How do you describe concurrent code?
await, **async**, asyncio.run(), asyncio.to_thread()
To run things concurrently: asyncio.gather()
How does concurrent code share state?
Until you call **await**, nothing will change
Mutable objects may be different after **await**

```
async def apollo():
    await asyncio.sleep(600)
async def kay():
    await doordash_boba()
async def harry():
    await cupcake()
async def start():
    await asyncio.gather(
        apollo(),
        kay(),
        harry(),
    )
asyncio.run(start())
```

- A coroutine function (may await!)
- Give up control until asyncio.sleep(600) finishes
- Run all of the awaitables (e.g., coroutines) concurrently
- Start an event loop (environment that knows how to handle concurrent work)

Exceptions are raised with a raise statement.

raise <expr>

<expr> must evaluate to a subclass of BaseException or an instance of one.

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
```

The <try suite> is executed first.

If, during the course of executing the <try suite>, an exception is raised that is not handled otherwise, and

```
>>> try:
        x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0

handling a <class 'ZeroDivisionError'>
>>> x
0
```

If the class of the exception inherits from <exception class>, then
The <except suite> is executed, with <name> bound to the exception.

(append s t): list the elements of s and t; append can be called on more than 2 lists

(map f s): call a procedure f on each element of a list s and list the results

(filter f s): call a procedure f on each element of a list s and list the elements for which a true value is the result

(apply f s): call a procedure f with the elements of a list as its arguments

```
(define size 5) ; => size
(* 2 size) ; => 10
(if (> size 0) size (- size)) ; => 5
(cond ((> size 0) size) ((= size 0) 0) (else (- size))) ; => 5
((lambda (x y) (+ x y size)) size (+ 1 2)) ; => 13
(map (lambda (x) (+ x size)) (quote (2 3 4))) ; => (7 8 9)
(filter odd? (quote (2 3 4))) ; => (3)
(list (cons 1 nil) size 'size) ; => ((1) 5 size)
(list (equal? 1 2) (null? nil) (= 3 4) (eq? 5 5)) ; => (#f #t #f #t)
(list (or #f #t) (or) (or 1 2)) ; => (#t #f 1)
(list (and #f #t) (and) (and 1 2)) ; => (#f #t 2)
(list 'a 2) ; => (a 2)
(append '(1 2) '(3 4)) ; => (1 2 3 4)
(not (> 1 2)) ; => #t
(begin (define x (+ size 1)) (* x 2)) ; => 12
```

```
(define (factorial n)
  (if (= n 0) 1
      (* n (factorial (- n 1)))))

(define (fib n)
  (cond
    ((= n 0) 0)
    ((= n 1) 1)
    (else (+ (fib (- n 2)) (fib (- n 1))))))

(define (nines num)
  (if (= num 0)
      0
      (if (= (modulo num 10) 9 )
          (+ 1 (nines (floor (/ num 10))))
          (nines (floor (/ num 10))))))
```

The way in which names are looked up in Scheme and Python is called *lexical scope* (or *static scope*).

Lexical scope: The parent of a frame is the environment in which a procedure was *defined*. (lambda ...)

Dynamic scope: The parent of a frame is the environment in which a procedure was *called*. (mu ...)

```
> (define f (mu (x) (+ x y)))
> (define g (lambda (x y) (f (+ x x))))
> (g 3 7)
13
```

A **table** has columns and rows

Latitude	Longitude	Name
38	122	Berkeley
42	71	Cambridge
45	93	Minneapolis

A **row** has a value for each column

A **column** has a name and a type

SELECT [expression] AS [name], [expression] AS [name], ... ;

SELECT [columns] FROM [table] WHERE [condition] ORDER BY [order];

Tables **A** & **B** are *joined* by **JOIN** (or a comma) to form combos of an **A** row & a **B** row
A join often has some conditions for matching up the rows of two (or more) tables

titles

tconst	title	year
tt8267604	Capernaum	2018
tt8367814	The Gentlemen	2019
tt8404614	The Two Popes	2019
tt8503618	Hamilton	2020

ratings

tconst	avgRating	numVotes
tt8503618	8.3	134421
tt8579674	8.2	749471
tt8613070	8.0	123438
tt8404614	7.6	144516

SELECT * **FROM** titles **JOIN** ratings **ON** titles.tconst=ratings.tconst;

tconst	title	year	tconst	averageRating	numVotes
tt8404614	The Two Popes	2019	tt8404614	7.6	144516
tt8503618	Hamilton	2020	tt8503618	8.3	134421

Explicit join syntax: Use **FROM** [table] **JOIN** [table] **ON** [condition]

```
SELECT title, averageRating FROM titles JOIN ratings
  ON titles.tconst=ratings.tconst LIMIT 3;
```

Implicit join syntax: Use a comma (or just **JOIN**) and put all conditions in the **WHERE** clause

```
SELECT title, averageRating FROM titles, ratings
  WHERE titles.tconst=ratings.tconst LIMIT 3;
```

Leaving out the **ON** or **WHERE** clause is allowed and creates all pairs of rows
A table can be joined by itself to compare one row to another row
Aliases are required

E.g., create a table of remakes: two movies that have the same title

```
SELECT old.title, old.year AS first, new.year AS second
  FROM titles AS old JOIN titles AS new
    ON old.title=new.title AND old.year < new.year;
```

```
CREATE TABLE lift AS
  SELECT 101 AS chair, 2 AS single, 2 AS pair UNION
  SELECT 102         , 0          , 3        UNION
  SELECT 103         , 4          , 1;
```

101
102
103

String values can be combined to form longer strings

```
sqlite> SELECT "hello," || " world";
hello, world
```

Basic string manipulation is built into SQL, but differs from Python

```
sqlite> CREATE TABLE phrase AS SELECT "hello, world" AS s;
sqlite> SELECT substr(s, 4, 2) || substr(s, instr(s, " ")+1, 1)
          FROM phrase;
low
```

The number of groups is the number of unique values of an expression
A having clause filters the set of groups that are aggregated

```
SELECT weight/legs, count(*) FROM animals
                    GROUP BY weight/legs
                    HAVING COUNT(*)>1;
```

kind	legs	weight
dog	4	20
cat	4	10
ferret	4	10
parrot	2	6
penguin	2	10
t-rex	2	12000

weight/legs=5
weight/legs=2
weight/legs=2
weight/legs=3
weight/legs=5
weight/legs=6000

weight/legs	count(*)
5	2
2	2

An aggregate function in the [columns] clause computes a value from a group of rows:

- MAX([expression]) evaluates to the largest value of [expression] for any row in a group
- COUNT(*) evaluates to the number of rows in a group
- MIN, SUM, & AVG are also aggregate functions similar to MAX

With no GROUP BY clause, aggregation is performed over all rows:

```
select max(legs) from animals;
```

max(legs)
4

Scheme programs consist of expressions, which can be:
- Primitive expressions: 2, 3.3, true, +, quotient, ...
- Combinations: (quotient 10 2), (not true), ...

Numbers are self-evaluating; *symbols* are bound to values.
Call expressions have an operator and 0 or more operands.

A combination that is not a call expression is a *special form*:
- **If** expression: (if <predicate> <consequent> <alternative>)
- Binding names: (define <name> <expression>)
- New procedures: (define (<name> <formal parameters>) <body>)

```
> (define pi 3.14)          > (define (abs x)
> (* pi 2)                      (if (< x 0)
6.28                                (- x)
                                    x))
                            > (abs -3)
                            3
```

Lambda expressions evaluate to anonymous procedures.

```
(lambda (<formal-parameters>) <body>)
```

λ

Two equivalent expressions:

```
(define (plus4 x) (+ x 4))
(define plus4 (lambda (x) (+ x 4)))
```

An operator can be a combination too:

```
((lambda (x y z) (+ x y (square z))) 1 2 3)
```

In the late 1950s, computer scientists used confusing names.
- **cons:** Two-argument procedure that **creates a Link**
- **car:** Procedure that returns the **first element** of a Link
- **cdr:** Procedure that returns the **second element** of a Link
- **nil:** The empty list

They also used a non-obvious notation for linked lists.
- A (linked) Scheme list has a first element and the rest, which may either be a Scheme list or nil (the empty Scheme list)
- Scheme lists are written as space-separated combinations.

```
> (define x (cons 1 nil))
> x
(1)
> (car x)
1
> (cdr x)
()
> (cons 1 (cons 2 (cons 3 (cons 4 nil))))
(1 2 3 4)
```

Symbols normally refer to values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of "a" and "b" in the resulting value

Quotation is used to refer to symbols directly in Lisp.

```
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

Symbols are now values

Quotation can also be applied to combinations to form lists.

```
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

```
(car (cons 1 nil)) -> 1
(cdr (cons 1 nil)) -> ()
(cdr (cons 1 (cons 2 nil))) -> (2)
```

The built-in Scheme list data structure (which is a linked list) can represent a Scheme expression

```
scm> (list 'quotient 10 2)      scm> (eval (list 'quotient 10 2))
(quotient 10 2)                 5
```

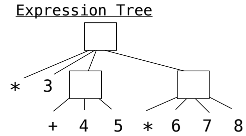
```
scm> (define (fact n)
       (if (= n 1) 1 (* n (fact (- n 1)))))
fact
scm> (fact 5)
120
scm> (define (fact-expr n)
       (if (= n 1) 1 (list '* n (fact-expr (- n 1)))))
fact-expr
scm> (fact-expr 5)
(* 5 (* 4 (* 3 (* 2 1))))
scm> (define (fact-expr-quasiquoted n)
       (if (= n 1) 1 `(* ,n ,(fact-expr-quasiquoted (- n 1)))))
fact-expr-quasiquoted
scm> (fact-expr-quasiquoted 5)
(* 5 (* 4 (* 3 (* 2 1))))
```

The Calculator language has primitive expressions and call expressions

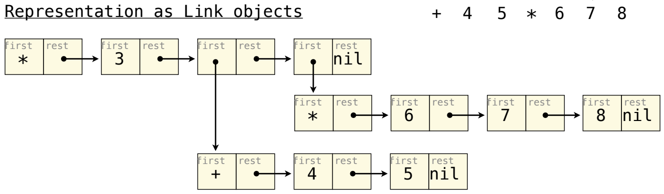
Calculator Expression

```
(* 3
   (+ 4 5)
   (* 6 7 8))
```

Expression Tree



Representation as Link objects



A Scheme list is written as elements in parentheses:

($<element_0>$ $<element_1>$ ... $<element_n>$)   A Scheme list

Each <element> can be a combination or atom (primitive).

(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))

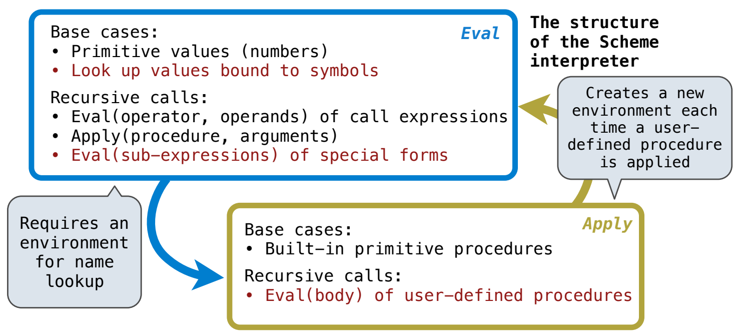
```
def reduce(f, s, initial):
    """Combine elements of s pairwise using f, starting with initial.
    E.g., reduce(mul, [2, 4, 8], 1) is equivalent to
          mul(mul(mul(1, 2), 4), 8).

    >>> reduce(mul, [2, 4, 8], 1)
    64
    """
    for x in s:
        initial = f(initial, x)
    return initial
```

f is ...
  a two-argument function that returns a first argument
s is ...
  a sequence of values that can be the second argument
initial is ...
  a value that can be the first argument

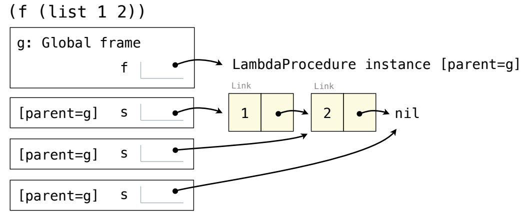
The Scheme version of reduce doesn't have an initial argument:

```
scm> (reduce * (list 2 4 8))
64
```



To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the **env** of the procedure, then evaluate the body of the procedure in the environment that starts with this new frame.

```
(define (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))
(f (list 1 2))
```



There are two ways to quote an expression

```
  Quote:       '(a b)    =>   (a b)
  Quasiquote:  `(a b)    =>   (a b)
```

Parts of a quasiquoted expression can be unquoted with ,

```
               (define b 4)
  Quote:       '(a ,(+ b 1))  =>   (a (unquote (+ b 1)))
  Quasiquote:  `(a ,(+ b 1))  =>   (a 5)
```

Quasiquotation is convenient for generating Scheme expressions:

```
               (define (make-add-lambda n) `(lambda (d) (+ d ,n)))
               (make-add-lambda 2)  => (lambda (d) (+ d 2))
```

A macro is an operation performed on the source code of a program before evaluation
Macros exist in many languages, but are easiest to define correctly in a language like Lisp
Scheme has a **define-macro** special form that defines a source code transformation

Constructs & evaluates the expression: (begin (print 2) (print 2))

```
(define-macro (twice expr)        > (twice (print 2))
  (list 'begin expr expr))        2
                                  2
```

Evaluation procedure of a macro call expression:
- Evaluate the operator sub-expression (evaluates to a macro)
- Call the macro procedure on the operand expressions ***without evaluating them first***
- Evaluate the expression returned from the macro procedure