

While Loops

15-110 – Friday 01/31

Announcements

- Check2 due **Monday at noon**
- Check1/Hw1 revision deadline: **Tuesday 02/04 noon**
 - If you want to update your submission based on feedback, just make the changes to your solution and resubmit (applies to exercises too!)
 - TAs will regrade within a week, usually
 - Note that revision submissions are capped at **90 points** – don't resubmit if you already scored a 90 or above. (Still look at your feedback, though!)

Learning Goals

- Use **while loops** when reading and writing algorithms to repeat actions while a certain condition is met
- Identify **start values, continuing conditions, and update actions** for **loop control variables**
- Translate algorithms from **control flow charts** to Python code
- Use **nesting** of statements to create complex control flow

Repeating Actions is Annoying

Let's write a program that prints out the numbers from 1 to 10. Up to now, that would look like:

```
print(1)
print(2)
print(3)
print(4)
print(5)
print(6)
print(7)
print(8)
print(9)
print(10)
```


Loops Repeat Actions Automatically

A **loop** is a control structure that lets us repeat actions so that we don't need to write out similar code over and over again.

Loops are generally most powerful if we can find a **pattern** between the repeated items. Noticing patterns lets us separate out the parts of the action that are the same each time from the parts that are different.

In printing the numbers from 1 to 10, the part that is the **same** is the action of printing. The part that is **different** is the number that is printed.

same



```
print(1)
print(2)
print(3)
print(4)
print(5)
print(6)
print(7)
print(8)
print(9)
print(10)
```



different

While Loops

While Loops Repeat While a Condition is True

A **while loop** is a type of loop that keeps repeating only while a certain condition is met. It uses the syntax:

```
while <booleanExpression>:  
    <LoopBody>
```

The **while** loop checks the Boolean expression, and if it is **True**, it runs the loop body. Then it checks the Boolean expression again, and if it is still **True**, it runs the loop body again... etc.

When the **while** loop finds that the Boolean expression is **False**, it skips the loop body the same way an **if** statement would skip its body.

Conditions Must Eventually Become False

Unlike `if` statements, the condition in a `while` loop **must eventually become False**. If this doesn't happen, the `while` loop will keep going forever!

The best way to make the condition change from `True` to `False` is to use a variable as part of the Boolean expression. We can then change the variable inside the `while` loop. For example, the variable `i` changes in the loop below.

```
i = 1
while i < 5:
    print(i)
    i = i + 1
print("done")
```


Infinite Loops Run Forever

What happens if we don't ensure that the condition eventually becomes `False`? The `while` loop will just keep looping forever! This is called an **infinite loop**.

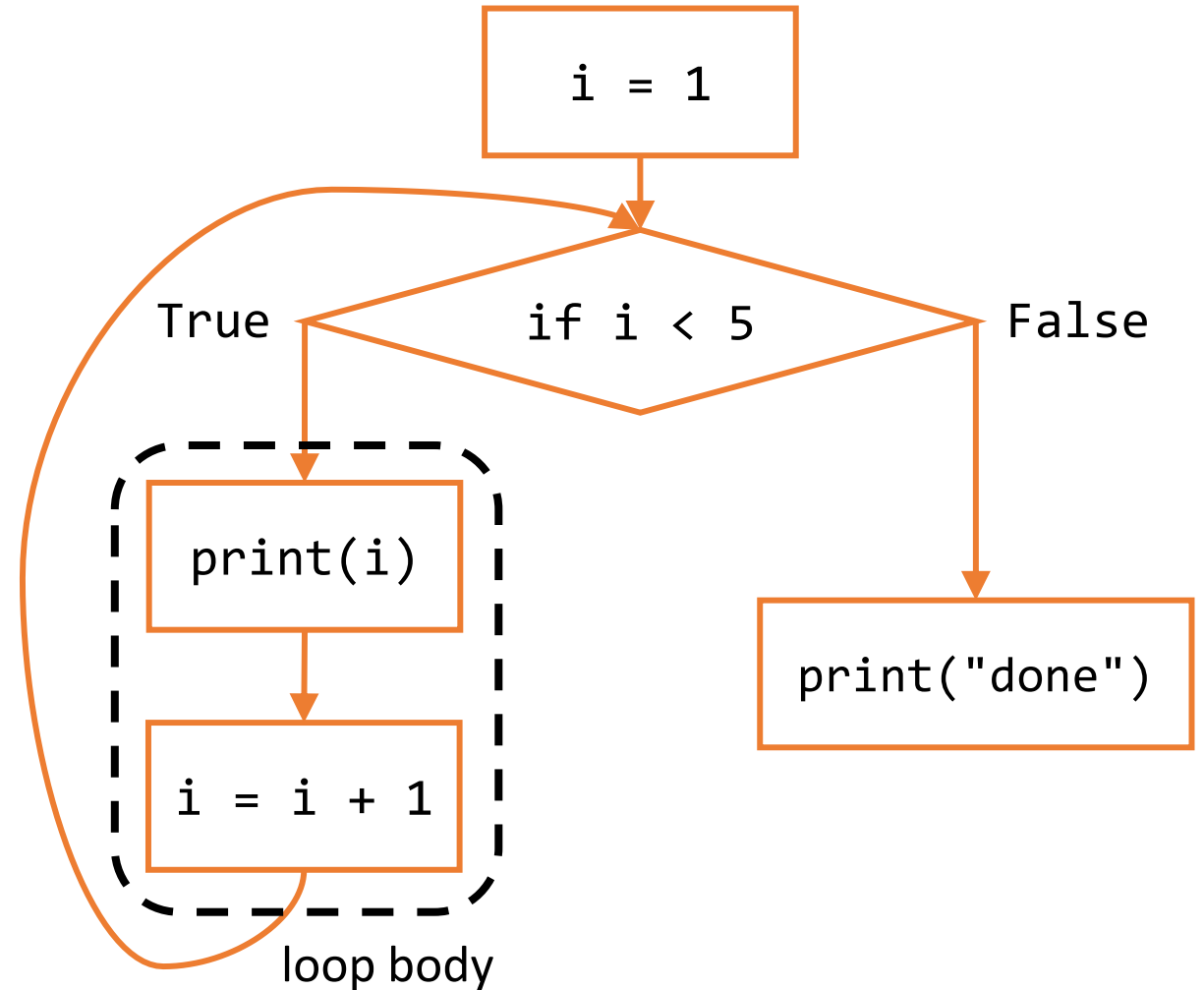
```
i = 1
while i > 0:
    print(i)
    i = i + 1
```

If you get stuck in an infinite loop, press the  button to make the program stop. Then investigate your program to figure out why the variable never makes the condition `False`. Printing out the variable that changes can help pinpoint the issue.

while Loop Flow Chart

Unlike an `if` statement, a `while` loop flow chart needs to include a transition from the `while` loop's body back to itself.

```
i = 1
while i < 5:
    print(i)
    i = i + 1
print("done")
```



You Do: Trace the Program

You do: if we slightly change the code from the previous program, what happens to the program?

```
i = 1
while i < 5:
    i = i + 1 # moved up one line
    print(i)
print("done")
```

Loop Control Variables

Use Loop Control Variables to Design Algorithms

Now that we know the basics of how loops work, we can start to write `while` loops that produce specific repeated actions.

First, we need to identify which parts of the repeated action must change in each iteration. This changing part will be created by the **loop control variable**, which is updated in the loop body.

To use this variable, we'll need to give it a **start value**, an **update action**, and a **continuing condition**. All three need to be coordinated for the loop to work correctly.

Loop Control Variables - Example

In our print 1-to-10 example, our variable is the number being printed. We want to **start** the variable at 1 and **continue while** the variable is less than or equal to 10. Set `num = 1` at the beginning of the loop and continue looping `while num <= 10`. The loop ends when `num` is 11.

Each printed number is one larger from the previous, so the **update** should set the variable to the next number (`num = num + 1`) in each iteration.

```
num = 1
while num <= 10:
    print(num)
    num = num + 1
```

Loop Control Variables – Counting Backwards

How would we change the program if we wanted to count backwards instead? The loop control variable is the same (the number being printed), but its components change.

Set `num = 10` at the beginning of the loop and continue looping `while num >= 1`. The loop ends when `num` is 0.

Each printed number is one smaller from the previous, so the **update** should set the variable to the next number (`num = num - 1`) in each iteration.

```
num = 10
while num >= 1:
    print(num)
    num = num - 1
```

Activity: Print Even Numbers

You do: your task is to print the even numbers from 2 to 100.

What is your loop control variable? What is its start value, continuing condition, and update action?

Once you've determined what these values are, use them to write a short program that does this task.

Loops in Algorithms

Implement Algorithms by Changing Loop Body

Suppose we want to add the numbers from 1 to 10.

We need to keep track of two different numbers:

- the current number we're adding
- the current sum

Both numbers need to be updated inside the loop body, but only one (the current number) needs to be checked in the condition.

```
result = 0
num = 1
while num <= 10:
    result = result + num
    num = num + 1
print(result)
```

Which is the loop control variable?

Tracing Loops

Sometimes it gets difficult to understand what a program is doing when that program uses loops. It can be helpful to manually trace through the values in the variables at each step of the code, including each iteration of the loop.

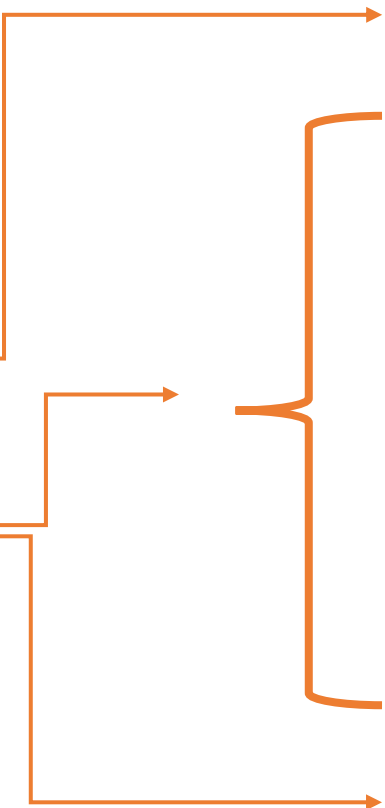
```
result = 0
num = 1
while num <= 7:
    result = result + num
    num = num + 1
print(result)
```

step	result	num
pre-loop	0	1
iteration 1	1	2
iteration 2	3	3
iteration 3	6	4
iteration 4	10	5
iteration 5	15	6
iteration 6	21	7
iteration 7	28	8
post-loop	28	8

Update Order

When updating multiple variables in a loop, **order matters**. If we update `num` before we update `result`, it changes the value held in `result`.

```
result = 0
num = 1
while num <= 7:
    num = num + 1
    result = result + num
print(result)
```



Note: Python checks the condition only at the start of the loop; it doesn't exit the loop as soon as `num` becomes 8.

step	result	num
pre-loop	0	1
iteration 1	2	2
iteration 2	5	3
iteration 3	9	4
iteration 4	14	5
iteration 5	20	6
iteration 6	27	7
iteration 7	35	8
post-loop	35	8

Nesting Conditionals in `while` Loops

We showed previously how we can nest conditionals in other conditionals or function definitions. We can do the same thing with `while` loops!

For example, let's make ascii art. Write code to produce the following printed string:

```
X-X-X
-O-O-
X-X-X
-O-O-
X-X-X
```

The loop will iterate over the rows that are printed. The program decides whether to print the x line or the o line based on **the value of the loop control variable**.

If it's even (0, 2, and 4) print x; if it's odd (1 and 3) print o.

```
row = 0
while row < 5:
    if row % 2 == 0:
        print("x-x-x")
    else:
        print("-o-o-")
    row = row + 1
```

Nesting `while` Loops in Functions

We can also nest loops inside of function definitions.

If we `return` inside a loop, Python **immediately** exits the function- no further iterations will run.

For example, if we want to check whether a multiple of `factor` occurs within a certain range `[start, end]`, we can return `True` as soon as we find a multiple (inside the loop), or `False` if we never find a multiple (outside the loop).

Normally you return in a conditional nested inside the loop, not the loop body itself. If you return directly in the loop, it will exit on the first iteration!

```
def multipleInRange(start, end, factor):  
    i = start  
    while i <= end:  
        print(i) # shows loop ends early  
        if i % factor == 0:  
            return True  
        i = i + 1  
    return False
```

Coding with Multiple Data Points

Now that we have loops, we can start writing algorithms to solve real-world problems. For example, we often want to analyze **multiple data points** while writing code.

Loops make it possible for us to repeat an action multiple times- that should make it possible for us to get multiple data points. But how can we receive that data?

For now, we'll use the `input` built-in function to repeatedly ask the user for data. Later we'll learn about a new data type that can store multiple values in one place.

Looping with input

If we call `input` inside the loop body, we can get multiple inputs from the user and process them like a data stream.

We'll need to give the user a way to signal that they're done entering numbers. This can be done with a special input, like the string `'q'`.

For example, this code sums the numbers entered by the user until they signal an end to the numbers.

```
result = 0
value = input("Enter a number, or q to quit:")
while value != "q":
    num = int(value)
    result = result + num
    value = input("Enter a number, or q to quit:")
print("Total sum:", result)
```

Note: our loop control variable here is `value`. It starts as a user input, is updated by asking for new input, and continues looping while it is not `"q"`.

Learning Goals

- Use **while loops** when reading and writing algorithms to repeat actions while a certain condition is met
- Identify **start values, continuing conditions, and update actions** for loop control variables
- Translate algorithms from **control flow charts** to Python code
- Use **nesting** of statements to create complex control flow

Extra Slides:

Advanced Loops in Algorithms

This content will not be tested, but is interesting to know!

Loop Control Variables – Advanced Example

It isn't always obvious how the start values, continuing conditions, and update actions of a loop control variable should work. Sometimes you need to think through an example to make it clear!

Example: simulate a zombie apocalypse. Every day, each zombie finds and bites a human, turning them into a zombie. If we start with just one zombie, how long does it take for the whole world (7.5 billion people) to turn into zombies?

We'll need to track and update **two** variables- one for the number of zombies, one for the number of days passed.

Loop control variable: # of zombies

Start value: 1 zombie

Continuing condition: while the number of zombies is less than the population

Update action: double the number of zombies every day

```
zombieCount = 1
population = 7.5 * 10**9
daysPassed = 0
while zombieCount < population:
    daysPassed = daysPassed + 1
    zombieCount = zombieCount * 2
print(daysPassed)
```

Loop Control Variables – Another Example

Example: how would you count the number of digits in an integer?

One answer: A number **abc** can be written as:

$$a*100 + b*10 + c*1$$

or

$$a*10^2 + b*10^1 + c*10^0$$

Check each power of 10 until one is bigger than the number. A separate variable can track the actual number of digits counted.

Loop control variable: which power of 10 is being checked

Start value: 1 (10^0)

Continuing condition: while the power of 10 isn't greater than the number

Update action: multiply the power by 10

```
num = 2021
power = 1
digits = 0
while power < num:
    digits = digits + 1
    power = power * 10
print(digits)
```

Loop Control Variables – Another Example

Another answer: instead of comparing a power of 10 to the number, change the number itself.

For example, to count the digits in **abc**, change:

abc ->

ab ->

a

The number of times you can **divide the number by 10** is the number of digits.

Loop control variable: the number itself

Start value: the number's initial value

Continuing condition: while the number is not yet 0 (no digits)

Update action: divide the number by 10

```
num = 2021
```

```
digits = 0
```

```
while num > 0:
```

```
    digits = digits + 1
```

```
    num = num // 10
```

```
print(digits)
```