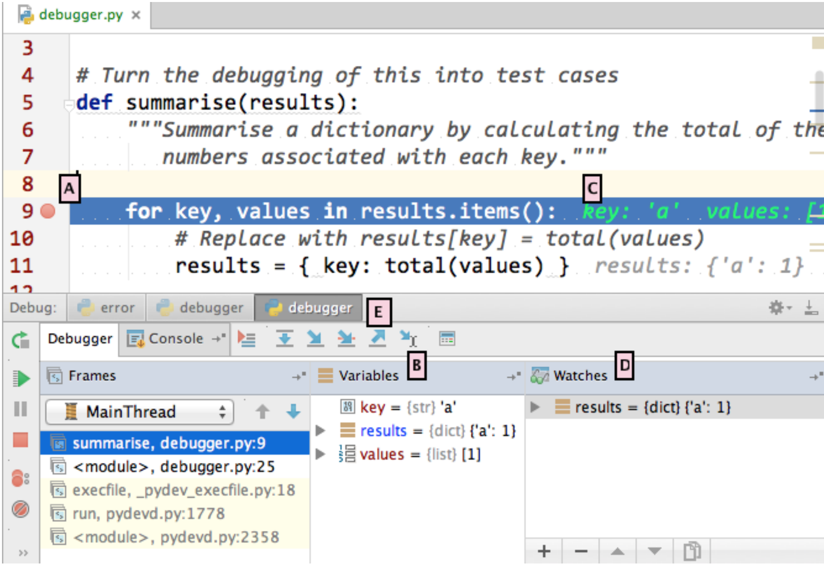


Figure 3.49: A screen capture of PyCharm's debugger.

its execution at the highlighted line (line 9). The red circle at position A denotes a *breakpoint* in the code. The Python interpreter to pause its execution whenever it reaches a line with a breakpoint. You can set a breakpoint by clicking just right of a line number. Clicking on a breakpoint circle will remove it.

When code execution is paused at a breakpoint the debugger pane (shown at the bottom) will appear with several piece of information about the state of the code. In the "Variables" pane at position B you can see that several variables in the scope inside the function `summarise` are shown, along with their current values (e.g., `key` is a string set to `'a'`). The values of variables on the current line are also shown in green at position C. The "Watches" panel in position D lets you track the value of variables or expressions that may be outside the current scope. These are added by using the '+' symbol at the bottom of the panel. You could, for example, add a watch that calculates the length of the results dictionary by pressing '+' and typing in `len(results)`. Each time results is changes this expression will be updated to show its current value.

The icons showing various arrows at position E provide several types of control over the way in which the code is executed after it has hit a breakpoint (these are also available from the `Run` menu and hotkeys). They are described in Table 3.8.

Get to know these debugging commands by trying them out on your own code. The ability to slowly step through your code and inspect the values of variables and evaluate expressions is an extremely powerful way to track down problems.

3.19.4 Know Your Enemy

By definition, a bug in your code is a difference between what you expected to happen and what actually happened when you ran your code. This means that in order to find a bug you need to have a precise idea of what it is you expect

Icon	Name	Description
	Show	Show the current execution point.
	Step	Steps one line forward in the current file.
	Step Into	Step into the next function or method to be called. This may open a new file and possibly library code.
	Step Into My Code	Step into the next function or method to be called but do not show execution through library code.
	Step Out	Execute until the end of the current function and stop at first line after it was called.
	Run To Cursor	Execute until the line containing the cursor in the edit window.

Table 3.8: Debugging commands and their PyCharm icons.

your code to do. Before you start trying to track down a bug it is worthwhile stepping back and thinking about—or even better, writing down—exactly what you want your code to do. Code comments or function docstrings are very good places to do this.

Often, simply clarifying exactly what it is you want your code to do will make it obvious where the problem lies. If that's not the case, there a several common "gotchas" that cause bugs. This are discussed with examples below and are sometimes good things to check for when trying to track down a bug.

Common Bugs

In this section we will briefly look at a number of common bugs that cause problems for new (and sometimes even experienced) programmers.

Off-by-one and indexing errors Most languages, including Python, index lists from zero instead of one. If you haven't programmed much before this can often catch you out, especially when having to loop through a list-like data structure.

Python Code

```python
# Print the first and last element of a list
xs = [1,2,3]
print(xs[1])     # BUG: Prints 2.
print(xs[3])     # => IndexError: list index out of range
```

The correct way to get the first and last element of a list `xs` is to use `xs[0]` to get the first element and `xs[len(xs) - 1]` or just `xs[-1]` to get the final element. More generally, if you want to access the nth element in a list you need to use the index $n - 1$, not n.

Infinite Loops The following piece of code is attempting to use a `while` loop to print the numbers from 0 to 4 inclusive:

Python Code
```python
i = 0
while i < 5:
    print(i)
    i + 1                   # BUG: should be i = i + 1
```

The intention here is to increment the value of `i` at the end of each loop but the expression `i + 1` does not update the value of `i`.

One reason bugs like this occur is because the programmer must keep track of the variable `i` "by hand". Whenever possible, you should try to re-express loops like these so that this sort of updating is handled automatically. The following piece of code does exactly the same as what the previous example attempted but notice that the `for` loop makes sure `i` is incremented each loop.

Python Code
```python
for i in range(5):
    print(i)
```

Unintended side-effects A very common cause of bugs is unintentionally modifying the state of some object or data structure. This can happen in a number of ways. A very simple instance is shown below where two variables `xs` and `ys` are referencing the same list `[1,2,3]`. Because both variables point to the same list, any modification of list via one of the variables will also affect what is returned when the other variable is accessed.

Python Code
```python
xs = [1,2,3]
ys = xs
ys[0] = 100     # NOTE: xs[0] == 100 too!
```

If you expected `ys` to be a new, independent *copy* of the list that remains unchanged then you will likely have a bug in your code. To make a new copy of a list and ensure that it will not be modified by changes to the original, you must use the `copy()` method on a list like so:

Python Code
```python
xs = [1,2,3]
ys = xs.copy()
ys[0] = 100     # NOTE: xs[0] is still 1
```

Unintended side-effects can be caused is many other ways. Another common instance is when functions modify the arguments they are given as input. For example, a function might make a change to an element of a list without first making a copy of it, as in the previous example.

It is important to carefully read the documentation of any function you use to make sure the arguments you pass in do not change. When you are writing functions, it is generally good practice to avoid modifying its arguments inside the function. If you do need to modify an input argument, make sure this is very clearly stated in the documentation for your function or method.

Machine Precision A problem common to almost all program languages is *machine precision* when it comes to representing and working with numbers, and Python is no exception.

Floating point numbers in Python use 53 bits to represent numbers and although this is more than enough for most circumstances it can result in unexpected behaviour, especially when working with very large and very small numbers simultaneously.

However, the following code shows that small representational errors can occur even when dealing with simple arithmetic.

Python Console
```
>>> 0.1 + 0.2
0.30000000000000004
```

The reason that the answer is not exactly 0.3 is because the numbers 0.1 and 0.2 are not exactly representable in binary. This is similar to why 1/3 is 0.33333... in decimal. Because only 53 bits are available, approximations of 0.1 and 0.2 are used and when these are added this approximations means their sum is slightly bigger than 0.3.

These sorts of approximation means you have to be very careful when testing whether two numbers are equal in Python. A test like `0.1 + 0.2 == 0.3` would return `False` due to the approximation error. Instead of checking whether two values are exactly equal, one often checks if they are "close" using a function like the following:

Python Code
```
def are_close(x, y, tolerance=10e-7):
    """Test whether the absolute difference between the two
    numbers are within a specific tolerance of each other."""
    return abs(x-y) < tolerance

are_close(0.1 + 0.2, 0.3)   # Returns True
```

Notice that in the above example, 0.1 + 0.2 are considered "close enough" to 0.3. The `tolerance` parameter specifies that "close enough" is equal up to 7 decimal places. The `unittest` module, which we saw earlier, provides a method called `assertAlmostEqual` that provides essentially this functionality for unit testing.

3.19.5 Fixing and Preventing Bugs

Once you have found the cause of your bug and understand why it happened, you are in a good position to fix it. Once a bug is vanquished it is a good idea to take a step back to see whether similar bugs might be avoided in the future.

Fixing Bugs

There is often a strong temptation to dive in, make the first change to the code that comes to mind, then run it and see whether the problem is fixed. However, this is not advisable. It is better to think through a few possible changes you could make and consider their merits. For example, you may have found an off-by-one error in a loop and think to yourself, "I'll just subtract one from

this index variable and that will fix it". An alternative might be to rewrite the loop using `for ... in ...` so the index variable is no longer needed. Another alternative might be to write a separate function that takes the index variable and the list it refers to as arguments and returns the appropriate value. You may even want to encapsulate this by writing a new class. The best solution will, of course, depend on the code and problem and your experience but choosing the right fix can save you time and improve the overall quality of your code.

Sometimes a bug fix will suggest a large-scale change to the design of your code so as to avoid similar bugs in the future. Changing the structure of your code without significantly changing its behaviour is called *refactoring* and is covered in detail in Lecture 3.22.

3.19.6 Preventing Bugs

No Code is Good Code!

There are a number of things you can do to minimise the chance of having errors in your programs but the simplest (recalling the defect rates from the introduction) is *write less code*! Provided you are not using "clever" tricks to cram lots of functionality into each line, having less code means fewer "moving parts" that you have to reason about in your program which means you will be less likely to make a mistake.

There are two very easy ways to reduce the amount of code you use to solve a problem:

1. **Use libraries or existing code whenever appropriate**. Code that has been used by yourself or others in the past is likely to have been tested and debugged already. Trying to reinvent a solution that already exists is typically a bad idea.

   The relative ease in which code can be copied, shared, and reused is one of the major reasons software has developed incredibly rapidly over the last several decades.

2. **Don't Repeat Yourself**. If you find very similar code in two or more places in your codebase try to extract the common behaviour into functions or classes. The big advantage to doing this is that it reduces the number of possible places an error may occur. Also, if you do find a problem you only need to fix it in one place, not several.

Finally, whenever possible, delete code that is no longer in use. If you have been using version control you can alway get it back later if it is useful. However, having it lying around when it is not being used just makes your program harder to understand and make cause inadvertent problems, such as variable name clashes.

Write Test Cases

If you are writing a function or class that implements some non-trivial data processing or simulation, consider writing unit tests before you write the code to solve the problem. This has two big advantages. First, writing the tests will force you to think clearly about the intended behaviour of your functions.

Second, when you do write the code you can immediately test whether it is working as intended, thereby finding bugs earlier than if you did not have tests.

As a bonus, having tests for your code means that if you do decide to make changes later you can quickly and easily ensure that you have not introduced any new bugs by running the tests.

Two Heads are Better Than One

If you are collaborating on a project, it is often useful to have someone else look over your code to help you find bugs. If you have been staring at your code for an extended period of time, you can often miss things that are obvious to a fresh set of eyes.

Some programming methodologies have this idea as a central tenant and call it *pair programming*. In pair programming two developers sit side-by-side at the same computer. While one developer writes the code, the other watches on and makes suggestion or corrections. After a while, the developers swap so the one writing becomes the one critiquing and *vice versa*. As well as catching bugs earlier, this method has the advantage of familiarising two developers with the same parts of a codebase. It is also a useful technique for mentor junior programmers since the less experienced programmer can watch and learn from the more experienced one.

3.19.7 Case Study

Consider the buggy code below, which is supposed to produce a summary of the values in a dictionary. We will walk through the steps of detecting the bug, using the Pucharm debugger to isolate and fix the bug, and then update documentation to warn against future bugs.

Python Code

```
#!/usr/bin/env python3
# COMP1040: The Craft of Computing
#
# NOTE: This code is deliberately incorrect! Use the PyCharm
#       debugger to walk through this code to find and fix the
#       bugs. There are at least two.
#
# The intention of the code is to take as input a dictionary
# containing lists of numbers as values and producing the sum
# of those values for each dictionary entry.

def summarise(results):
    """Summarise a dictionary by calculating the total of the
       list of numbers associated with each key."""

    for key, values in results.items():
        results = { key: total(values) }

    return results

def total(xs):
    """Computes the total of the numbers in the given list."""
    result = 0
    for x in xs:
        result += result + x

    return result

# Simple example usage
simple = { 'a': [1, 2] }
print(summarise(simple))

# More complex example usage
sales = { 'apples': [89, 55, 20], 'oranges': [30, 67, 90]}
print(summarise(sales))
```

Running the code produces the following output.

```
{'a': 4}
{'apples': 486}
```

The first thing to do is determine whether this matches our expectation of what the code should produce. Clearly, the second line is wrong since we're expecting two dictionary elements—apples and oranges—but the code only produces one. On closer inspection the first line is wrong too. The summary for `a` should be 3 not 4.

Try find the bug by mentally executing the code or using the PyCharm debugger. You can even write some unit test cases to help understand what the code should be doing. A live demonstration is given in the lecture.[16]

After fixing the bugs we note that there is an additional aspect of the code that may lead to future bugs—the `results` argument to the `summarise` function is modified by the function. This side-effect is not documented and may cause problems if the programmer is not expecting variables `simple` or `sales` to change. It is important, therefore, to ammend the docstring for the `summarise` function to bring this to the attention of the programmer.

[16] Answer: the bugs are on lines 17 and 25.

3.20 Lecture 20: Software Design

Learning Outcomes

- Appreciate that a large software project, like any engineering effort, requires detailed design and planning.
- Describe the different stages in the software development process including requirements elicitation, system specification, design (architecture, interface, component, etc), implementation, testing, and maintenance.
- Understand agile methods do not follow the above stages in strict order. Rather software development is seen as an iterative process.
- Define what a design pattern is and recognise a few common design patterns used in solving programming problems.
- Understand the importance of conventions and standards.

Overview

Software design and development is an entire field of study—this lecture gives just a brief taste of some of the important aspects of design. We cover some methodologies for designing good software and study the core problem of abstracting from a real world problem to a piece of software for solving that problem. We also present common design patterns that allow you to quickly map programming problems onto established solutions. Finally, we discuss some practical issues such as following programming conventions when implementing large software projects.

A lot of the code that we have written so far in this course can be done without much planning or forethought. However, as you develop bigger software projects planning and design become ever more important. We cannot hope to cover all of software design in a single lecture—there are entire courses on the topic—so will only present a brief glimpse of methodologies and techniques here.

3.20.1 The Importance of Good Software Design

There are countless examples of catastrophic software failures that provide evidence for why software design is important. Here are two classic examples.

- **Y2K (Design) Bug.** Leading up to the year 2000, developers and users started noticing a significant design failure in almost all software systems. The software has been designed to store years using two digits (e.g., "99" for "1999"). This affected an enormous number of systems and is estimated at costing USD 500 billion worldwide to fix.

- **Knight Capital "Flash Crash".** At 2:32pm EDT on 6 May 2010 a poorly designed (and tested) computerised trading system started switching between buying and selling stock. The result was an estimated loss of $440 million in about 30 minutes. Special "circuit breaker" software now exists to prevent such re-occurrences, but these preventative measures are implemented in software too, so also subject to failure.

Good software design doesn't only help to reduce bugs, it may also help to make development more efficient thereby reducing cost and improving time-to-market.

3.20.2 Development Methodologies and Design Principles

There are many different software development methodologies. Here we contrast two quiet different approaches. Both approaches incorporate the main tasks of software development—requirements gathering, system specification, system design, implementation, testing, and maintenance—differing mainly in how these tasks are staged.

The **Waterfall Method** is a methodology introduced as a strawman by Royce in 1970 as a *non-working model* of software development, however, one that mirrors closely the development practices in some industries. In this approach development proceeds as a strict sequence going from gathering requirements through design and implementation to verification and maintenance. This "big upfront design" approach is very brittle.

At the other end of the spectrum are **Agile Methods** of which there are many variants (e.g., eXtreme Programming, Scrum, etc.) but all with similar philosophy. In these methods, design and development is seen as an iterative process; accepting the fact that you will not get every aspect of the design perfect the first time before moving on to implementation. As a consequence, the philosophy is to produce something quickly that meets most of the requirements (termed *early delivery*) and then continuously improve. The approach requires adaptive planning and dynamic teams.

Design techniques included to various degrees within all modern software development methodologies include:

- **Use Cases.** Sometimes called User Stories, this technique captures requirements through short stories about how certain features or services might be used. They facilitate concrete discussions and are very helpful in making sure stakeholders and developers agree.

- **Test-Driven Development.** Here required functionality is specified through unit tests and other forms of automated testing. This allows very simple verification that certain functionality has been implemented but it is difficult to discuss behaviour with project members who are non-programmers. Some tools exist for systematically tuning high-level behavioural specification into executable tests (e.g., Python *behave!*).

- **Release Early, Release Often.** Agile methods place emphasis on having a working (but incomplete) system early (i.e., *minimal viable product*). This enhances feedback from users but care must be taken in managing expectations especially if only a subset of the functionality is implemented, which can frustrate users if they are expecting a fully working product. This also allows requirements to be validated with stakeholders.

- **Fail Fast.** Knowing when something is broken as soon as possible means that it can be fixed faster. Continuously testing code and verifying requirements with automated tests and user trials will help find (and fix) problems early. Don't work under the assumption that everything will work when put together the first time.

There are also a number of design principles that are adopted by many software development methods. Once again, we cannot hope to cover all design

principles in a single lecture. Moreover, most of these principles are only appreciated once you start using them (or not!)—experience and practice counts a lot in design. Some of the more common design principles include:

- **Keep it Simple (KISS).** Albert Einstein famously said "Everything should be made as simple as possible but no simpler." This is not as easy as it sounds and our first attempts at solving a problem are often way more complicated than they need to be (sometimes because we don't fully understand the problem). Try to keep breaking down a problem until what you are trying to do can be explained precisely in a few sentences.

- **Separation of Concerns.** Reasoning about multiple interactions is difficult. It is much easier to break down a problem so that you don't have to think about the whole problem. Rather focus on solving one aspect of your problem at a time and move between different layers of abstraction. A pattern that is often followed in software design is the separation of *model*, *view* and *control*.

- **Least Surprise.** Designs should be consistent and adhere to familiar conventions. For example, if you travelled to a new city and found that all stop signs were green that would be very weird (and dangerous). Design functions and objects so that naming, behaviour, arguments, etc. are consistent. This reduces *cognitive load* needed to develop and understand your software.

- **Don't Repeat Yourself (DRY).** Hunt and Thomas summarise this principle nicely as "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." The idea is that if there is a piece of information that needs to be passed around in a system, then keep one copy of that information otherwise you risk needing to synchronise the copies (i.e., keep them up to date).

- **You Ain't Gonna Need It (YAGNI).** It is very easy to dream up extension and improvements to your initial idea and be tempted to implement these well before you have completed your first design. Don't. Only code features when you actually need them otherwise you introduce breeding grounds for bugs. Revision control is a perfect antidote here—file an issue and address it later.

3.20.3 Design Patterns

A *design pattern* is a generic solution to a commonly occurring problem. It is not a complete design that can be directly implemented in source code. Rather it is a template that can be applied in many different situations and contexts. Because design patterns are well used and tested they simplify design decisions and speed software development. Design patterns are lower level than the design principles we discussed above. Below we study a small selection of the (technical) design patterns you may encounter.

The **Adapter** pattern allows two incompatible interfaces to work together typically by wrapping one object with a lightweight shim that translates between the two interfaces. A similar idea is the **Decorator** pattern, which allows

behaviour to be added to an object without affecting other objects of the same class. Python has a special syntax for implementing the decorator pattern. For example, assume we have a function to determine whether someone is tall given their height in feet and inches. This function can be arbitrarily complicated but in our example we assume that is just checks if the person is over six foot.

Python Code

```
def is_tall(feet, inches):
    """Returns whether someone is tall based on their height in feet
    and inches. We consider someone tall is they are over 6 feet."""

    return (feet + inches / 12) > 6
```

We may want to execute the function using SI units (i.e., meters) but not want to change the implementation of the function. A decorator is perfect for this situation.

Python Code

```
def si_units(fcn):
    """Decorator to convert perform calculation in SI units."""
    def convert(height):
        inches = 39.3701 * height  # convert from meters to inches
        feet = int(inches / 12)    # convert from inches to feet...
        inches -= 12 * feet        # ...and inches
        return fcn(feet, inches)
    return convert

@si_units
def is_tall(feet, inches):
    ...
```

We can now call our function using SI units:

Python Console

```
>>> print(is_tall(1.75)) # well under 6 foot
False
>>> print(is_tall(1.82)) # just under 6 foot
False
>>> print(is_tall(1.83)) # just over 6 foot
True
>>> print(is_tall(1.90)) # well over 6 foot
True
```

Decorators are useful when they need to be applied in a variety of contexts. You can think of them as rebinding the function, in the case above as `is_tall = si_units(is_tall)`. Of course in this example an alternative (and probably better) solution is to write a *wrapper* function, say `is_tall_metric` to make the operation explicit.

Python Code

```
def is_tall_metric(height):
    """Variant of is_tall for heights in meters."""
    inches = 39.3701 * height  # convert from meters to inches
    feet = int(inches / 12)    # convert from inches to feet...
    inches -= 12 * feet        # ...and inches
    return is_tall(feet, inches)
```

The **Iterator** pattern is one which we have already seen repeatedly throughout the course. It provides a mechanism to traverse every element of a container without having to know the implementation of the container. For example, we can iterate through every element of a set or a list in a consistent way without needing to know the storage details. (For a list we can also index elements by their position in the list but this already assumes the implementation and is prone to off-by-one errors as discussed in Lecture 3.19.)

Python Code

```
for element in container:
    do_something(element)
```

The **Factory** pattern is useful for creating objects when the class of the object is not known until runtime. For example, we may have a class that is specialized for different spoken languages, but we don't know which language to use until runtime. A factory can be used to create an object of the right class for us as the following example demonstrates.

Python Code

```
class Salutation:
    def __init__(self, name):
        self.name = name

    def sayHello(self):
        pass

    def sayGoodbye(self):
        pass

class English(Salutation):
    def __init__(self, name):
        super().__init__(name)

    def sayHello(self):
        print("Hello {}".format(self.name))

    def sayGoodbye(self):
        print("Goodbye {}".format(self.name))

class French(Salutation):
    def __init__(self, name):
        super().__init__(name)

    def sayHello(self):
        print("Bonjour {}".format(self.name))

    def sayGoodbye(self):
        print("Au revoir {}".format(self.name))

class Factory:
    def makeSalutation(self, name, language):
        if language == "English":
            return English(name)
        if language == "French":
            return French(name)
```

The last design pattern that we'll look at is the **Command** pattern. This is a pattern that used when you want to prepare a sequence of operations and then

execute them later. One of the nice features of the command pattern is that you can very easily reverse the operations by keeping a history (e.g., implement undo). The following code gives a simple example of the pattern.

Python Code

```python
class Engine():
    """Engine for executing commands. Also keeps a history so that
    commands can be undone. A command must have both an execute
    function and an undo function."""
    def __init__(self):
        self.history = list()

    def execute(self, command):
        self.history.append(command)
        command.execute()

    def undo(self):
        self.history.pop().undo()

class Increment():
    """Command to add some amount to every number in an array."""
    def __init__(self, array, delta = 1):
        self.array = array
        self.increment = delta

    def execute(self):
        for i in range(len(self.array)):
            self.array[i] += self.increment

    def undo(self):
        for i in range(len(self.array)):
            self.array[i] -= self.increment

class Zero():
    """Command to zero every element in an array."""
    def __init__(self, array):
        self.array = array
        self.backup = None

    def execute(self):
        self.backup = self.array.copy()
        for i in range(len(self.array)):
            self.array[i] = 0

    def undo(self):
        for i in range(len(self.array)):
            self.array[i] = self.backup[i]
        self.backup = None

# create and array and perform some operations and show results
engine = Engine()
array = [1, 5, 7, 2];                   print(array)
engine.execute(Increment(array));       print(array)
engine.execute(Increment(array, 5));    print(array)
engine.execute(Zero(array));            print(array)
engine.undo();                          print(array)
engine.undo();                          print(array)
engine.undo();                          print(array)
```

3.20.4 Conventions and Standards

Programming languages are incredibly flexible. In order to write software that is understandable, maintainable, and able to be re-used by others it is important to follow conventions and standards.

Many companies, open-source projects, and individual programmers have their own style. This is captured by a wonderful quote attributed to Grace Hopper, "the great thing about standards is that there are so many of them to choose from." And while this is true, the important thing is to *choose one*.

When working on a large team project or in an open-source community it is important to establish *and follow* a set of coding conventions. Not all team members will like all of the conventions, but the project as a whole will benefit from the consistency and uniformity that standards bring. This is especially important when you consider that most software is maintained by someone other than the original author(s).

In addition to consistency, standards also have other benefits. For example:

- They allow the creation of tools to assist with development, documentation, refactoring, and testing;
- They eliminate the need to make lots of unimportant decisions (should you use two space indentation or four?);
- They lower the barrier to learning a new tool or language (and prevent lock-in). E.g., standard key bindings for many applications (Ctrl-C and Ctrl-V for cut and paste, respectively).

In this lecture we use the words **standards** and **conventions** quite loosely. A rough delineation is that standards relate to rules for writing and formatting code that is enforced within a project, company or industry, whereas conventions are guidelines for how you should write your code so that it is consistent, robust and easily understood by a community of programmers.

We have already seen a number of coding conventions under the guise of *good programming practice*. For example, the use of descriptive variable names is a convention—there is nothing in the programming language that forces you to use meaningful names, but it turns out to just be a very good thing to do.

Naming Conventions

One of the biggest efforts in establishing coding conventions surrounds the naming of variables, functions and classes. Python restricts names to start with a letter or underscore and contain only letters, numbers and underscores. The rest is up to the programmer.

A fairly standard convention (in Python) is to use all capital letters for constants, to capitalise the first letter of each word in a class name, and to use lowercase with underscores for function and variable names. The following gives examples of each:

Python Code

```python
# constants
ALL_CAPS_WITH_UNDERSCORES

# class names
camelCase
UpperCamelCase

# variable names
underscore_separated
_underscore_prefixed
__underscore_surrounded__
```

Some naming conventions are so well established that they appear to be rules of the programming language. A good example is the use of "self" in member function declarations. This is, in fact, a convention and "self" can technically be replaced by any valid variable name (although that is not advisable). Consider the readability of the following class definitions.

```python
class MeaningfulClassName:
    # constructor
    def __init__(self, val1, val2):
        self.total = val1 + val2

    # a method to ...
    def scale(self, factor):
        return factor * self.total
```

```python
class C:
    # __init__
    def __init__(r, a, b):
        r.H = a + b

    # method
    def m(s, _123):
        return _123 * s.H
```

Code Layout

Another important convention has to do with code structure and layout. In Python indentation is significant so the language forces a certain layout. However, you still have a choice of tab or space, and how many spaces. The use of four-space indentation is the norm.

The maximum length of any one line of code is also a convention. Older programmers used 80 characters, but the new standard is more like 120. PyCharm indicates the maximum line limit with a margin line. Of course, this is not a hard rule and can sometimes be violated. When a line is broken there is a choice about where to break it. Try break lines within parentheses, at a comma or inline operation and continue the line at an indented position.

A very common (and useful) convention for Python scripts is to wrap the entry-point code in `if __name__ == "__main__":` to prevent execution when the script is imported into another piece of code (e.g., to reuse some of its internally defined functions).

Documentation and Commenting

Various standard exist for documenting code. In Python use **docstrings** for functions and classes. PyCharm can assist by providing a template for the docstring (by typing a triple inverted commas and hitting enter).

Python Code

```python
"""
Concise description of the function's purpose.

:param first_argument: A short description of the first argument.
:param second_argument: A short description of the second argument.
:return: A short description of the return value.

Optionally, more details about the function including, perhaps, a
description of how it works, what algorithms are implemented, any
side effects of the function, special cases that the caller should
be aware of, and examples for how it can be used.
"""
```

Comments, discussed in an earlier lecture, should also be used appropriately and always kept up-to-date with the code.

Python also has a well accepted convention for file headers. For example, the following file header is quite common:

Python Code

```python
#!/usr/bin/env python

"""
Docstring describing the module.
"""
```

Older code used to include special variables in the header like `__author__` and `__copyright__` but these have largely been superseded by package level LICENSE and README files.

Unit Testing

There are no hard rules about unit testing but the following conventions are popular:

- Build tests using the unittest module. Similar libraries exist for other programming languages.
- Have one unit test class for each class/module in your code.
- Name unit tests based on what functionality and behaviour is being tested.
- Do not mix test code with production code.

Miscellaneous Conventions

There are many other conventions that experienced programmers follow and that generalise across to any other programming language. These include:

- **Brevity.** Keep expressions short (i.e., no more than about one line). For example, if your lambda expression or list comprehension is more complicated then define a function.
- **Consistency.** When modifying someone else's code follow the existing conventions (even if they don't seem right to you at first).

- **More Whitespace.** Actually, use less: avoid extraneous whitespace, but don't over crowd your code either.

Many Python specific coding conventions are described in *PEP (Python Enhancement Proposal) 0008—Style Guide for Python Code*, which is based on the tenet that code is read more often that it is written. The PEP can be found at `https://www.python.org/dev/peps/pep-0008/`.

Different programming languages and communities of programmers adopt different conventions. For example, Windows developers tend to use the so-called Hungarian notation to name all variables. Here character prefixes such as `lpsz` are used to denote a variable's type, in this case a "long pointer to a zero-terminated string". The rest of the variable name is in CamelCase. Linux developers tend to use underscores to separate words in a variable name.

Another example is naming conventions for member variables (inside classes) in different programming languages. In C/C++ member variables are often prefixed by "`m_`". This is not necessary in Python, which forces the use of the "`self.`" prefix.

3.20.5 Starting a New Project

One of the most difficult questions you will ask when you begin a new project is *where do I start?* Unfortunately the answer depends on a number of factors—the complexity of the project, the experience of you and your team, the time and cost budget, the market and your customers. While the design and software engineering methodologies discussed above outline a formal process for developing software, in this section we give some practical advice on where to start.

- Design top-down; implement bottom-up. Do a rough design an iterate.
- Make sure you understand the problem. Can you break down the problem into subproblems, which are more easily solved?
- What is the high-level idea? Who are the main actors (people, things, and interactions) and how will these be modelled? How do these map onto code?
- Have you seen software that solves a similar problem? Can you base your design on that?
- Ask what are your inputs and what are your outputs? How will data flow through your system?
- Set up a software revision control repository.
- How will you represent your data? Will you need to implement a class? What interface should it have?
- Implement and test components in isolation to the whole system.
- Write test cases.
- Leave unimportant functionality until later (e.g., use `pass` or similar). You want to make sure your design works before spending too much effort getting every feature right.
- Refactor as you go, but don't optimise too early.

3.21 Lecture 21: Collaborating

Learning Outcomes

- Recognise techniques for more effective collaboration with others.
- Understand the role played by software revision control and issue tracking systems in group projects.

Overview

This lecture covers the skills needed to work on a software project in teams. We discuss some of the features of revision control systems, such as GitLab, which facilitate collaborative projects.

There are many approaches to programming in groups that have various pros and cons. All give some basic principles that you can use to help make decisions about your project. Moreover, there is a vast body of literature on how to effectively manage teams on technical projects. We won't go into too much detail here but there are some techniques you can investigate that may be applicable to your work:

- Pair programming
- Test-driven development
- Continuous integration

The focus of this lecture is to show you how to use the tools that you are already familiar with to help you to collaborate better.

3.21.1 Using GitLab to Collaborate

GitLab has many features that can be used to help you collaborate with your team mates:

- Permissions and group control
- Commit messages, comments, and version history
- Issues, issue assignment, and milestones
- Wiki pages and other documentation

Do not feel that you need to use all of the bells and whistles that come along with thes features; some (like milestones) are there for larger and longer-term projects. Furthermore, these tools should not replace other forms of communication, such as email, instant messaging, and face-to-face meetings, which are still needed to collaborate effectively.

Managing Group Members

The first step in collaborating on a project hosted by a remote revision control system is to set up your project and its members. To add members to a project in GitLab click on the Gear button for your project and choose “Members”.

You can then search for members to add by their real name, username (ANU ID) or email address. You also get to set the permissions for each member of your team. It's best to set up collaborators as Developer or Master giving them full permission to *push* code to the repository.

As the Owner of a repository you can always add or remove members later and change permissions for individual members.

Commit Messages and Comments

Everytime you commit code you are prompted to enter a commit message. For the small projects that you've been working on so far, and for the examples that we have seen in class, these commit messages fairly banal. However, when working on larger projects and in teams, commit messages become a invaluable mechanism for comminicating between team members. A well written commit message will help your team mates understand the changes that you have made. As noted in an earlier lecture, commit messages can also be used to close an issue.

Team members can also comment on invididual commits on to start a new discussion thread. To do this in GitLab, simply view the commit (from the "Repository ¿ Commits" tab) and write your comment in the space provided at the bottom of the screen. Comments can even be attached to specific lines of code (by clicking on that line). The original committer of the code will receive a notification depending on their Profile Settings.

Issues, Issue assignment, and Milestones

Issues are a fantastic way of planning tasks and tracking development. For example, at the start of a project you can create an issue of all the high-level tasks that need to be completed. And as development proceeds a new issue can be created for each new task (big or small) that comes up. This generation of issues can be done in project meetings or in an ad-hoc manner (and discussed online).

Issues can be assigned to specific members of your group. They will receive notifications and can list all issues assigned to them. This gives you a (crude) form of resource allocation, but more importantly let's everyone know who is responsible for what.

Milestones are collections of issues that can be assigned a due date. They are more useful for larger, ongoing projects.

Wiki Pages and the README file

A wiki is a collaboratively editable collection of web pages. Each GitLab project can have an associated wiki (you decide this when you first create the project). Wikis are useful for planning, keeping meeting notes, documenting links to external libraries, etc. Pages within the wiki are written in markdown, a text-based formatting language.

Generally wiki pages are edited online. However, one of the nice features of having a wiki integrated with revision control is that they can also be checkedout as a git repository for offline editing.

Special files `README.md`, `NOTES.md`, and `MEETINGS.md` are also a common way of managing software projects. Generally, the first place someone new to a

project will look is the `README.md` file. This is such a common convention that GitLab displays the `README` file on the main project page.

3.21.2 Communicating Effectively

There are plenty of other tools that facilitate effective communication—email, chat, cloud-based messaging (e.g., slack), etc. However, one of the most effective communication tools, when used properly, are face-to-face meetings.

Meetings

Using the tools discussed above, it should be possible to coordinate much of your activity on a project without having to be in the same room or in the same conference call. However, meetings are a good way to brainstorm and plan. They are especially useful to coordinate who will work on which part of the codebase so that merge conflicts can be avoided.

Come into meetings with a well-defined start and end time, a plan (i.e., agenda) of what will be discussed and end on time (have someone chair) with a list of "action items" for the team members to do before the next meeting. If something looks like it is dragging on, schedule a new time to discuss it further and move on so the other items can be dealt with.

If something complicated or not well defined needs discussing, allocate a set amount of time and try to work towards clarifying or simplifying. Make sure notes are taken about what the team figured out—most importantly, document all *action items* and *decision*.

3.21.3 Communication and Learning

Don't get too hung up on process—you are still learning how to code and project management is a completely different (albeit as important) skill to learn. The most important things to remember when collaborating are:

- Keep open all lines of communication: email, meetings, phone, messaging. But remember that other members of your team may operate on a different schedule to you—e.g., don't always expect an immediate response to an email.
- Tell your team mates as early as possible if something goes wrong or your plan needs revising. You can file an issue or send an email or bring it up in a meeting if there is one coming up soon.
- Don't get hung up on mistakes; solve the problem, learn from it, and move on—the aim is to learn.
- Look for opportunities to help your team mates and improve their understanding—don't just do it all yourself but by the same token don't just assign everything to someone else.
- Every team member should all feel a sense of ownership in the project—i.e., be able to point to a part and honestly say "I did this" and collectively point to the project and proudly say "And we did this."

3.22 Lecture 22: Refactoring Code

Learning Outcomes

- Be able to define refactoring as the process of re-writing code to improve design or performance without changing behaviour.
- Understand how software revision control and regression testing play an important part in the refactoring process.
- Be able to refactor small pieces of code, e.g., by encapsulating repeated code in a function.

Overview
This lecture covers the art of re-writing an existing piece of code to either make it simpler, faster, or provide additional functionality.

Refactoring is the process of re-writing software with the intention of making it simpler (i.e., easier to understand and maintain) or use less resources (i.e., faster or less memory hungry). Refactoring can also be done to prepare for new functionality to be added. The main point is that during refactoring the behaviour of the code should not change. As such, regression tests are a very important for making sure you don't introduce bugs during the refactoring process.

3.22.1 Refactoring Categories

Refactoring can be broken down into three main categories:

Refactoring Names. Renaming variables or functions is a common operation and useful for correcting spelling mistakes, coming up with a more appropriate/descriptive name, or avoiding clashes with library or built-in functions. Renaming can also be used to introduce constants in place of strings and *magic numbers*—you should always be suspicious of strings and numbers that are not used for output. Replacing these with constants will make your code much more readable and maintainable.

Renaming also occurs when you move code between modules or break up a large file into smaller ones to help with understanding and readability. This is especially true when functions that work on similar data are grouped together and separated from functions that have no clear relationship to one another.

Refactoring Processes. Long sequences of code are often difficult to understand and the operations better expressed as functions. A typical example might be to break a long script into functions that (i) read data, (ii) process pieces of data, (iii) combine the processed data, and (iv) visualise or output the results.

Duplicated code should always be turned into functions and reused. In the process of extracting common code you should try to recognise common usage patterns and generalise functions to cover these patterns (e.g., by introducing parameters).

Loops are another area where refactoring can be focused. Try simplify loops. For example, use iterators over `items` or `enumerate`. Sometimes list comprehension simplifies the construction of an output list. You can also explore using the `map` function for transforming one list into another.

When your data processing involves complex state you should encapsulate the data and operations that change the data into a class. This is typically needed when several data structures or elements need to be updated simulataneously (e.g., day and month when representing dates).

Refactoring Structure. Sometimes decisions made early in a design, such as choice of data structure, do not hold up as you add functionality or become cumbersome to deal with as you get deeper into your implementation. When this happens it may be time to change the type of data structure used to hold your data. For example, two equal length lists used together may be better stored as a list of tuples or dictionary.

Code structure can also be improved by simplifying dependencies between functions or classes. For example, if you are passing many values into or out of functions consider wrapping them in a dictionary or class. You may also consider splitting a single large class into smaller units that are easier to manage. Other simplifications may be to move methods up a class hierarchy so that derived classes get access to that method. Alternatively, if several classes have similar methods you can make them inherit from a common base (super) class.

3.22.2 Refactoring Guidelines

Since refactoring should not change behaviour you can use your existing code to develop unit tests. Write these before making any changes to the code so that you can regress against your previous "working" code at any stage of the refactoring process.

Once you start refactoring you will probably initially break functionality and then attempt to put it back together in a simplified way. Sandboxing and revision control can be a life-saver here. Make sure your code is committed and pushed to a remote repository before starting any major refactoring. If this get out of hand you can always compare to the last working version or, as a last resort, revert back to a working version and start again.

Inline with the above, try make lots of small changes instead of one big change. For each change make sure your unit tests pass and then commit the change to the repository. Of course big changes will sometimes be unavoidable. In such cases consider starting a new branch in the repository so that you don't hold up development on other parts of the project.

When refactoring avoid early optimization. As you make changes to the code computational bottlenecks may move to a different part of the process. Wait until you've made all your changes, then profile the code to understand where the bottlenecks really are. We have a whole future lecture on this topic.

Finally, always keep in mind the number one rule: **don't try to do too much at once—make small changes and test often.**

3.22.3 Refactoring Support in PyCharm

Many IDEs, including PyCharm, provide support for automating many refactoring steps. In the following case study, we will assume that we have written some (bad) code to get quickly get some statistics out of the wine quality dataset that we saw earlier. Specifically, we will write code to compute the average pH and alcohol content for red and white wines. The wine datasets can be downloaded from `https://archive.ics.uci.edu/ml/datasets/Wine`.

Our quick-and-dirty solution is shown below and was mostly written by replicating (cutting and pasting) large blocks of code.

Python Code

```
# Code to compute some statistics from the wine quality data sets.
#
# NOTE: This is deliberately poorly written so as to demonstrate
#       why refactoring is important

import csv

# Get the means for red wine
print('Red wine:')

print('\tpH:')
# Calculate the average pH content in the red wine data set
with open('winequality-red.csv') as datafile:
    sum = 0
    count = 0
    for row in csv.DictReader(datafile, delimiter=';'):
        sum += float(row['pH'])
        count += 1
print('\t\tmean: {:.2f}'.format(sum / count))

print('\talcohol:')
# Calculate the average alcohol content in the red wine data set
with open('winequality-red.csv') as datafile:
    sum = 0
    count = 0
    for row in csv.DictReader(datafile, delimiter=';'):
        sum += float(row['alcohol'])
        count += 1
print('\t\tmean: {:.2f}'.format(sum / count))

# Do the same for white wine
print('White wine:')

print('\tpH:')
# Calculate the average pH content in the white wine data set
with open('winequality-white.csv') as datafile:
    sum = 0
    count = 0
    for row in csv.DictReader(datafile, delimiter=';'):
        sum += float(row['pH'])
        count += 1
print('\t\tmean: {:.2f}'.format(sum / count))

print('\talcohol:')
# Calculate the average alcohol content in the white wine data set
with open('winequality-white.csv') as datafile:
    sum = 0
    count = 0
    for row in csv.DictReader(datafile, delimiter=';'):
        sum += float(row['alcohol'])
        count += 1
print('\t\tmean: {:.2f}'.format(sum / count))
```

One poor aspect of the program above is that it includes a lot of repeated code that can be consolodated. It is also difficult to extend, for example, if we wished to compute the maximum pH instead of the average. We can address these problems (and others) by refactoring the code.

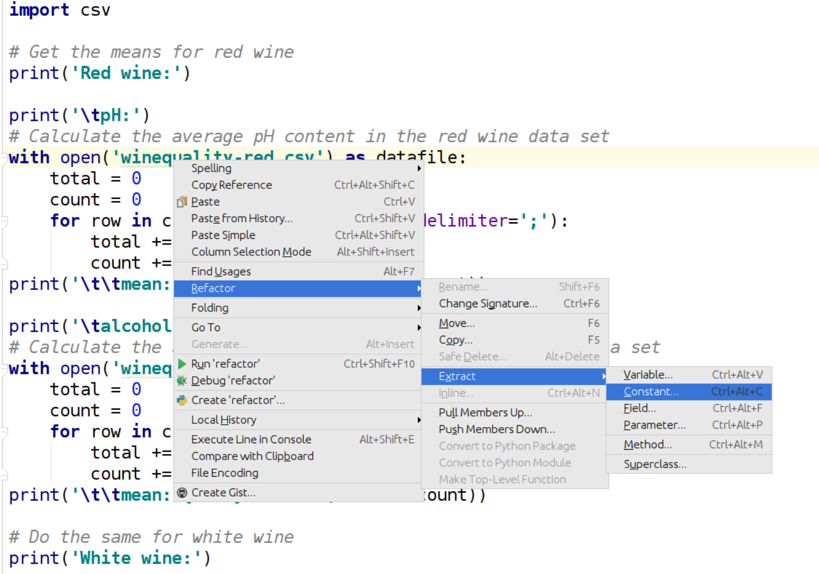


Figure 3.50: Snapshot of PyCharm IDE during refactoring operation.

We will tidy up the code in a number of steps:

- Rename `sum`, which clashes (or shadows) with a built-in function, with `total`. To do this we can right-click on one instance of the variable and select `Refactor|Rename...` (Shift-F6). Type in the new name and notice that PyCharm replaces all instances of the variable.

- The filenames "winequality-red.csv" and "winequality-white.csv" have been hardcodes, which is in general bad programming practice. Let's factor them out into constants which will make it easier if we need to change the file locations later. In PyCharm we can right-click on one of the filenames (e.g., "winequality-red.csv") and select `Refactor|Extract|Constant...` (Ctrl-Alt-C). This step is shown in Figure 3.50. Notice how PyCharm suggests a name for the constant and place it at the top of the code where it will be easy to redefine if needed. We can move elsewhere if we don't like the placement. Repeat for the other filename (e.g., "winequality-white.csv").

- Observe that the expression `total / count` appears in numerous places within the code. Let's make the code more readable by giving the expression a name. Right-click on the expression and select `Refactor|Extract|Variable...` (Ctrl-Alt-V). PyCharm suggests a variable name, but you can override the suggestion by typing a new one. Let's type `mean` (and notice that it changes in both places; the variable assignment and the variable use).

- At this point there is still quite a lot of duplication of code. For example, the "`with open`" suite of code appears in four different places and essentially does the same thing. Let's turn that code into a function. In PyCharm, highlight the code you want to turn into a function (say

the first occurance), right-click and select `Refactor|Extract|Method...` (Ctrl-Alt-M). Given the function a name, say "mean_pH", and click OK. PyCharm has done some work for us automatically, but we can still do better for example by removing the global variables created by PyCharm. The function new looks as follows:

Python Code

```
def mean_pH():
    with open(RED_CSV) as datafile:
        total = 0
        count = 0
        for row in csv.DictReader(datafile, delimiter=';'):
            total += float(row['pH'])
            count += 1
    return total / count
```

- We can further improve the code by parameterising the function to be able to pass in the filename and the attribute. This will then let us reuse the function elsewhere in the code. Again, PyCharm can help with the refactoring via `Refactor|Extract|Parameter...` (Ctrl-Alt-P). We will rename the function to "mean" (which we can do using `Refactor|Rename...` and even preview the change before making it) and make sure we provide the correct arguments when we call it.

Python Code

```
def mean(filename, attribute):
    """Compute mean of given attribute from the file."""
    with open(filename) as datafile:
        total, count = 0, 0
        for row in csv.DictReader(datafile, delimiter=';'):
            total += float(row[attribute])
            count += 1
    return total / count
```

In just a short time of refactoring, we have made some pretty good progress. The new code, shown below, is much more readable and maintainable.

Python Code

```
# Partially refactored code to compute statistics from the wine
# quality dataset.

import csv

RED_CSV = 'winequality-red.csv'
WHITE_CSV = 'winequality-white.csv'

def mean(filename, attribute):
    with open(filename) as datafile:
        total, count = 0, 0
        for row in csv.DictReader(datafile, delimiter=';'):
            total += float(row[attribute])
            count += 1
    return total / count

# Get the means for red wine
print('Red wine:')

print('\tpH:')
# Calculate the average pH content in the red wine data set
print('\t\tmean: {:.2f}'.format(mean(RED_CSV, 'pH')))

print('\talcohol:')
# Calculate the average alcohol content in the red wine data set
print('\t\tmean: {:.2f}'.format(mean(RED_CSV, 'alcohol')))

print('White wine:')

print('\tpH:')
# Calculate the average pH content in the white wine data set
print('\t\tmean: {:.2f}'.format(mean(WHITE_CSV, 'pH')))

print('\talcohol:')
# Calculate the average alcohol content in the white wine data set
print('\t\tmean: {:.2f}'.format(mean(WHITE_CSV, 'alcohol')))
```

Let's make one more change to put the repeated calls to "mean" into a loop. We will make this change manually. We can also very easily now add functionality, such as calculating the mean of another attribute, say "quality". The final code is shown below.

Python Code

```
# Version of code to compute means of some attributes from the
# wine quality data sets. Compare against version above.

import csv

RED_CSV = 'winequality-red.csv'
WHITE_CSV = 'winequality-white.csv'


def mean(filename, attribute):
    with open(filename) as datafile:
        total, count = 0, 0
        for row in csv.DictReader(datafile, delimiter=';'):
            total += float(row[attribute])
            count += 1
    return total / count


# Get results for red and white wine
for filename in [RED_CSV, WHITE_CSV]:
    print('Results for {}:'.format(filename))

    for attr in ['pH', 'alcohol', 'quality']:
        print('\t{}:'.format(attr))
        print('\t\tmean: {:.2f}'.format(mean(filename, attr)))
```

With the code in a much more managable state, it is now relatively easy to extend the functionality. For example, in the code below we include two additional statistics—max and min for each attribute.

Python Code

```
# Final version of code to compute various statistic over some
# attributes from the wine quality data sets.

import csv

RED_CSV = 'winequality-red.csv'
WHITE_CSV = 'winequality-white.csv'

def mean(xs):
    return sum(xs) / len(xs)

def read_attribute(filename, attribute):
    data = []
    with open(filename) as datafile:
        for row in csv.DictReader(datafile, delimiter=';'):
            data.append(float(row[attribute]))
    return data

# Get results for red and white wine
for filename in [RED_CSV, WHITE_CSV]:
    print('Results for {}:'.format(filename))

    for attr in ['pH', 'alcohol', 'quality']:
        print('\t{}:'.format(attr))
        data = read_attribute(filename, attr)
        for stat in [mean, max, min]:
            print('\t\t{}: {:.2f}'.format(stat.__name__, stat(data)))
```

Note that for brevity in this case study we have not included unit tests or discussed the use of revision control but as mentioned above these are both very important steps when embarking on a large refactoring task.

3.23 Lecture 23: Advanced Programming

Learning Outcomes

- Define the concept of an anonymous function.
- Be able to use lambda expressions as arguments to functions such as `sorted`, `map` and `filter`.
- Understand and implement simple recursive functions.

Overview

This lecture covers some advanced programming concepts such as recursion and lambda expressions.

With the programming concepts covered in the course so far you can solve a great many computational problems. In this lecture we will discuss some advanced programming concepts that, in certain situations, will make your solutions more elegant.

3.23.1 Anonymous Functions

Most modern programming languages support anonymous functions. They are functions that are created without a specified name (or identifier). This may seems strange given that we usually invoke a function by providing its name as part of the function call. We cannot do this with anonymous functions (unless we assign the function to a variable as we will see below). However, we can invoke the function at the point at which it is defined. For this reason anonymous functions are often small and can be thought of *throw away functions*, that is, functions that we use at the place where they are defined and not again.

Let's consider the concrete example of sorting a list of names and final course grades. We will assume that the names and grades are provided as a 2-tuple with name first as shown in the code below.

Python Code

```
# define final grades as list of (name, grade) pairs
final_grades = [
    ("Homer Simpson", 33.75),
    ("Marge Simpson", 72.5),
    ("Bart Simpson", 50.0),
    ("Lisa Simpson", 98.5),
    ("Maggie Simpson", 8.75),
    ("Carl Carlson", 68.75),
    ("Ned Flanders", 72.5),
    ("Barney Gumble", 37.5),
    ("Lenny Leonard", 40.0),
    ("Otto Mann", 85.0),
    ("Seymour Skinner", 81.25)
]

# sort and print
sorted_by_name = sorted(final_grades);
for student in sorted_by_name:
    print(student)
```

If we simply apply the `sorted` function and then print the names, we see that Python has returned the list sorted by the first element of each tuple (ties, which

do not appear in this example, are broken by looking at the second element). The output is shown below.

```
('Barney Gumble', 37.5)
('Bart Simpson', 50.0)
('Carl Carlson', 68.75)
('Homer Simpson', 33.75)
('Lenny Leonard', 40.0)
('Lisa Simpson', 98.5)
('Maggie Simpson', 8.75)
('Marge Simpson', 72.5)
('Ned Flanders', 72.5)
('Otto Mann', 85.0)
('Seymour Skinner', 81.25)
```

What if we wanted the list stored by grade instead of name? There is an old programming trick called *decorate-sort-undecorate*, which constructs a new list called the decorated list with the fields of interest as the first element of each tuple. This list is then sorted and then the extraneous fields removed. An alternative, provided in Python 3's `sorted` function is to provide an optional `key` argument.

The `key` argument refers to a function which returns a *key* for each item being sorted. The key is then used to compare items. So, for example, if we want to sort by grade then we would want the key function to return the second element in the (name, grade)-pair. One way to do this is to define the key function explicitly and provide that to the `sorted` function as in:

Python Code

```
def grade_key(student):
    return student[1]

sorted_by_grade = sorted(final_grades, key=grade_key)
for student in sorted_by_grade:
    print(student)
```

However, since the `grade_key` function is small and only really being used in one place, we can use an anonymous function to more efficiently write our code. In Python anonymous functions are declared using the `lambda` keyword. The general pattern for a lambda expression is

In theoretical computer science anonymous functions are related to field of study known as λ Calculus.

```
lambda <arguments>: <return_expression>
```

In the case of our student sorting example we would have:

Python Code

```
sorted_by_grade = sorted(final_grades, key=lambda stdnt: stdnt[1])
for student in sorted_by_grade:
    print(student)
```

which produces:

```
('Maggie Simpson', 8.75)
('Homer Simpson', 33.75)
('Barney Gumble', 37.5)
('Lenny Leonard', 40.0)
('Bart Simpson', 50.0)
('Carl Carlson', 68.75)
('Marge Simpson', 72.5)
('Ned Flanders', 72.5)
('Seymour Skinner', 81.25)
('Otto Mann', 85.0)
('Lisa Simpson', 98.5)
```

Here is a more complicated example where we sort by last name and then first name. The lambda expression is on Line 2.

Python Code

```
sorted_by_lastname = sorted(final_grades,
    key=lambda student: ''.join(student[0].split()[::-1]))
for student in sorted_by_lastname:
    print(student)
```

Note that the `sorted` function does not modify the items themselves, even though the key function is acting on the first element to swap the order of first and last names.

```
('Carl Carlson', 68.75)
('Ned Flanders', 72.5)
('Barney Gumble', 37.5)
('Lenny Leonard', 40.0)
('Otto Mann', 85.0)
('Bart Simpson', 50.0)
('Homer Simpson', 33.75)
('Lisa Simpson', 98.5)
('Maggie Simpson', 8.75)
('Marge Simpson', 72.5)
('Seymour Skinner', 81.25)
```

Other Common Uses

Another common use of anonymous functions is in converting or mapping from one list to another. Consider the problem of converting a list of heights in feet and inches to centimeters. Knowing that there are 12 inches in a foot and 2.54cm in an inch we can write the following expression for converting from the tuple `(<feet>, <inches>)` to the float `<centimeters>`,

```
lambda height: 2.54 * (12 * height[0] + height[1])
```

Python Code

```
height_imperial = [(6, 2), (5, 11), (6, 0), (5, 8), (5, 7)]
height_mertic = list(map(lambda x: 2.54 * (12 * x[0] + x[1]),
    height_imperial))
print(height_mertic)
```

Of course the same result could have been achieved with *list comprehension*,

Python Code

```
height_imperial = [(6, 2), (5, 11), (6, 0), (5, 8), (5, 7)]
height_mertic = [2.54 * (12 * x[0] + x[1]) for x in height_imperial]
print(height_mertic)
```

Lambda expressions can also be used to filter a list, for example, keeping only heights from our list above 6 foot,

Python Code

```
height_imperial = [(6, 2), (5, 11), (6, 0), (5, 8), (5, 7)]
height_tall = filter(lambda x: x[0] >= 6, height_imperial)
print(height_tall)
```

If you prefer, this too could be done using *list comprehension*,

Python Code

```
height_imperial = [(6, 2), (5, 11), (6, 0), (5, 8), (5, 7)]
height_tall = [x for x in height_imperial if x[0] >= 6]
print(height_tall)
```

Naming an Anonymous Function

Anonymous functions can be assigned to a variable, thus giving them a name (which remains valid while the variable remains in scope).

Python Code

```
# define a function to convert height from imperial to metric
convert_height = lambda x: 2.54 * (12 * x[0] + x[1])
# demonstrate the function
print(convert_height((6, 2)))
```

Closures

There are more sophisticated uses for lambda expressions in programming technique known as *closures*. These are advanced techniques which are beyond the scope of this course, but if you're interested you can read about them here:

`https://en.wikipedia.org/wiki/Closure_(computer_programming)`

3.23.2 Recursion

Recursion is a method for solving a problem where the solution is found by combining the solutions to smaller instances of the same problem. In the context of programming languages, a recursive function is a function that calls itself on smaller input as a step in evaluating its output. A *base case* is always needed to stop the recursion (i.e., a case where the recursive function no longer needs to call itself but can evaluate its output directly on its input).

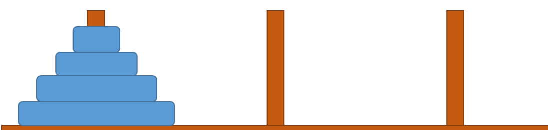


Figure 3.51: The Towers of Hanoi puzzle. The aim is to move the stack of disks from the leftmost column to the rightmost column with only every moving one disk at a time and never placing a larger disk on a smaller disk.

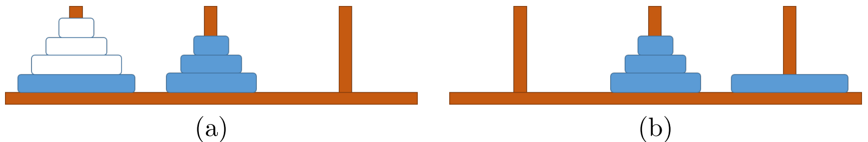


Figure 3.52: Developing the recursive solution for the Towers of Hanoi puzzle.

Example 3.23.1. The *Towers of Hanoi* puzzle is a classic problem that is used in computer science that is used to demonstrate the concept of recusion. In the puzzle you are given a stack of n disks of varying size and three rods or columns. Let's call them "A", "B" and "C". Initially the disks are placed on "A" in descending order of size as depicted in Figure 3.51 (for n equals 4). The objective of the puzzle is to move the entire stack from "A" to "C" with the following two restrictions: First, you may only move one disk at a time. Second, you may never place a larger disk on a smaller disk.

In deriving a recursive solution, consider a situation in which we are able to move the top $n-1$ disks from column "A" to the middle column, "B". The scenario is depicted in Figure 3.52. We know that we cannot move the $n-1$ disks in one go so there must be a sequence of moves to achieve this. However, for the moment we will not be concerned with how this is done, only that it is possible. Now, given that we have moved the the top $n-1$ disks to "B" we are free to move the biggest disk from "A" to "C" thereby making progress towards our solution—the first disk is in the right place.

It remains to move the rest of the disks to column "C". But, observe that this is just a smaller instance of the orginal puzzle. Namely, we can ignore the largest disk and reduce the problem to moving the stack of disks from column "B" to column "C". Now, let's return to the question of how to move the $n-1$ disks from "A" to "B" in the first place. Well, this is also a smaller instance of the original puzzle, except now the destination stack is "B" instead of "C". Again we can ignore the largest disk.

We now have a recursive algorithm for solving to the Towers of Hanoi puzzle. Let us denote the algorithm by `SolveTowersOfHanoi(n, A, C)`, meaning solve the Towers of Hanoi problem with n disks originally on column "A" with the goal of moving them to column "C". The algorithm can then be summarised as:

- `SolveTowersOfHanoi(n, A, C)`:
  - Move $n-1$ disks from "A" to "B" which can be done using `SolveTowersOfHanoi(n - 1, A, B)`

  - Move the n-th disk from "A" to "C"
  - Move $n - 1$ disks from "B" to "C" which can be done using `SolveTowersOfHanoi(n - 1, B, C)`

The only thing left to do is to define a base case. For the Towers of Hanoi problem the base case occurs when $n = 1$. Here we simply move the single disk from its starting column to the destination column. (We could have also define the base case to be when $n = 0$, which would be to do nothing.)

With the algorithm in hand, writing code to solve the Towers of Hanoi puzzle is fairly straightforward. We begin by developing some infrastructue code to display the solution. Our code makes use of a class to hold the data (number of disks and their current locations). We initialise the class by providing an argument specifying the number of disks. The constructor `_init_` below places all disks on column "A" and leaves the other columns empty. A disk is encoded by an integer specifying its size (from the smallest, 1, to the largest, n).

Python Code

```
class Hanoi(object):

    def __init__(self, n):
        """Initialise the board state with n disks."""
        self.n = n
        self.stacks = {'A': list(range(n, 0, -1)), 'B': [], 'C': []}
```

Let us write some helper member functions for displaying the state of the puzzle at any point. We will use a row of $2m + 1$ stars to depict a piece of size m. The code for displaying a single piece and the whole puzzle is then:

Python Code

```
    def print_disk(self, m):
        """Prints the m-th disk."""
        prefix = ' ' * (self.n - m)
        print(prefix + ('*' * (2 * m + 1)) + prefix, end='')

    def print(self):
        """Prints the puzzle state."""
        empty_row = (' ' * self.n) + '|' + (' ' * self.n)
        for row in range(self.n, 0, -1):
            print('{0:2d}: '.format(row), end='')

            for col in ['A', 'B', 'C']:
                if len(self.stacks[col]) < row:
                    print(empty_row, end='')
                else:
                    self.print_disk(self.stacks[col][row - 1])
            print()
        print('    ' + '-' * (self.n * 6 + 3))
        print()
```

We can now initialise and display a puzzle. What about moving disks? To do this we need to pop the disk from one column and push in onto another. Let us implement this as the `move` member function.

Python Code
```
    def move(self, src, dst):
        """Move the disk at the top of stack `src` to `dst`."""
        self.stacks[dst].append(self.stacks[src].pop())
        self.print()
```

To facilitate outputting the solution we have included a statement to print the puzzle state at the end of each move.

Let us start writing code to move a partial stack of m disks from one column to another. This corresponds to our recursion discussed above. The code is:

Python Code
```
    def move_partial_stack(self, m, src, dst):
        """Move stack of depth `m` from `src` to `dst`."""
        # base case
        if (m == 1):
            self.move(src, dst)
            return

        # recursion
        oth = self.other(src, dst)
        # 1. move m-1 disks from src to oth
        self.move_partial_stack(m - 1, src, oth)
        # 2. move m-th disk to dst
        self.move(src, dst)
        # 3. move m-1 disks from oth to src
        self.move_partial_stack(m - 1, oth, dst)
```

Note that our code makes use of the, as yet, undefined function `other`, which returns the spare column for the current puzzle instance. For example is `src` is "A" and `dst` is "C" then `other` should return "B". The function can be implemented in various ways. One way, that uses list comprehension, is:

Python Code
```
    def other(self, src, dst):
        """Returns the stack which is not `src` or `dst`."""
        return [disk for disk in self.stacks.keys()
            if disk not in [src, dst]][0]
```

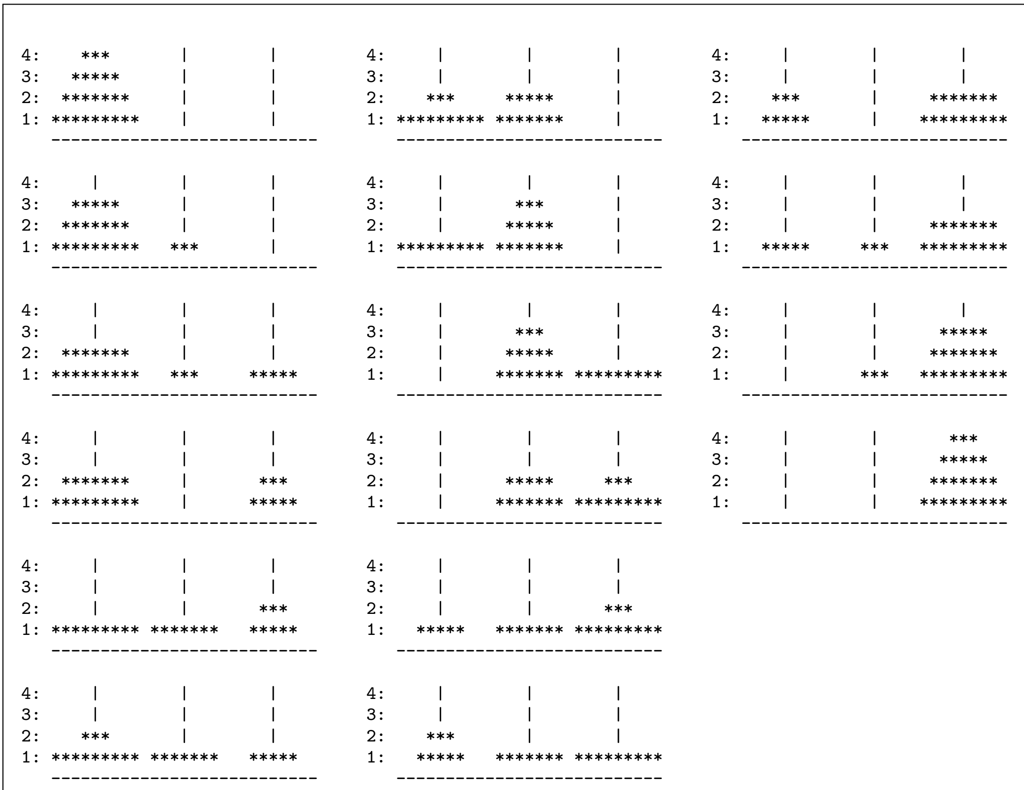
Finally, to solve the original puzzle we need to invoke `move_partial_stack` with the right arguments:

Python Code
```
    def solve(self):
        """Solve the Towers of Hanoi problem."""
        self.move_partial_stack(self.n, 'A', 'C')
```

To solve a Towers of Hanoi problem with four disks we construct the class with argument `4` and call the `solve` method:

Python Code
```
puzzle = Hanoi(4)    # construct the puzzle instance
puzzle.print()       # print the initial puzzle state
puzzle.solve()       # solve the puzzle
```

The code developed above generates the following sequence of moves (executed columnwise):



Every recursive function can be reimplemented as an iteration without recursion. However, thinking about the solution to a problem recusively may result in a more elegant formulation and simpler code.

Example 3.23.2. The Towers of Hanoi puzzle can also be solved via an iterative algorithm, which simply alternates between the following two moves:

- Move the smallest disk one column clockwise if the number of disks is even (or anti-clockwise if the number of disks is odd);
- Move another disk legally (there will only be one such move).

Note that discovering this algorithm is somewhat more difficult than coming up with the recursive solution above. But we can see that this is the exact set of moves that the recursive algorithm is generating in the example solution shown.

Recursion can also increase the cost of solving a problem if implemented naively. A classic example is when using recusion to compute the n-th number in the Fibonacci sequence.

The Fibonacci numbers are those in an integer sequence where each number in the sequence is defined as the sum of the previous two numbers, written mathematically as:

$$F_n = F_{n-1} + F_{n-2}$$

and where the first two numbers in the sequence are 0 and 1 (i.e., $F_0 = 0$ and $F_1 = 1$). Evaluating the equation above we see that the first few numbers in the sequence are:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

From the definition of the Fibonacci sequence it is very straightforward to code a recursive function to compute the n-th number as we show below.

Python Code

```
def fibonacci_recusive(n):
    """Compute the n-th Fibonacci number recursively."""

    # base cases
    if n == 0: return 0
    if n == 1: return 1

    # recursion
    return fibonacci_recusive(n - 1) + fibonacci_recusive(n - 2)
```

Here Lines 5 and 6 deal with the base cases and Line 9 handles the recusion. Note that if `n` is not one of the base cases then the return statement calls `fibonacci_recusive` twice. This is somewhat wasteful given that the first call, `fibonacci_recusive(n-1)`, will itself invoke the second call, `fibonacci_recusive(n-2)`. This results in double the computation needed. A much faster solution is to compute the n-th number directly as the following code shows.

Python Code

```
def fibonacci_iterative(n):
    """Compute the n-th Fibonacci number iteratively."""

    # initialisation
    f_last = 0
    f_curr = 1

    # iteration
    for i in range(1, n):
        f_curr, f_last = f_curr + f_last, f_curr

    return f_curr
```

In the direct solution we simply keep track of the current and previous fibonacci numbers in variables `f_curr` and `f_last`, respectively. At each step in the iteration we compute the new current value by adding `f_curr` and `f_last` and updating the new previous value to the current one. This is efficient in both storage and time.

3.23.3 Next Lecture

- Regular expressions

3.24 Lecture 24: Regular Expressions

Learning Outcomes

- Understand that regular expressions are a language for encoding patterns for string matching.
- Understand standard regular expression constructs including anchoring, alternation, grouping and quantification.
- Be able to read and write simple regular expressions.

Overview

This lecture covers the topic of regular expressions for pattern matching and replacement. We show how regular expressions can be useful in parsing or re-formatting data.

Standard string matching algorithms look for an exact match of a substring within a larger string. However, this is restrictive and we would often like to search for more expressive patterns, e.g., find either the word "dog" or "cat" in a document. A regular expression is a language for encoding such search patterns. Of course the above example (of searching for "dog" or "cat") could easily be done using two exact string matches, but, as we will see, regular expressions allow for way more flexible patterns (such as finding every word starting with "d" and ending with "g" and being no more than five letters long). Regular expressions are defined using a so-called regular language. They appear in both programming languages and stand-alone applications such as word processors and genomic databases, and can be used to search for patterns or validate input.

3.24.1 Learning from Examples

Regular expressions (or *regexp* for short) can be daunting when first encountered. The best way to learn about regular expressions, just like learning to program, is to build up from some simple examples. The following is a regular expression that searches for either the word "dog" or the word "cat".

The following quote attributed to Jamie Zawinski summarises the frustration many people feel when working with regular expressions: *Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.*

```
dog|cat
```

The vertical bar represents **alternation**, i.e., the choice between two alternative sub-patterns. Alternation can be extended arbitrarily, e.g.,

```
dog|cat|mouse|bird
```

From this example we can see that regular expressions define a (possibly infinite) set of strings to match, in this case the set is "dog", "cat", "mouse", "bird". Constructs more powerful than alternation allow use to express a much richer set very compactly, but let's first see how regular expressions are written in Python.

Regular Expressions in Python

The Python package `re` is used for regular expression matching.

Python Code

```
import re
sentence = "My dog can catch tennis balls."
m = re.search(r"dog|cat", sentence)
```

The prefix 'r' before the regular expression string passed to `search` tells Python to treat the string as a *raw string*. In other words, special characters like backslashes are left unaltered in the string allowing them to be interpretted as regular expression tokens.

Python Console

```
>>> if m: print("matched")
matched
>>> print(m.group())
dog
```

The `search` function returns the first location where the pattern matches within the string. But our example sentence actually contains multiple matches. The `findall` function allows us to extract all matches.

Python Code

```
import re
sentence = "My dog can catch tennis balls."
m = re.findall(r"dog|cat", sentence)
```

Python Console

```
>>> print(m)
['dog', 'cat']
```

3.24.2 More Regular Expression Constructs

Character classes specify a set of single character alternations. They are specified by square brackets and can support ranges (indicated by a dash). Negation is also supported using a caret (^). For example, the set of query strings "dog", "dag", "dig" can be encoded by the regular expression

`d[aio]g`

The **wildcard** (single dot character) is shorthand for the ultimate character set—i.e., match any single character. Other shorthands for common character classes are shown in Table 3.9.

Quantification qualifiers allow us to specify repetition of sub-patterns, for example, optionally allowing an "s" on the end of a word. Thus, the pattern

`d[aio]gs?`

would any of "dag", "dags", "dig", "digs", "dog", "dogs".

A very common quantification qualifier is the *match zero or more* quantifier. For example, matching the string "Dogs" or "dogs" followed by "cats" with any number of words in between them would be represented by the regular expression

`[Dd]ogs\s.*\scats`

Shorthand	Description
`.`	Matches anything other than newline.
`\d`	Matches any digit. Same as `[0-9]`.
`\D`	Matches non-digits. Same as `[^0-9]`.
`\s`	Matches a whitespace character.
`\S`	Matches a non-whitespace character.
`\w`	Matches letters, digits and underscore. Same as `[a-zA-Z0-9_]`.
`\W`	Matches anything other than letters, digits and underscore.
`\.`	Matches a dot (period).
`\\`	Matches a backslash (\).

Table 3.9: Shorthands for common character classes.

Quantifier	Description
`*`	Match zero or more of the preceding group.
`+`	Match one or more of the preceding group.
`?`	Match zero or one of the preceding group.
`{m}`	Match exactly m copies of the preceding group.
`{m,n}`	Match between m and n copies of the preceding group.

Table 3.10: Standard quantification qualifiers.

Table 3.10 shows the standard quantifiers.

Sometimes it is useful to **group** characters/tokens into sub-patterns. Quantification can then be applied to the group. Grouped matches can also be retrieved later (as we will soon see). A group is a subsequence of a regular expression surrounded by round brackets `()`. Compare the following three expressions.

Python Console

```
>>> sentence = "My dog can catch tennis balls."
>>> print(re.findall(r"dog|cat?", sentence))
['dog', 'ca', 'cat']
>>> print(re.findall(r"(dog|cat)?", sentence))
['', '', '', 'dog', '', '', '', '', '', 'cat', '', '', '', '', '',
    '', '', '', '', '', '', '', '', '', '', '', '']
>>> print(re.findall(r"dog|(cat)?", sentence))
['', '', '', '', '', '', '', '', '', 'cat', '', '', '', '', '', '',
    '', '', '', '', '', '', '', '', '', '', '']
```

The second two expressions match at every character in the string with the `?` quantifier matching greedily.

Note that when using grouping, the `findall` function will return matches as tuples that include the substrings that are matched by each subexpression. For example,

Python Console

```
>>> re.findall(r"((dog)|(cat))", sentence)
[('dog', 'dog', ''), ('cat', '', 'cat')]
```

We can anchor our expressions to the start or end of a string using metacharacters `^` and `$`, respectively. For example, matching filenames that end in

either ".py" or ".txt" can be done using:

```
\.(py|txt)$
```

Sometimes you will want to match the same subsequence that you have encountered earlier, e.g., "dogs and dogs" or "cats and cats" but not "dogs and cats". **Back references** allow you to capture a partial match and refer to it later in your regular expression. The subsequence is captured using round brackets `()`. The back reference, or captured subsequence, is specified using `\<n>` where `<n>` is counts the open brackets. For example,

```
((dog|cat) and \2)
```

3.24.3 Replacement

Regular expressions are also useful is specifying complex **find and replace** operations. In the most straightforward case we would replace a matched pattern with a fixed string. For example, replacing either "dog" or "cat" with the string "animal" could be done with the following Python snippet:

Python Code

```
print(re.sub(r"dog|cat", "animal", sentence))
```

We can also use back references to capture sub-patterns and use them in our replacement string. For example, replace "Stephen Gould" or "Mark Reid" with "Stephen" or "Mark", respectively can be done with:

Python Code

```
sentence = "COMP1040 is taught by Stephen Gould and Mark Reid"
pattern = r"((Stephen|Mark) (Gould|Reid))"
print(re.sub(pattern, r"\2", sentence))
```

Warning. This expression also replaces "Stephen Reid" and "Mark Gould" with "Stephen" and "Mark", respectively.

3.24.4 Regular Expressions as Finite State Automata

Strings are matched against regular expressions using an abstract machine known as an **automata**, which you will learn about in advanced courses on computation theory and algorithms. The process of compiling a regular expression to an automata takes time so if you're running the same regular expression against multiple strings it helps to precompile the expression.

Python Code

```
import re

# precompile the regexp
pattern = r"dog|cat"
prog = re.compile()

# process each line of a file for the pattern and print matches
with open("file.txt", "r") as fh:
    for line in fh:
        if prog.match(line):
            print(line.strip())
```

3.24.5 Final Word

Regular expressions can get complicated. Don't try to solve everything with a regular expression. You often have other programming tools at your disposal. For example, you may wish to verify valid email addresses on a web form. The following regular expression will validate that an address is RFC5322 compliant:

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]
+)*|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|  \\
[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z
0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:25[0-5]|2[0-4]
[0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9]
[0-9]?|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a
\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

However, there is no guarantee that anyone actually owns that email address. A simpler solution is to perform some basic match, say

```
\S+@\S+\.\S+
```

and then send a welcome email and see if it bounces!

3.25 Lecture 25: Data Pipelines and Command Line Processing

Learning Outcomes

- Appreciate that data processing often involves a pipeline of tasks.
- Be able to glue together pieces of existing software to achieve some data processing task.
- Understand the importance of re-purposing existing tools over writing your own.
- Have an appreciation of the history, breadth, and depth of command line tools and their importance for software development.
- Be familiar with some basic command line commands for navigating file systems.
- Understand that IDEs such as PyCharm provide often provide graphical interfaces to many command line tools, such as interpreters and revision control systems.
- Understand how to write your own simple command line tools in Python.

Overview

A typical application may involves getting some raw data, preprocessing it, performing some analysis on it, generating some output, and then interpretting and reporting the results. This lecture shows how these steps come together in a data processing pipeline. Moreover, often you can solve a problem, or a large portion of a problem, with pre-existing software tools if only the tools spoke to each other. This lecture will present strategies for gluing bits of code together to form a larger application and introduces the idea of command line tools and interfaces.

Despite the prevalence of graphically-oriented interfaces for operating systems such as in Mac OS X, Windows, and several Linux distributions, all of these systems still provide tools for interacting with the files on your computer via a *command line interface* (CLI). On OS X and Linux a command line can be open through the Terminal app, while on Windows this is achieved by open the Command Prompt application or through third-party programs such as PowerShell. PyCharm also comes with a command line interface which can be opened by pressing Alt-F12.

In this lecture we will focus on the commands and tools provided by what are called *POSIX*-compatible terminals such as those provided on Mac and Linux systems, or through PowerShell on Windows. Command line tools are extremely useful in the context of a data processing pipeline where multiple processing stages need to be glued together. We conclude the lecture with a demonstration of such a pipeline.

3.25.1 Basic Terminal Commands

When you first open up a terminal you will be greeted with a *command prompt*, as shown (for Mac OS X) in Figure 3.53. This prompt acts much like the

Python console we have been using elsewhere in this course: input is typed at the prompt and when return is pressed the input is interpreted, some output is displayed, and another prompt is shown.

Figure 3.53: A command prompt is shown when the Terminal application on Mac OS X is first opened.

The main difference between the command line and the Python console is that the command line understands a different set of commands to those understood at a Python console. A list of commonly used commands that you can use at a command line are shown in Table 3.11. These commands mimic several of the ways you can work with files and directories by dragging, dropping, and clicking them in the graphical user interface to your operating system.

Command	Description
`cd DIRNAME`	Change current directory to `DIRNAME`.
`ls`	Show the contents of the current directory.
`cp SRC DEST`	Copy the file `SRC` to `DEST`.
`mv SRC DEST`	Move or rename the file `SRC` to `DEST`.

Table 3.11: Some common command line commands for working with files.

As well as these basic commands for working with files, there are hundreds of other commands. These range from `grep` which can be used to search for strings or patterns of text within files, to `ssh` which lets you open command lines on remote machines, to `zip` which can be used to compress files and directories.

3.25.2 The Command Line and Programming

Several of the things you do when working with code in PyCharm can also be done via the command line. In fact, PyCharm actually uses the command line "under the hood" to implement some of its functionality.

Running Python Programs

When you run a piece of python code, PyCharm constructs a string that can is sent to a command line to run that code. For example, when you right-click on a file and select "Run" you will notice that in the window that displays the output of the code that is run there is text like:

```
/Users/mreid/bin/python /Users/mreid/code/mycode.py
```

This is a command line instruction to call the python interpreter located in the `/Users/mreid/bin/` directory and pass it the program in the `mycode.py` file located in the `/Users/mreid/code/` directory.

You can use the "Edit Configurations..." dialog window to construct more complex command line instructions from PyCharm, including being able to add arguments after the program name and other options.

Version Control

The other way in which PyCharm uses the command line is when executing version control instructions for git. When you clone, commit, pull, and push to and from repositories in PyCharm, command line versions of these instructions are called and their output is captured and turned into graphical representation within the PyCharm IDE.

If you open a terminal (Alt-F12) and run the `git` command, you will see a list of ways you can run Git from the command line:

```
$ git
usage: git [--version] [--help] [-c name=value]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

The most commonly used git commands are:
   add        Add file contents to the index
   bisect     Find by binary search the change that introduced a bug
   branch     List, create, or delete branches
   checkout   Checkout a branch or paths to the working tree
   clone      Clone a repository into a new directory
   commit     Record changes to the repository
   diff       Show changes between commits, commit and working tree, etc
   fetch      Download objects and refs from another repository
   grep       Print lines matching a pattern
   init       Create an empty Git repository or reinitialize an existing one
   log        Show commit logs
   merge      Join two or more development histories together
   mv         Move or rename a file, a directory, or a symlink
   pull       Fetch from and merge with another repository or a local branch
   push       Update remote refs along with associated objects
   rebase     Forward-port local commits to the updated upstream head
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index
   show       Show various types of objects
   status     Show the working tree status
   tag        Create, list, delete or verify a tag object signed with GPG

'git help -a' and 'git help -g' lists available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
```

This can sometimes be useful for more advanced operations. For example, to get the URL for the remote repository that your code is being feteched from or pushed to, you can use:

Terminal
```
$ git remote -v
```

You can even change where you'd like your changes to be pushed to:

Terminal
```
$ git remote set-url origin $NEW_URL
$ git push
```

where `$NEW_URL` is the full URL of the new remote repository.

3.25.3 Writing Command Line Tools in Python

Example 3.25.1. The following code shows a very simple Python program that can be run from the command line. We will assume its file is called `countwords.py`. When this code is given the name of a file as an argument on the command line, it will count the number of words (i.e., space delimited strings) in the given file.

Python Code
```
#!/usr/bin/env python3
import sys

# Get the first command line argument to as filename to process
filename = sys.argv[1]

# Read in each line, split on white space, and count words
wordcount = 0
with open(filename) as file:
    for line in file:
        # Break up line into words and add to count
        wordcount += len(line.split())

# Display the count
print(wordcount)
```

There are few difference between this program and the others you have seen so far in this course. First, the beginning of the file contains a cryptic looking `#!/usr/bin/env python3` comment. This is actually an instruction to the command line interpreter to treat the rest of the file as a Python program. More specifically, the command line will "hand over" the running of the file to the Python 3 interpreter that is found somewhere within the user's environment (that is what the `/usr/bin/env` part is doing).

The second thing to notice is that line 2 is `import sys`. This makes the `sys` module available to the program and, on line 5, we can see that there is a variable in that module called `argv` that is used. This variable is a list that contains the space delimited strings that make up the command that was called to run this program.

For example, if the program is called on the command line using `countwords.py words.txt` then the `sys.argv` list will contain

['countwords.py', 'words.txt']. Thus, the contents of sys.argv[1] will be the filename to process.

The remainder of this code is standard Python. The print statement at the very end will send the word count that is computed in the previous lines back out to the terminal.

Many scripts include optional arguments that control their functionality (e.g., setting verbose output). The getopt Python module provides a principled way of processing optional arguments. A short example is shown below.

Python Code

```python
import sys, getopt

# Usage statement to display if an error is encountered or
# the user provides the help option.
def usage(result = 1):
    print("USAGE: python3 " + sys.argv[0] + " [<OPTIONS>]")
    print("OPTIONS:")
    print("  -a <argument>  :: option with argument")
    print("  -b             :: option with no argument")
    print("  -h | --help    :: print help")
    sys.exit(result)

# Parse the command line
try:
    opts, args = getopt.getopt(sys.argv[1:], "a:bh", ["help"])
except getopt.GetoptError:
    usage()

# Make sure there are no "required" parameters
if len(args) != 0:
    usage()

# Start processing command line arguments
for opt, arg in opts:
    if opt in ("-h", "--help"):
        usage(0)
    elif opt in ("-a"):
        print("argument for -a is {}".format(arg))
    elif opt == "-b":
        print("option -b encountered")
```

Note that you can set command line options in PyCharm by editing the Run... configuration *script parameters*. However, it is more common to run scripts with command line options/arguments from the command line.

3.25.4 Data Pipelines

One of the big advantages of command line tools is that they facilitate data processing pipelines, i.e., gluing applications together to achieve some goal. A pipeline breaks a long operation into smaller self-contained stages. Typical stages in a data processing pipeline are shown in Figure 3.54 and summarised below:

- Acquisition and Extraction

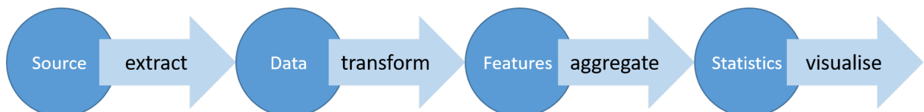


Figure 3.54: Typical stages in a data processing pipeline.

  - Getting the data onto your computer
  - Pulling out the information you need and discarding the rest

- Cleaning and Transformation
  - Text processing
  - Converting between data formats
  - Deriving new values

- Aggregation
  - Compute statistics
  - Grouping by shared column values
  - Fusion with other data

- Analysis and Visualisation
  - Plotting, reporting, etc.

Rather than write a single monolithic application to perform all of these steps, typical pipelines glue different applications together using a script. This is especially time-saving when existing tools can be used to perform parts of the pipeline. As we will see a master script can be used to execute stages implemented as command-line tools or through an API. In either case it is important to work out the data interface between the stages, typically by saving intermediate results.

Storing Intermediate Results

Saving intermediate results has a number of other advantages, including:

- Testing and Sanity Checks
  - Examining the output of each stage of pipeline makes testing easier and helps catch bugs earlier

- Manual Editing and Correction
  - Allows for manual intervention in some cases

- Recovery and Partial Evaluation
  - For long running processes, a failure mid-way through can cost time
  - Design pipelines so process can pick up where it was stopped
  - Especially important in large-scale clusters where machine failure is inevitable

You may find the Python `pickle` library useful for storing and retrieving intermediate calculations.

Warning: Modifying the code for a stage in a pipeline generally invalidates any results produced from that stage and all subsequent downstream stages. It is very important to clear all intermediate data when you modify your code. Resuming a data pipeline from intermediate results that were produced by a different version of code may result in invalid conclusions or non-reproducible findings.

Logging

Another important aspect of long-running data pipelines is *logging*. When writing a possibly long-running data pipeline, it is important to regularly output messages to the user explaining what is happening. This can include errors, warnings, progress messages, and debugging information. A very useful Python package for managing logging (including options to write the log to the screen and save it to disk) is the `logging` package.

A typical log will include timestamps and output a message at the start and end of each major step in the pipeline. These messages may also include information such as number of items processed, etc. Pipelines should also include tests for data validity, especially in early stages. Show-stoppers (e.g., missing files) shoudl result in an error being logged (and the pipeline halted). Lesser problems can be reported through warnings.

When processing large numbers of files, it is useful to indicate progress in some way. For example, output a dot or other indication at semi-regular intervals. For example, if it takes a few seconds to process a file and you have 10,000 files to process you could output a dot every 10–100 files.

3.25.5 Data Pipeline: Case Study

We demonstrate a data pipelining case study in which we are interested in analysing blog postings from a single author. We will use Mark Reid's blog, `http://mark.reid.name/blog/`. Our pipeline will operate in a number of stages:

- **Fetch.** First, we will fetch a list of blog URLs by parsing an index file. We will then download each blog posting and store the raw HTML to disk. We will make use of the `urllib` and `BeautifulSoup` Python libraries for downloading the URLs and converting them to text, respectively.
- **Extract.** Second, we will extract all the words from the blog and count the number of times each word appears. We will use the `NLTK` Python library for converting text to word counts.
- **Compare.** Third, we will compare blog postings by computing a similarity distance between the histogram of word counts. Here we will use the so-called cosine similarity metric.
- **Analysis.** Fourth, we will analyse the data to produce a histogram of similarities.
- **Graph.** Fifth (and last), we will produce a graph that visualises the most similar blog postings.

We have written the code—available from the course GitLab account—to be able to be run as separate scripts from the command line or controlled from a single script. This is for demonstration purposes. Usually, you will choose one one method or the other (depending on the availability of existing tools). The usage for each command line stage is shown below.

Terminal

```
$ ./fetch.py
USAGE: python3 ./fetch.py [<OPTIONS>] <LINKS_FILE> <POSTS_DIR>
Fetches blog posts from links found in <LINKS_FILE> and stores them
in the <POSTS_DIR> directory. The -b option updates the links file.
OPTIONS:
  -b <url>        :: base URL to update links file
  --log=<level>   :: change logging level
  -h | --help     :: print help

$ ./extract.py
USAGE: python3 ./extract.py [<OPTIONS>] <POSTS_DIR> <TEXT_DIR>
Reads HTML files from <POSTS_DIR>, extracts the text then stems and
tokenizes. The result for each HTML file is written to <TEXT_DIR>.
OPTIONS:
  --log=<level>   :: change logging level
  -h | --help     :: print help

$ ./compare.py
USAGE: python3 ./compare.py [<OPTIONS>] <FEATURES_DIR> <OUT_FILE>
Computes the pairwise similarity as a bag-of-words model
for all files in <FEATURES_DIR>.
OPTIONS:
  --log=<level>   :: change logging level
  -h | --help     :: print help

$ ./analyse.py
USAGE: python3 ./analysis.py <SIMS_FILE>
Plots a histogram of pairwise similarities.

$ ./graph.py
USAGE: python3 ./graph.py [<OPTIONS>] <SIMS_FILE>
Visualises similarities as a graph. The -t option sets a threshold
on which edges get displayed.
OPTIONS:
  -t <threshold> :: similarity threshold (default: 0.75)
  --log=<level>   :: change logging level
  -h | --help     :: print help
```

Command line applications can be run from Python using the `call` method from the `subprocess` standard library module.

3.26 Lecture 26: Optimising Code

Learning Outcomes

- Recognise when to optimise code and when to not.
- Identify the steps involved in optimising code.
- Understand how to use a profiling tool.

Overview

A software developer's time is often more valuable than a computer's. However, sometimes you really need software to run fast or use a small memory footprint. This lectures is all about strategies for optimising code.

Often your first attempt at writing a piece of software will not result in the most efficient code. However, if the software is well written (i.e., commented and tested), meets the requirements of the project, and is not slowing you down in other ways then you are done—remember *perfect is the enemy of good.*[17] Otherwise, you may need to refactor (or in some cases completely rewrite) your code to optimise it.

The first step to optimising a piece of code is to make sure it is correct, i.e., it produces the correct output. There is no point optimising something that does not work![18] The second step is to make sure everything is revision controlled and to write regression tests so that as you make changes to the code you can continuously check that it still works as expected.

There are a number of reasons that code could be running inefficiently. We have already briefly discussed complexity analysis and the fact that different algorithm (which solve the same problem) can have markedly different running times. However, other design choices and implementation details can also have a significant affect on running time. After making sure the code is working and is revision controlled, the next step is to determine which parts of the code are running slowly. For this we will need to do some *profiling.*

The basic steps for optimising code are then:

- Make sure the code is working (i.e., tested)
- Make sure the code is revision controlled
- Profile the code to determine bottlenecks (on common use cases)
- Refactor code to eliminate the bottlenecks
- Repeat

3.26.1 Profiling Code

Code profiling lets you know where you code is spending most of its time. Commercial IDEs (including the commercial version of PyCharm) come with in-built code profilers. Fortunately, Python comes with a number of standard libraries and tools for profiling. One of the most useful is `cProfile`, which can be invoked from the command line, e.g.,

[17] See `https://en.wikipedia.org/wiki/Perfect_is_the_enemy_of_good`.

[18] There is one caveat here: if the code is running too slowly for you to effectively test it then you may want to consider do some optimisation before the code is fully tested.

```
python -m cProfile -s tottime <filename>
```

or included within a script, e.g.,

Python Code

```python
import cProfile

cProfile.run("<command string>", sort="tottime")
```

The profiler generates timing information for each function in the program. By providing the `sort` optional argument we are asking for this information to be printed out in decreasing order of total running time, which is most useful for determining bottlenecks in the code. The exact format of the output will be discussed as we work through the case study below. Note that timing is subject to other processes running on your machine and profiling may change slightly between runs.

Case Study: Actor-Movie Database

To demonstrate the process of profiling and optimising code we will consider the following case study. We are given a file containing a list of actor-movie pairs for the top 250 movies according to IMDB. The file is a text file with one actor-movie pair per line. The actor and movie title are separated by a semi-colon (`;`) as illustrated below:

```
Carney, Thom; It! The Terror from Beyond Space (1958)
Biehn, Michael; The Terminator (1984)
Dijon, Alain; La dolce vita (1960)
Hoffman, Otto; Mad Love (1935)
Washington, Blue; Gone with the Wind (1939)
...
```

Our task is to write code that will find all actors co-starring in at least one movie (in the list of top 250 movies) with a given actor. The list of costars is to be returned in alphabetical order. We will develop the code by writing four functions:

- A function to read the data
- A function to find all movies starring a given actor
- A function to find all stars in a given movie
- A function that uses the previous two functions to find all costars of a given actor

The first function, `read_data`, reads the actor-movie pairs from file and is shown below. Here we choose to represent the data as a list of actor-movie pairs (2-tuple), where both actors and movies are represented as strings (the name of the actor or the title of the movie). The function opens the `actor-movie.csv` file and creates a CSV `reader` object specifying the field delimiter as “;”. It then proceeds to read each line of the file, extract the actor-movie pair, and append it to the `dataset` list. After reading each line, the function closes the file, prints out the number of actor-movie pairs read, and returns the list.

Python Code

```
import csv

def read_data(filename):
    """Read in dataset as list of (actor, movie) pairs."""
    fh = open(filename, "r")
    reader = csv.reader(fh, delimiter=";")

    dataset = []
    for actor, movie in reader:
        dataset.append((actor.strip(), movie.strip()))

    fh.close()
    print("...read {} actor-movie pairs".format(len(dataset)))
    return dataset
```

The second function, `find_movies`, returns a list of movie titles which star the given actor. The function, shown below, iterates over all actor-movie pairs in the list `dataset`. If the actor matches the given actor (`star` argument for the function) then the movie title is appended to the list of movies to be returned.

Python Code

```
def find_movies(star, dataset):
    """Find all movies starring the given actor."""
    acted_in = []
    for actor, movie in dataset:
        if actor == star:
            acted_in.append(movie)

    return acted_in
```

The third function, `find_actors` shown below, is the dual of the `find_movies` function just discussed. Given a movie title, the `find_actors` function will iterate through the list of actor-movie pairs and return a list of actors who all appeared in the given movie.

Python Code

```
def find_actors(title, dataset):
    """Find all actors starring in the given movie."""
    cast = []
    for actor, movie in dataset:
        if movie == title:
            cast.append(actor)

    return cast
```

The fourth and final function, `find_costars`, uses the previous three functions to achieve the task for this problem. Given an actor (`star`) and the pre-read dataset, the function first finds all movies in which the actor appears using the `find_movies` function. Then, for each movie title in the returned list it finds the cast for that movie using the `find_actors` function. For each actor in the cast, it adds the actor to the list of costars if it has not already done so (and if the actor is not the given actor). To ensure that the returned list of costars is in alphabetical order we sort the list each time a new actor is added to the list. We also print out debugging information, which helps us to check

that the code is working as expected. The code is shown below.

Python Code

```python
def find_costars(star, dataset):
    """Finds all costars who have appeared with the given actor in
    at least one movie and returns in sorted order."""

    # first find movies containing the star
    movies = find_movies(star, dataset)
    print("{} appears in {}".format(star, ", ".join(movies)))

    # initialise list of costars
    costars = []

    # for each movie search for the costars
    for title in movies:
        cast = find_actors(title, dataset)
        for actor in cast:
            if actor != star:
                print("{} appears with {} in {}".format(actor, star, title))
                if actor not in costars:
                    # insert into list and maintain sorted order
                    costars.append(actor)
                    costars.sort()

    # return costars
    return costars
```

For the purposes of this case study we will profile the code by first loading the dataset and then running the same query one thousand times to search for the co-stars of Tom Cruise. This will simulate multiple queries and give us sufficient running time statistics. A more elaborate scheme could be setup for testing with different queries, but that is unlikely to give different results in any material way. The code snippet for performing the profiling is shown below.

Python Code

```python
# search costars of Tom Cruise
def main():
    dataset = read_data("actor-movie.csv")
    for i in range(1000):
        costars = find_costars("Cruise, Tom", dataset)

# do profiling
if __name__ == "__main__":
    cProfile.run("main()", sort="tottime")
```

There is one more important step that we need to perform before we begin profiling and optimisation. That is to write some regression tests. For this problem we could write out the list of co-stars returned from one call to `find_costars` and save it to disk. Then everytime we modify the code we can check that we produce the same list. For brevity we omit regression test details for the remainder of the case study, but this is very important for real-world code optimisation.

Running the code produces detailed profiling information, the top few lines of which are shown below. There are some important items of the profiling output that you should note. First, the top line indicates the total number of function calls that were profiled (some built-in functions may not be profiled since Python

itself optimises them out) and the total running time. This running time may be slower than running the code without profiling since the act of profiling adds some small overhead. The output then lists timing information for each function, in this case ordered by the "tottime" column. The information shown is:

- **ncalls.** The total number of calls made to the function during the profiling run.
- **tottime.** The total amount of time spent running code within the function *excluding* any time taken by calls to sub-functions.
- **percall.** The previous column divided by the number of function calls.
- **cumtime.** The total amount of time spent running code within the function *including* time taken by calls to sub-functions.
- **percall.** The previous column divided by the number of function calls.
- **filename:lineno(function)** The function being profiled and its location in your source code.

```
        1305871 function calls in 4.071 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   251000    1.207    0.000    1.207    0.000 {method 'sort' of 'list' objects}
     3000    1.199    0.000    1.212    0.000 movie_optimization_demo.py:39(find_actors)
     1000    0.642    0.001    4.053    0.004 movie_optimization_demo.py:48(find_costars)
   253001    0.450    0.000    0.450    0.000 {built-in method print}
     1000    0.400    0.000    0.401    0.000 movie_optimization_demo.py:30(find_movies)
   253001    0.129    0.000    0.129    0.000 {method 'format' of 'str' objects}
   520246    0.026    0.000    0.026    0.000 {method 'append' of 'list' objects}
        1    0.013    0.013    0.016    0.016 movie_optimization_demo.py:17(read_data)
        1    0.002    0.002    4.071    4.071 movie_optimization_demo.py:74(main)
    22492    0.001    0.000    0.001    0.000 {method 'strip' of 'str' objects}
   ...
```

The first thing we notice is that the `sort` method is taking up sizeable chunk of the overall running time—about 30%, in fact. This relates to the sorting of the `costars` list that we do each time we add an actor on Line 21 of function `find_costars` above. Reflecting on what we wish to achieve it should be clear that we do not need to maintain the list of `costars` in sorted order during the inner workings of the function. All we need to do is make sure we sort the list before the function returns. Thus, we can remove the repeated calls to `sort` from Line 21 and move it to just before the `costars` list is returned on Line 24.

While we're looking at this part of the code, also notice that we keep checking whether the actor already exists in the `costars` list before adding, afterall we do not want the list to contain repeats of the same actor's name. It turns out that a `set` instead of a `list` data structure will do this for us. Let's make that change at the same time (but remember to change back to a list before returning). The modified code is shown below.

Python Code

```python
def find_costars(star, dataset):
    """Finds all costars who have appeared with the given actor in
    at least one movie and returns in sorted order."""

    # first find movies containing the star
    movies = find_movies(star, dataset)
    print("{} appears in {}".format(star, ", ".join(movies)))

    # initialise set of costars
    costars = set()

    # for each movie search for the costars
    for title in movies:
        cast = find_actors(title, dataset)
        for actor in cast:
            if actor != star:
                print("{} appears with {} in {}".format(actor, star, title))
                costars.add(actor)

    # sort and return costars
    costars = sorted(list(costars))
    return costars
```

We are now ready to run the code again to see how these changes have affected the profiling. The results are run are shown below.

```
        1056871 function calls in 2.333 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     3000    1.181    0.000    1.194    0.000 movie_optimization_demo2.py:39(find_actors)
   253001    0.407    0.000    0.407    0.000 {built-in method print}
     1000    0.385    0.000    0.385    0.000 movie_optimization_demo2.py:30(find_movies)
     1000    0.125    0.000    2.315    0.002 movie_optimization_demo2.py:48(find_costars)
   253001    0.117    0.000    0.117    0.000 {method 'format' of 'str' objects}
     1000    0.068    0.000    0.068    0.000 {built-in method sorted}
   252000    0.019    0.000    0.019    0.000 {method 'add' of 'set' objects}
        1    0.014    0.014    0.016    0.016 movie_optimization_demo2.py:17(read_data)
   269246    0.013    0.000    0.013    0.000 {method 'append' of 'list' objects}
    22492    0.002    0.000    0.002    0.000 {method 'strip' of 'str' objects}
   ...
```

We have already made some progress—the code is running almost twice as fast. Now looking down the profile list we notice that the second most time-consuming operation is the "built-in method print". This is low hanging fruit for optimisation. As a general rule printing out lots of information, writing lots of data to disk, or showing graphics and animations will slow down execution of a piece of code. By all means, use these when debugging your code but remove unnecessary output before shipping your final product.

Commenting out the print statements on Lines 7 and 17 we get the following profile output. Note that depending on how PyCharm or your terminal is configured, printing may take up more time (and be the dominant bottleneck initially—specifically if the output is not bufferred so that the screen is updated everytime print is called.

```
         549871 function calls in 1.695 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     3000    1.184    0.000    1.196    0.000 movie_optimization_demo2.py:39(find_actors)
     1000    0.368    0.000    0.368    0.000 movie_optimization_demo2.py:30(find_movies)
     1000    0.064    0.000    0.064    0.000 {built-in method sorted}
     1000    0.036    0.000    1.677    0.002 movie_optimization_demo2.py:48(find_costars)
        1    0.014    0.014    0.016    0.016 movie_optimization_demo2.py:17(read_data)
   252000    0.014    0.000    0.014    0.000 {method 'add' of 'set' objects}
   269246    0.012    0.000    0.012    0.000 {method 'append' of 'list' objects}
    22492    0.002    0.000    0.002    0.000 {method 'strip' of 'str' objects}
        1    0.001    0.001    1.694    1.694 movie_optimization_demo2.py:72(main)
        1    0.000    0.000    1.695    1.695 <string>:1(<module>)
   ...
```

As we can see from the profile information, changing the sorting and removing the unnecessary debugging information has resulted in a 58% reduction in running time (1.695 seconds versus 4.071 seconds). The new `find_costars` function implementing these changes is shown below.

Python Code

```python
def find_costars(star, dataset):
    """Finds all costars who have appeared with the given actor in
    at least one movie and returns in sorted order."""

    # first find movies containing the star
    movies = find_movies(star, dataset)

    # initialise set of costars
    costars = set()

    # for each movie search for the costars
    for title in movies:
        cast = find_actors(title, dataset)
        for actor in cast:
            if actor != star:
                costars.add(actor)

    # sort and return costars
    costars = sorted(list(costars))
    return costars
```

We now need to start thinking a bit about our high-level implementation. So far we have decided to simply read in the data and store it as a list of actor-movie pairs. However, looking at our profiling information we see that a large amount of time is spent looking up which actors appear in which movies. Thinking back to the lecture on data structures, we know that certain data structures, such as dictionaries are optimised for lookup. Perhaps a change of how we store/represent the data will lead to faster code. In particular, instead of storing the data as a list of actor-movie pairs we can store it as a dictionary indexed by movie, where the values are sets of actors appearing in that movie. There is some cost associated with pre-processing the data into this format, but that is more than saved when we come to look up actors. The new `read_data` function is shown below.

Python Code

```python
import csv

def read_data(filename):
    """Read in dataset as list of (actor, movie) pairs and return
    as a dictionary of movie casts indexed by movie title."""
    fh = open(filename, "r")
    reader = csv.reader(fh, delimiter=";")

    dataset = {}
    nCount = 0
    for actor, movie in reader:
        movie = movie.strip()
        if movie not in dataset:
            dataset[movie] = set()
        dataset[movie].add(actor.strip())
        nCount += 1

    fh.close()
    print("...read {} actor-movie pairs".format(nCount))
    return dataset
```

With the data in this new representation we will need to change the `find_movies` and `find_actors` functions.

Python Code

```python
def find_movies(star, dataset):
    """Find all movies starring the given actor."""
    acted_in = []
    for movie, cast in dataset.items():
        if star in cast:
            acted_in.append(movie)

    return acted_in
```

Python Code

```python
def find_actors(title, dataset):
    """Find all actors starring in the given movie."""
    return dataset[title]
```

With these changes in place and the code behaving correctly we are ready to perform another profiling run.

```
      295870 function calls in 0.146 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1000    0.064    0.000    0.064    0.000 {built-in method sorted}
     1000    0.037    0.000    0.126    0.000 movie_optimization_demo3.py:48(find_costars)
        1    0.015    0.015    0.019    0.019 movie_optimization_demo3.py:17(read_data)
   263246    0.014    0.000    0.014    0.000 {method 'add' of 'set' objects}
     1000    0.012    0.000    0.012    0.000 movie_optimization_demo3.py:35(find_movies)
    22492    0.002    0.000    0.002    0.000 {method 'strip' of 'str' objects}
        1    0.001    0.001    0.145    0.145 movie_optimization_demo3.py:71(main)
        1    0.000    0.000    0.146    0.146 <string>:1(<module>)
     3000    0.000    0.000    0.000    0.000 movie_optimization_demo3.py:44(find_actors)
     3000    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
   ...
```

Our final optimised version of the code is around 28 times faster than the initial version (0.146 seconds versus 4.071 seconds). This is fast enough for our needs so is probably a good place to stop. Observe that our code is still quite easy to follow and is maintainable.

Further Optimisation

If we really wanted to squeeze more performance out of the code we could start looking at some lower level optimisations. However, this often leads to code that is difficult to read and less maintainable. For example, looking back at our `find_costars` function we see that each time we get the cast for a movie we iterate over those actors and add them to the list of co-stars unless they are the actor given to the function (`star`). We can optimise this loop by simply adding the entire cast and then remove the given actor at the end. We have also used more a succinct *add* operation in the form of set union (`|=`). The resulting function is

Python Code

```python
def find_costars(star, dataset):
    """Finds all costars who have appeared with the given actor in
    at least one movie and returns in sorted order."""

    # first find movies containing the star
    movies = find_movies(star, dataset)

    # initialise set of costars
    costars = set()

    # for each movie search for the costars
    for title in movies:
        costars |= set(find_actors(title, dataset))

    # remove star, sort, and return costars
    costars.remove(star)
    return sorted(list(costars))
```

and corresponding profile output

```
        44870 function calls in 0.107 seconds

  Ordered by: internal time

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    1000    0.063    0.000    0.063    0.000 {built-in method sorted}
       1    0.015    0.015    0.018    0.018 movie_optimization_demo3.py:13(read_data)
    1000    0.012    0.000    0.012    0.000 movie_optimization_demo3.py:31(find_movies)
    1000    0.011    0.000    0.087    0.000 movie_optimization_demo3.py:44(find_costars)
   22492    0.002    0.000    0.002    0.000 {method 'strip' of 'str' objects}
       1    0.002    0.002    0.107    0.107 movie_optimization_demo3.py:64(main)
   11246    0.001    0.000    0.001    0.000 {method 'add' of 'set' objects}
       1    0.000    0.000    0.107    0.107 <string>:1(<module>)
    3000    0.000    0.000    0.000    0.000 movie_optimization_demo3.py:40(find_actors)
    3000    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
  ...
```

Arguably this code is still easy to understand (and the code is now 38 times faster than when we started), but you can imagine situations where further and further optimisation leads to less and less readable code. This can be a delicate trade-off and at some point you need to decide that your code is fast enough.

Case Study Summary

This case study has demonstrated some of the most important considerations when optimising code. First, as with any refactoring, always check that your code remains correct as you make changes. Second, using profiling to determine where the bottlenecks are. Third, removing unnecessary output, finding better representations and algorithms, and pre-computing (or caching) information are the best ways to improve performance. Regarding the last point, there is often a trade-off between an algorithms speed and space/memory requirements. In this case study we have focused on improving speed. However, sometimes it is also important to optimise for memory (e.g., when writing software that runs in embedded devices).

3.26.2 Amdahl's Law

Amdahl's law is a statement on the maximum expected improvement in the performance of a system if only part of the system is optimised. Consider a piece of software that takes T seconds to complete a given task. We identify a bottleneck in the code, which we then optimise from taking T_1 seconds to T_2 seconds. How much faster is the optimised code?

The optimised code now takes $T - T_1 + T_2$ seconds to run. In terms of speedup this is

$$\frac{T}{T - T_1 + T_2} \tag{3.1}$$

times faster. If the bottleneck was originally taking 20% of the running time and we spent a lot of development effort to get it to run in half the time (i.e., $T_2 = \frac{1}{2}T_1$), then we have only sped up the end-to-end performance by a factor of

$$\frac{T}{T - 0.2T + 0.1T} = 1.11 \tag{3.2}$$

That is, the software now only 11% faster, depsite the bottleneck component being 50% faster.

Amdahl's law suggests that it is only worth spending time optimising parts of the code that take a *significant* portion of the total running time. Optimising pieces of code that do not have a significant effect on the end-to-end running time, even if inefficient, is a waste of your time as a developer.

3.27 Lecture 27: Defensive Programming

Learning Outcomes

- Understand the importance of guarding against errors in code.
- Develop practices for writing less buggy and more maintainable code.
- Recognise that just because a programming language allows you to write code in obscure ways does not mean that you should write that way.

Overview

Garbage in, garbage out. This lecture discusses how to write code that guards against bad data and unexpected events. We distinguish between assertions used during development and error checking and processing during production.

Debugging is hard, so when writing code try make you code easy to understand, test and debug. You will thank youself later. While writing code think to yourself how can I avoid debugging it later? For example, consider the following two code snippets.

```
print(["Over Ten",
    "Under 10"][x < 10])
```

```
if x < 10:
    print("Under 10")
else:
    print("Over Ten")
```

Both code snippets do the same thing. The code on the left is more concise but also much more difficult to understand (and debug especially if it appears within a 1000-line piece of code). The code on the right is less "clever" but much better. The developer of the Unix operating system and C programming language, Brian Kernighan sums up up this idea nicely with "Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"

This lecture discusses some techniques that you can use when writing code that will make your code more robust and easier to debug when things do go wrong.

3.27.1 Unit Tests

Probably the single most useful piece of advice for writing robust code is to **use unit tests**. Write the unit tests before you implement your code. This will help you to understand what you want each function to achieve. The same applies to refactoring and optimising code—write unit tests first and keep testing as you go to make sure you haven't broken anything obvious. The same applies yet again to debugging—find a bug, write a test (which also acts to document the bug), then fix the bug.

If it seems too difficult to write a unit test for some code then the code probably needs to be redesigned. You should always be able to write testable code. Otherwise you have no way of knowing if your code is working.

But remeber that unit tests cannot possibly cover every scenario. Even with the best unit tests bugs may still happen. Even worse bugs may be in the unit tests themselves. Defensive programming techniques give us additional techniques to help reduce bugs in our code.

3.27.2 Production versus Development

When **developing code** or writing small scripts for your personal use it is okay for the code to crash occasionally when it encounters something unexpected, e.g., a misspelled filename. If that happens you are right there to fix the problem and run your code again.

However, **production code** needs to handle errors gracefully. It should check user input and return helpful error messages to the user if something goes wrong. Furthermore, it needs to maintain data integrity of the application. Imagine what would happen if your email client crashed and in the process deleted all of your emails? Your code should never leave data in an inconsistent state.

Malicious users may deliberately try to crash a production system. Honest users may accidently crash a system too. It is very important to check user input and reject input that will result in excessive resource usage. For example, don't allow a user to attach very large files to an email. Worse than exessive resource usage (which results in denial of service) are inputs that expose security vulnerabilities. A classic example is allowing a user to enter malicious HTML code or SQL queries as input into a web application (known as cross-site scripting or SQL injection).

3.27.3 Guidelines for Writing Defensive Code

Here we describe a few techniques for writing defensive code:

Understanding Pre- and Post-conditions

When writing a function it is useful to think about the pre- and post-conditions for the function. **Pre-conditions** state what should be true when a function is called. The function is not guaranteed to perform correctly if the pre-conditions are not satisfied. **Post-conditions** describe what will be true when the function returns. If the pre-conditions are met, then the post-conditions are guaranteed to be true.

Pre-conditions apply to function arguments. They can (and should) be explicitly tested within the code. For example, think about the pre-conditions for the following function signature and how you would test them.

Python Code

```python
def set_time(hours, minutes, seconds):
    """What are the pre-conditions?"""
```

Post-conditions can be checked with unit tests.

Assertions and Exceptions

Asserts are a systematic way to guard against errors in code. They add some

overhead to execution time, but can be turned off with the `-O` command line option or in PyCharm via the "Interpreter options" under `Run|Edit Configuration...`

The following code provides an example of an assert.

Python Code
```
def square_root(x):
    """Compute a square root."""
    assert x >= 0, "negative argument"
    return math.sqrt(x)
```

Failed asserts will raise an `AssertionError` exception. This usually causes Python to terminate and display an error message.

Exceptions are more general than asserts. They are a way of alerting your code that an unexpected error has occurred and provides a mechanism for handling the error. For example, exceptions can be used to catch errors associated with trying to open a non-existing file or performing an illegal mathematical operation (divide-by-zero or taking the square root of a negative number). The following code snippet provides a general pattern for exception handling:

Python Code
```
try:
    do_something()  # may cause an error
except:
    handle_error()  # run if error
else:
    do_even_more()  # run if no error
finally:
    clean_up()      # run always
```

A list of common exceptions is shown in Table 3.12.

Exception	Description
`Exception`	Base class.
`OverflowError`	Numerical overflow has occurred.
`ZeroDivisionError`	Division or modulo by zero.
`AssertionError`	As assert has failed.
`NameError`	A variable or function does not exist.
`IndexError`	Invalid sequence index.
`IOError`	E.g., file not found.

Table 3.12: Common Exception Types.

You can write code to handle different types of exceptions differently. For example, consider the following piece of code, which opens a file containing a number per line and computes the square root of each number. Two different types of error can occur; either the file does not exist or a number in the file is negative. The code treats these errors differently without needing to explictly check when the file is opened or each time a number is read.

Python Code

```python
try:
    # open file and print out the square root of each line
    with open("testfile.txt", "r") as file:
        for line in file:
            print(math.sqrt(float(line)))

except IOError:                    # handles input/output errors
    print("file not found")

except ValueError:                 # handles numerical errors
    print("negative number")
```

You can trigger an exception in your own code using the `raise` keyword as the following demonstrates.

```python
def square_root(x):
    """Compute a square root."""
    assert x >= 0, "negative argum
    return math.sqrt(x)
```

```python
def square_root(x):
    """Compute a square root."""
    if x < 0:
        raise ValueError("negative argument")
    return math.sqrt(x)
```

Unhandled (uncaughted) exceptions bubble *upwards*. That is, they move up the call stack through each successive function call until they are caught. If an exception reaches the *top* of your code, i.e., the main script, then Python terminates with an error.

Explicit is Better Than Implicit

Explicit is better than implicit is one of the Python aphorisms. Unfortunately the Python language does not enforce this and in some situations encourages implicit behaviour. This is a habit you need to develop as a programmer, and is especially important if inexperienced programmers are part of the team or your project involves coding in a few different languages which may have different rules. The following two examples show that explicit use of parentheses and explicit return statements make the intent of the code much more clear.

```python
if x < 10 and y > 20 or z < 0:
    do_something(x, y, z)
```

```python
if (x < 10 and y > 20) or (z < 0):
    do_something(x, y, z)
```

```python
def do_something(x, y, z)
    n = x + y + z
```

```python
def do_something(x, y, z)
    n = x + y + z
    return None
```

Guard Against Language Idiosyncrasies

Every programming language has quirks and idiosyncrasies. Some people think that using these idiosyncrasies is smart and shows that you are proficient in a language; often, however, their use leads to hard to find bugs. You should definitely know about the quirks of a language; but resist the temptation to use them in code that you write. One good example in Python is operator chaining.

Python Code

```
>>> 1 == 2
False
>>> False is not True
True
>>> 1 == 2 is not True
False
```

Always use parentheses to disambiguate meaning in logical and arithmetic expressions (expecially if you frequently switch between programming in different languages).

Python Code

```
>>> (1 == 2) is not True
True
>>> (1 == 2) and (2 is not True)
False
```

3.27.4 Next Lecture

- Different programming languages
- Learning a new language

3.28 Lecture 28: Programming Languages

Learning Outcomes

- Recognise that there are multiple programming languages, each with their own strengths and weaknesses. And that the choice of programming language for a project depends on many factors.
- Understand that many programming concepts in Python map across to other programming languages.
- Be equipped with skills to leverage knowledge of Python to learn a new programming langauge.

Overview

In this lecture we give a very brief taxonomy of programming languages. We introduce some popular programming languages other than Python and discuss where each might be used. We also discuss common features of different classes of programming languages. Finally, presented with having to work with a new programming langauge, this lecture gives some ideas on how to learn the new langauge. We focus on what general constructs you can expect to find in all mainstream languages.

This course has used the Python programming language as a vehicle for teaching you the **craft of computing**. However, Python is only of many programming languages that are widely used in academia and industry (some of the most popular ones being C/C++, Java, Javascript, PHP, Matlab). There are similarities between Python and many of these (imperative) programming languages.

Moreover, all of the tools and techniques taught in the class (text editors, IDEs, revision control, debuggers, refactoring and code profiling) extend to other programming languages, and are just as an important a component of software development as the programming language itself.

In this lecture we will give a brief tour of some other programming languages and some tips on how to learn another language given that you now know Python. We will restrict our attention to imperative programming languages and do not get into any details about distinctions in implementation such as compiled versus interpretted, etc.

Ultimately the choice of programming language for any given project will depend on many things: existing codebases, available programmers with experience in that language, suitability of the language for the task, third-party libraries, etc. There is no one right language and an important skill is to be able to move between languages. For an interesting visualization on the influence and evolution of programming languages take a look at:

`https://exploringdata.github.io/vis/programming-languages-influence-network/`

3.28.1 Hello World

Often the first program you write in a new programming language is called *hello world*. Ironically in this course, we leave this program for last. The aim of the program is to display the string "hello world". This exposes you to the process of editing and running a piece of code for that language. For some languages

there is an intermediate step required to *compile* the source code into a form that can be run. The following code snippets show *hello world* written in some common languages.

Python Code

```python
# python
print("hello world")
```

C/C++ Code

```cpp
// c++
#include <stdio>

int main() {
    std::cout << "hello world\n";
}
```

Java Code

```java
// java
public class Hello {
    public static void main() {
        System.out.println("hello world");
    }
}
```

Javascript Code

```javascript
// javascript
document.writeln("hello world");
```

3.28.2 Code Concepts

As you've probably gathered by now, imperative programming languages are actually all very similar. **Mostly the difference is syntactical rather than conceptual.** So once you can program in one language it becomes fairly easy to learn another language. Of course to become proficient in a language, like any skill, you still need to put in a lot of practice and use the language regularly. Below we list the core concepts to think about when looking at a new language.

- Code structure and layout (line endings, defining code blocks, comments)
- Variables and expressions
  - Some programs are strongly-typed and require variables to be declared before used
  - Languages can differ in variable scope rules
- Declaring and calling functions
  - Languages differ in how functions are declared and called
  - Some languages don't support optional arguments
- Program control flow
  - All programming languages support conditional execution and loops

- Objects and classes (not supported in all languages)
- Data structures
  - Most programming languages include the main data structures we have seen in this course, but sometimes these are not a core part of the language (they need to be imported through libraries)
- Libraries and modules
  - Languages differ in how code is organised into libraries and how library code is used

Circle Program

The circle program that we introduced in Lecture 3 is a good example to demonstrate how variables are declared/assigned and expressions written in a language. It also shows basic program structure and how to display (text) output.

Python Code

```
# calculate area and circumference of a circle

import math

radius = 10
area = math.pi * radius ** 2
circumference = 2 * math.pi * radius
print(area, circumference)
```

C/C++ Code

```
// calculate area and circumference of a circle

#include <cstdlib>
#include <cmath>

int main()
{
    double radius = 10;
    double area = M_PI * pow(radius, 2.0);
    double circum = 2.0 * M_PI * radius;
    std::cout << area << ", " << circum << "\n";
}
```

Javascript Code

```
// calculate area and circumference of a circle

var radius = 10;
var area = Math.PI * Math.pow(radius, 2);
var circumference = 2 * Math.PI * radius
console.log(area + ", " + circumference)
```

For Loops and Iteration

Loops are common feature of most programs. Here we repeat the simple example of looping through a list of grades in order to count the number of high

distinctions. In the C++ example we show the grades stored in an array data structure, which requires elements to be accessed by indexing. Other data structures (e.g., `std::list`) support iterators similar (in concept) to Python's iterators.

Python Code

```python
# count high distinctions
grades = [65, 90, 70, 85, 92, 73, 62, 68, 81, 68]

num_high_distinctions = 0
for g in grades:
    if g >= 85:
        num_high_distinctions += 1
```

C/C++ Code

```cpp
// count high distinctions

int grades[] = {65, 90, 70, 85, 92, 73, 62, 68, 81, 68};

int main()
{
    int num_high_distinctions = 0;
    for (int i = 0; i < sizeof(grades)/sizeof(int); i++) {
        if (grades[i] >= 85)
            num_high_distinctions += 1;
    }
}
```

Javascript Code

```javascript
// count high distinctions

grades = [65, 90, 70, 85, 92, 73, 62, 68, 81, 68];

var num_high_distinctions = 0;
for (var i = 0; i < grades.length; i++) {
    if (grades[i] >= 85) {
        num_high_distinctions += 1;
    }
}
```

Functions

Another feature you will see in almost every program is a function. Here we demonstrate function declarations in Python, C/C++ and Javascript.

Python Code

```python
def convert_to_letter_grade(grade):
    """Converts from a numerical grade to a letter grade."""

    if (grade >= 85.0): return "HD"
    elif (grade >= 75.0): return "D"
    elif (grade >= 65.0): return "CR"
    elif (grade >= 50.0): return "P"
    else: return "F"
```

C/C++ Code

```
// Converts from a numerical grade to a letter grade
std::string convert_to_letter_grade(double grade)
{
    if (grade >= 85.0) return "HD";
    if (grade >= 75.0) return "D";
    if (grade >= 65.0) return "CR";
    if (grade >= 50.0) return "P";

    return "F";
}
```

Javascript Code

```
// Converts from a numerical grade to a letter grade
function convert_to_letter_grade(grade) {

    if (grade >= 85.0) return "HD";
    if (grade >= 75.0) return "D";
    if (grade >= 65.0) return "CR";
    if (grade >= 50.0) return "P";

    return "F";
}
```

Many other examples of the same algorithm or concept implemented in a variety of programming languages can be found at `http://rosettacode.org/`.

3.28.3 Advice for Learning a New Language

- Browse through lots of code examples
    - Try to understand how concepts you know from one language, Python, translate into the new language
- Look at the keywords (reserved words)
- Pick a project and try implement it in the new language
- Use resources like Stack Overflow, Rosetta Code, and others
- Use support/productivity tools (IDEs, revision control, sandboxing)
- Experiment