Math 121

Python can do math!

```
>>>3 + 7
10
>>>4 + 2 * 3
10
>>>(4 + 2) * 3
18
>>>4 / 16
.25
>>>.1 + .2
0.30000000000000004
```

Python has two types of numbers

- Integers: computation is exact
- Floating point numbers ("floats"): computation is approximate
  - Division outputs floats

```
>>>10 / 2
5.0
>>>3 + 4.0
7.0
>>>int(8.7)
8
```

Python has two more operators

- `//` is "integer division", division with integer output

```
>>>10 / 2
5.0
>>>10 // 2
5
>>>14 // 3
4
```

Integer division drops the "remainder" from the answer.

Python has two more operators

- `%` is "mod", gives the remainder of integer division

```
>>>10 % 2
0
>>>10 % 3
1
>>>14 % 3
2
>>>5 % 12
5
```

Experiment with `//` and `%` using negative numbers.

Python has some built-in functions to help as well

- List is linked from the class website

```
>>>pow(2,3)
8
>>>abs(-3)
3
>>>abs(4 + 2)
6
>>>min(3,7)
3
>>>max(4, 8.3, 6)
8.3
```

You can save values to variables

```
>>>x=7
>>>x
7
>>>x=8+5
>>>x
13
>>>new=x
>>>new
13
>>>x=y
>>>y
Error
```

```
>>>x = 3
>>>x = x + 1
>>>x
4
>>>y = 10
>>>x = y
>>>y = x
>>>x
10
>>>y
10
```

```
>>>x, y = 3, 10
>>>x
3
>>>y
10
>>>x, y = y, x
>>>x
10
>>>y
3
```

You can import more functions from libraries

```
>>>from math import sqrt
>>>sqrt(16)
4.0
>>>sqrt(2)
1.4142135623730951
>>>from operator import add, sub, mul
>>>add(15, 8)
23
>>>add(4, mul(7, sub(8, 3)))
39
```

```
>>>from math import sqrt as root
>>>root(16)
4.0
>>>root(2)
1.4142135623730951
>>>from math import *
>>>sqrt(16)
4.0
>>>log(4, 10)
0.6020599913279623
```

This can get risky.

Often better:

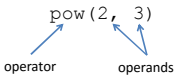
```
>>>import math
>>>math.sqrt(16)
4.0
>>>math.log(4, 10)
0.6020599913279623
```

Functions are just another type of value

- You can assign them to variables

```
>>>x = pow
>>>x(3, 2)
9
>>> x
<built-in function pow>
```

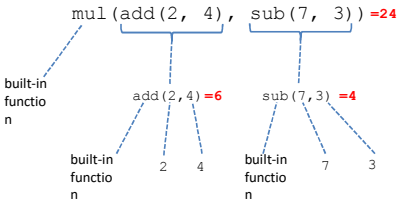
Let's get more formal. This is a "call expression":



Rules for evaluating:

1. Evaluate the operator and the operands
2. Apply the function that is the value of the operator to the values of the operands

This process is *recursive*.




```
>>>x = 7
>>>x(3, 2)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    x(3,2)
TypeError: 'int' object is not callable
```

Expressions: things that have a value

- `4`
- `x`
- `pow(3,4)`

Statements: things that do something

- `x = 3`
- `import math`
- `print(7)`

Interactive IDLE and a .py file work differently

- IDLE runs the line of code *and then prints the resulting value*
- Running a .py file just evaluates the lines
  - Only does something with the resulting values if you tell it to

You can write your own functions

- Recommend doing this in files, not in IDLE
- Set editor to use spaces as tabs! (usually 4)

```
def square(x):
    return x * x
```

Compound interest:

$$P = C(1+r/n)^{nt}$$

P = future value
C = initial deposit
r = annual interest rate, expressed as a decimal
n = number of times per year interest is compounded
t = number of years that pass

Lessons:
- Use useful variable names
- Use comments

Pure functions
- Output a value
- Always have the same value on the same input
- Have no other effects
- ex., `abs()`, `pow()`

Non-pure functions
- All other functions
- ex., `print()`, `randint()`

Why do we write functions?
- Saves time – no repetition
- More reliable
  - Fewer chances to make a mistake
  - Can be carefully tested
- Abstraction
  - Can think about several simple tasks, rather than one enormous one
  - Can be used without knowing how it works
  - Can change how it works

Strings

```
>>>x = "Hello"
>>>x
'Hello'
>>>y = 'world'
>>>y
'world'
>>>x+y
'Helloworld'
>>> z = input('What\'s your name?')
What's your name?Adam
>>> x+" "+z
'Hello Adam'
```

One special value: `None`
- "means" that something has no value, but it's a value

```
>>>print(8)
8
>>>print(print(7))
7
None
>>>x = print(3)
3
>>>x
>>>
```

Booleans
- Special type with only two values, `True` and `False`

```
>>>7 > 3
True
>>>4 < 2
False
>>>4 > 4
False
>>>4 >= 4
True
>>>3 <= 8
True
```

Booleans

- Special type with only two values, `True` and `False`

```
>>>7 == 3
False
>>>4 == 4
True
>>>3.0 == 3
True
>>>4 != 4
False
>>>3 != 8
True
```

You can combine these with `and`, `or`, and `not`.

```
>>>7 > 3 and 2 < 4
True
>>>4 < 2 and False
False
>>> 2 > 3 or (not 7 < 10)
False
```

Conditional statements: use the `if` clause

```
def print_positive(x):
    if x > 0:
        print(x)
```

Conditional statements: use the `if` clause

- `else` clause for when the condition isn't true

```
def my_max(x,y):
    if x > y:
        return x
    else:
        return y
```

Conditional statements: use the `if` clause

- `else` clause for when the condition isn't true
- `elif` (short for "else if") for additional cases

```
def my_max(x,y,z):
    if x > y and x > z:
        return x
    elif y > z:
        return y
    else:
        return z
```

Recreate absolute value?

Can output booleans

```
def above_ten(x):
    if x > 10:
        return True
    else:
        return False

if above_ten(12):
    print(12)
if above_ten(9):
    print(9)
```

A simpler option:

```
def above_ten(x):
    if x > 10:
        return True
    else:
        return False
```

Replace with:

```
def above_ten(x):
    return x > 10
```

```
if 7:
    print(7)
if 0:
    print(0)
if True:
    print(True)
if print("x"):
    print("print")
```

Result:

```
7
True
x
```

```
>>>7 and True
True
>>>True and 7
7
>>>0 and 7
0
>>>print(8) and 0
8
>>>0 and print(8)
0
```

To evaluate `<left> and <right>`:

1. Evaluate `<left>`
2. If the value is false, output that value
3. Otherwise, output the value of `<right>`

To evaluate `<left> or <right>`:

1. Evaluate `<left>`
2. If the value is true, output that value
3. Otherwise, output the value of `<right>`

To evaluate not `<exp>`:

1. Evalute `<exp>`
2. If the value is true, output `True`, and `False` otherwise

False values:

- `False`
- `0`
- `None`
- `""`

True values:

- `True`
- Numbers other than 0
- Other strings

```
>>>7 or True
7
>>>True or 7
True
>>>False or 7
7
>>>print(8) or 0
8
0
>>>0 or print(8)
8
>>>print(8) or 1
8
1
```

Another way to control the flow of a program: `while`

```
x = input("What is 3+4? ")
while x != "7":
    x = input("Sorry, try again: ")
print("Correct!")
```

Result:

```
What is 3+4? 5
Sorry try again: 3
Sorry try again: 2
Sorry try again: 42
Sorry try again: 7
Correct!
```

Another way to control the flow of a program: `while`

```
x = input("What is 3+4? ")
while x != "7":
    x = input("Sorry, try again: ")
print("Correct!")
```

Rules for execution:

1. Evaluate the expression in the header
2. If the expression is true, execute the body of the statement, and then return to step 1. (Otherwise, do nothing and move on.)

Another example

```
i, total = 1, 0
while i < 4:
    total = total + i
    i = i + 1
print(total)
```

Result:

```
6
```

Another example

```
i, total = 1, 0
while i < 4:
    i = i + 1
    total = total + i
print(total)
```

Result:

```
9
```

Note: Condition check only happens at the start of each cycle

Another example

```
i, total = 1, 0
while i <= 5:
    j=1
    while j <= 5:
        total = total + 1
        j = j + 1
    i = i + 1
print(total)
```

Result:

```
25
```

Another example

```
i, total = 1, 0
while i <= 5:
    j=1
    while j <= i:
        total = total + 1
        j = j + 1
    i = i + 1
print(total)
```

Result:
15

To see what's going on:

```
i, total = 1, 0
while i <= 5:
    j=1
    while j <= i:
        total = total + 1
        print(i, j)
        j = j + 1
    i = i + 1
print(total)
```

Result:
15

Exercise: If I'm adding 1+2+3+… how far do I need to go to get a total greater than 1000?

```
i, total = 0, 0
while total <= 1000:
    i = i + 1
    total = total + i
print(i)
```

Result:
45

Exercise: Given the polynomial x^4-8x^3+6x-4, what integer value of x between 0 and 10 results in the lowest value?

```
def poly(x):
    return x**4 - 8*x**3 + 6*x - 4

indexOfMin, min = 0, poly(0)
i=1
while i <= 10:
    if poly(i) < min:
        indexOfMin, min = i, poly(i)
    i = i + 1
print(indexOfMin)
```

What will happen?

```
x = 3
def square(x):
    return x*x
print(square(4))
print(x)
```

Result:
16
3

What will happen?

```
x = 3
y = 4
def square(x):
    y = 7
    return x*x
print(square(5))
print(y)
```

Result:
25
4

What will happen?

```
x = 3
def addto(y):
    return y + x
x = 4
print(addto(7))
```

Result:

```
11
```

Environments and frames

Global	
x	3

```
x = 3
y = 7
x = 10
def addto(y):
    return y + x
z = addto(4)
```

Environments and frames

Global	
x	3
y	7

```
x = 3
y = 7
x = 10
def addto(y):
    return y + x
z = addto(4)
```

Environments and frames

Global	
x	10
y	7

```
x = 3
y = 7
x = 10
def addto(y):
    return y + x
z = addto(4)
```

Environments and frames

Global	
x	10
y	7
addto	<function at __>

```
x = 3
y = 7
x = 10
def addto(y):
    return y + x
z = addto(4)
```

Environments and frames

Global	
x	10
y	7
addto	<function at __>

addto	
y	4

```
x = 3
y = 7
x = 10
def addto(y):
    return y + x
z = addto(4)
```

Environments and frames

```
x = 3
y = 7
x = 10
def addto(y):
    return y + x
z = addto(4)
```

Global	
x	10
y	7
addto	<function at __>

addto	
y	4

x?

Environments and frames

```
x = 3
y = 7
x = 10
def addto(y):
    return y + x
z = addto(4)
```

Global	
x	10
y	7
addto	<function at __>

addto	
y	4

x?

Environments and frames

```
x = 3
y = 7
x = 10
def addto(y):
    return y + x
z = addto(4)
```

Global	
x	10
y	7
addto	<function at __>
z	14

addto	
y	4

Revisit:

```
x = 3
def square(x):
    return x*x
print(square(4))
print(x)
```

Result:
16
3

Revisit:

```
x = 3
y = 4
def square(x):
    y = 7
    return x*x
print(square(5))
print(y)
```

Result:
25
4

Revisit:

```
x = 3
def addto(y):
    return y + x
x = 4
print(addto(7))
```

Result:
11

Return to this:

Exercise: Given the polynomial x^4-8x^3+6x-4, what integer value of x between 0 and 10 results in the lowest value?

```
def poly(x):
    return x**4 - 8*x**3 + 6*x - 4

indexOfMin, min = 0, poly(0)
i=1
while i <= 10:
    if poly(i) < min:
        indexOfMin, min = i, poly(i)
    i = i + 1
print(indexOfMin)
```

Say we want to handle any range.

```
def argmin(rangemin, rangemax):
    def poly(x):
        return x**4 - 8*x**3 + 6*x - 4
    indexOfMin, min = rangemin, poly(rangemin)
    i=rangemin+1
    while i <= rangemax:
        if poly(i) < min:
            indexOfMin, min = i, poly(i)
        i = i + 1
    return indexOfMin
```

Definition inside a function definition?

Say we want to handle any range.

```
def argmin(rangemin, rangemax):
    def poly(x):
        return x**4 - 8*x**3 + 6*x - 4
    indexOfMin, min = rangemin, poly(rangemin)
    i=rangemin+1
    while i <= rangemax:
        if poly(i) < min:
            indexOfMin, min = i, poly(i)
        i = i + 1
    return indexOfMin
```

Definition inside a function definition?

```
argmin(1, 3)
```

Handle other functions?

```
def argmin(rangemin, rangemax, func):
    indexOfMin, min = rangemin, func(rangemin)
    i=rangemin+1
    while i <= rangemax:
        if func(i) < min:
            indexOfMin, min = i, func(i)
        i = i + 1
    return indexOfMin

def poly(x):
    return x**4 - 8*x**3 + 6*x - 4

argmin(1, 3, poly)
```

```
def argmin(rangemin, rangemax, func):
    indexOfMin, min = rangemin, func(rangemin)
    i=rangemin+1
    while i <= rangemax:
        if func(i) < min:
            indexOfMin, min = i, func(i)
        i = i + 1
    return indexOfMin

def poly(x):
    return x**4 - 8*x**3 + 6*x - 4
```

What if we wanted a function that always minimized `poly`?

```
def argmin(rangemin, rangemax, func):
    indexOfMin, min = rangemin, func(rangemin)
    i=rangemin+1
    while i <= rangemax:
        if func(i) < min:
            indexOfMin, min = i, func(i)
        i = i + 1
    return indexOfMin

def poly(x):
    return x**4 - 8*x**3 + 6*x - 4

def buildMinimizer(func):
    def minimizer(rangemin, rangemax):
        return argmin(rangemin, rangemax, func)
    return minimizer

polymin = buildMinimizer(poly)

z = polymin(1, 10)
```

Rule for local frames: Parent frame is the frame in which the function was *defined*

- Environment: chain of frames
- Look for variables by moving up the chain

```
x = 4
def mult_const(x):
    def newfunc(y):
        return y*x
    return newfunc
f = mult_const(3)
print(f(2))
```

Resut:
6

```
def skipped(f):
    def g():
        return f
    return g

def composed(f,g):
    def h(x):
        return f(g(x))
    return h

def added(f, g):
    def h(x):
        return f(x) + g(x)
    return h

def square(x):
    return x*x

def two(x):
    return 2
```

```
a = composed(square, two)(7)

b = composed(two, square)(2)

c = skipped(added(square, two))()(3)
```

What are a, b, and c?

a = 4

b = 2

c = 11

Draw the environment diagram:

```
from operator import add
def curry2(h):
    def f(x):
        def g(y):
            return h(x,y)
        return g
    return f
make_adder = curry2(add)
add_three = make_adder(3)
five = add_three(2)
z = make_adder(1)(6)
```

Functions without names

```
s = lambda x: x*x
a = s(8)
print(a)
```

lambda	x	:	x*x
A function that	takes x	and returns	x*x

Functions without names

```
s = lambda x: x*x
```

is equivalent to

```
def square(x):
    return x*x
s = square
```

Why is this useful?

What happens?

```
a = lambda f: f(x+1)
x = 5
b = a(lambda x: x+1)
print(b)
```

Result:
7

What happens?

```
foo = lambda a: (lambda x: x+a)
b = foo(3)(8)
print(b)
```

Result:

```
11
```

What happens?

```
a = 4
b = lambda x, y: lambda z: z(x) + z(y)
c = lambda x: x+a
d = b(3,7)
print(d)
```

Result:

```
<function <lambda>.<locals>.<lambda> at
0x100732c80>
```

What happens?

```
a = 4
b = lambda x, y: lambda z: z(x) + z(y)
c = lambda x: x+a
d = b(3,7)
e = d(c)
print(e)
```

Result:

```
18
```

Back to Fibonacci numbers

Formal definition: $f_1 = 1$, $f_2=1$, $f_n = f_{n-1} + f_{n-2}$

Just write this as code

```
def fib(n):
    if n==1 or n==2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
print(fib(6))
```

Result:

```
8
```

Summing 1 + 2 + … + n (again)

```
def sumto(n):
    if n == 1:
        return 1
    else:
        return n + sumto(n-1)
sumto(5)
```

Result:

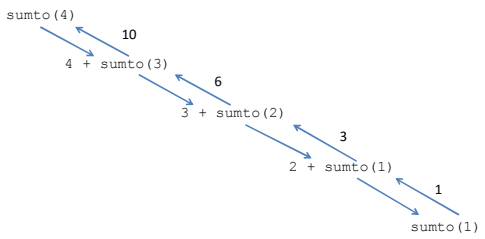
```
15
```

```
def sumto(n):
    if n == 1:
        return 1
    else:
        return n + sumto(n-1)
```

Base case

Be careful to avoid infinite loops!

How does evaluation work?



Iterative vs. recursive

```
def sumto(n):
    if 1 == 0:
        return 1
    else:
        return n + sumto(n-1)
```

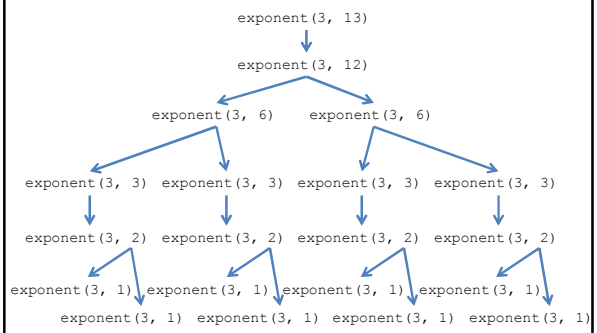
```
def sumto(n):
    i = 1
    total = 0
    while i <= 0:
        total = total + i
        i = i + 1
    return total
```

```
def mystery(x, y):
    if y == 0:
        answer = 1
    else:
        answer = x * mystery(x, y - 1)
    return answer
```

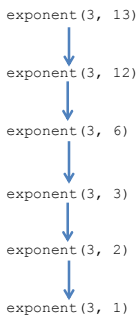
```
def exponent(base, power):
    if power == 0:
        answer = 1
    else:
        answer = base * exponent(base, power - 1)
    return answer
```

```
exponent(3, 13)
↓
exponent(3, 12)
↓
exponent(3, 11)
↓
exponent(3, 10)
↓
exponent(3, 9)
↓
⋮
↓
exponent(3, 1)
```

```
def exponent(base, power):
    if power == 0:
        answer = 1
    elif power % 2 == 1:
        answer = base * exponent(base, power - 1)
    elif power & 2 == 0:
        answer = exponent(base, power / 2)
                    * exponent(base, power / 2)
    return answer
```




```
def exponent(base, power):
    if power == 0:
        answer = 1
    elif power % 2 == 1:
        answer = base * exponent(base, power - 1)
    elif power % 2 == 0:
        squareroot = exponent(base, power / 2)
        answer = squareroot * squareroot
    return answer
```



Partitions: How many ways to write n as a sum?

5 = 5
5 = 4 + 1
5 = 3 + 2
5 = 3 + 1 + 1
5 = 2 + 2 + 1
5 = 2 + 1 + 1 + 1
5 = 1 + 1 + 1 + 1 + 1

`partitions(5) = 7`

How do we write it?

Sometimes easier to do something more general

`part_under(n, m)` = number of partitions of n using no numbers bigger than m

Important observation: Any such partition either uses m or doesn't

Number of partitions of n using pieces of up to size m and no pieces of size m
= number of partitions of n using pieces of up to size m-1

Number of partitions of n using pieces of up to size m and a piece of size m
= number of partitions of n-m using pieces of up to size m

`part_under(n,m)==part_under(n,m-1)+part_under(n-m,m)`

Make it into a function definition

- need a base case

```
def part_under(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        return part_under(n, m-1) + part_under(n-m, m)

def partition(n):
    return part_under(n, n)
```

```
def sudokuPossible(sudoku, currentBox):
    if currentBox == lastBox:
        return True
    try = 1
    nextBox = next(currentBox)
    while try <= 9:
        newSudoku = addToBox(sudoku, nextBox, try)
        if checkForErrors(newSudoku) and
                            sudokuPossible(newSudoku, nextBox):
            return True
        try = try + 1
    return False
```

Lists

- New data type
- Variable length

```
>>>a = [1, 3, 5]
>>>a
[1, 3, 5]
>>>a[1]
3
>>>a[0]
1
>>>a + a
[1, 3, 5, 1, 3, 5]
```

```
>>>b = [2, 1]
>>>c = b * 3
>>>c
[2, 1, 2, 1, 2, 1]
>>>len(c)
6
>>>d = [a, b]
>>>d
[[1, 3, 5], [2, 1]]
>>>len(d)
2
>>>d[1][0]
2
```

```
>>>b = [2, 1]
>>>b[1] = 3
>>>b
[2, 3]
>>>a = b
>>>a
[2, 3]
>>>a[0] = 7
>>>a
[7, 3]
>>>b
[7, 3]
```

```
>>>a = [1, 3, 5, 7, 9, 11, 13]
>>>a[-2]
11
>>>a[2:5]
[5, 7, 9]
>>>a[2:-2]
[5, 7, 9]
>>>a[3:]
[7, 9, 11, 13]
>>>a[:3]
[1, 3, 5]
>>>a[10] = 21
error
```

```
>>>b = [2, 1]
>>>b.append(3)
>>>b
[2, 1, 3]
>>>b.append([4, 5])
>>>b
[2, 1, 3, [4, 5]]
>>>c = b.append(2)
>>>c
>>>print(b.append(4))
None
>>>b
[2, 1, 3, [4, 5], 2, 4]
```

```
>>>b = [2, 1]
>>>b.append([4, 5])
>>>b
[2, 1, [4, 5]]
>>>b = [2, 1]
>>>b.extend([4, 5])
>>>b
[2, 1, 4, 5]
>>>b.insert(1, 3)
[2, 3, 1, 4, 5]
>>>b.index(3)
1
```

```
>>>b = b * 2
>>>b
[2, 3, 1, 4, 5, 2, 3, 1, 4, 5]
>>>b.index(3)
1
```

```
>>>b = [2, 1, 4]
>>>2 in b
True
>>>[1, 3] in b
False
>>>[1, 4] in b
False
```

```
>>>b = [2, 1, 4]
>>>c = b
>>>a = [2, 1, 4]
>>>c == b
True
>>>a == b
True
>>>c is b
True
>>>a is b
False
```

Create a list of first n squares, starting at 1

```
def squares(n):
    ls = []
    i = 1
    while i <= n:
        ls.append(i*i)
        i += 1
    return ls
```

Sum of a list:

```
def listSum(ls):
    total = 0
    index = 0
    while index < len(ls):
        total += ls[index]
        index += 1
    return total
```

That pattern is so common that Python has a shortcut:

```
def listSum(ls):
    total = 0
    for item in ls:
        total += item
    return total
```

`for` loops, executing the body once for each item in the list.

Update all elements of a list using some function

- Ex., square every element in the list

```
def listMap(func, ls):
    newList = []
    for item in ls:
        newList.append(func(item))
    return newList

a = [1, 3, 2, 4]
b = listMap(lambda x: x*x, a)
print(b)

[1, 9, 4, 16]
```

Converting a list of temperatures from Celsius to Farenheit

```
def CtoF(temp):
    return temp*1.8 + 32

Ftemps = listMap(CtoF, Ctemps)
```

Be careful of the built-in map function

This pattern is also very common

```
def listSum(ls):
    total = 0
    for item in ls:
        total += item
    return total
```

General idea: start "total" at some value, update it repeatedly, once with each element of the list.

General idea: start "total" at some value, update it repeatedly, once with each element of the list.

```
def listReduce(func, init, ls):
    result = init
    for item in ls:
        result = func(result, item)
    return result

a = [1, 5, 7]
from operator import add
b = listReduce(add, 0, a)
print(b)

13
```

Other useful list operation, filter, is in your homework.

Python also has "list comprehensions," which are abbreviations for map and filter.

```
>>>a = [1, 3, 5, 7]
>>>[x+1 for x in a]
[2, 4, 6, 8]
>>>[2*x for x in a if 25 % x == 0]
[2, 10]
```

One more data type: dictionaries

```
>>>dict = {"Bob": 7, "Alice": 10, "Carl": 4}
>>>dict
{"Alice": 10, "Bob": 7, "Carl": 4}
>>>dict['Alice']
10
>>>dict["Dave"] = 12
>>>dict
{'Dave': 12, 'Alice': 10, 'Bob': 7, 'Carl': 4}
>>>dict["Alice"] = 15
>>>dict
{'Alice': 15, 'Dave': 12, 'Bob': 7, 'Carl': 4}
```

Dictionaries

- Unordered
- Only one pair with each key (new pairs replace old ones)
- Keys must be immutable
  - Numbers
  - Strings
- Values can be anything
- Mutable, like lists

Data abstractions are useful

- Certain structures work better for certain tasks
- General advantages of abstraction
  - Modular design
  - Less repeated work
  - Easier error-checking
  - Easier-to-understand code

Python only has a couple very common types built in.

Let's make our own.

Example: rational numbers

- Start with the most basic operations

Make a new one.

```
def rational(n, d):
    return [n, d]
```

Get the numerator.

```
def numer(r):
    return r[0]
```

Get the denominator.

```
def denom (r):
    return r[1]
```

Now for slightly higher-level operations

Multiplying

```
def mul_rats(r, s):
    newNum = numer(r) * numer(s)
    newDen = denom(r) * denom(s)
    return rational(newNum, newDen)
```

Now for slightly higher-level operations

Adding

```
def add_rats(r, s):
    nr = numer(r)
    dr = denom(r)
    ns = numer(s)
    ds = denom(s)
    newNum = nr * ds + ns * dr
    newDen = ds * dr
    return rational(newNum, newDen)
```

Now for slightly higher-level operations

Checking equality

```
def is_equal_rats(r, s):
   return numer(r)*denom(s)==numer(s)*denom(r)
```

Now for slightly higher-level operations

Printing

```
def print_rats(r):
    print(numer(r), "/", denom(r))
```

Other basic operations:

- Subtraction
- Division

We can now use these.

```
a = rational(1, 3)
b = rational(1, 2)
c = add_rats(a, mul_rats(b, a))
print_rats(c)
```

Result:

```
9 / 18
```

Layers of abstraction

Parts of the program that….	Treat rationals as…	By using…
Use rational numbers for larger purpose	Single data values	`add_rats`, `mul_rats`, `print_rats`, `is_equal_rats`
Implement operations on rationals	Numerators and denominators	`rational`, `numer`, `denom`
Implement selectors and constructors	Two-element lists	list operations

We can change things

- ex., if we want to keep fractions in lowest terms

We change this…

```
def rational(n, d):
    return [n, d]
```

…to this.

```
def rational(n, d):
    g = gcd(n, d)
    return [n//g, d//g]
```

We can change the basic implementation

We change this…

```
def rational(n, d):
    return [n, d]
```

…to this.

```
def rational(n, d):
    define frac(x)
        if x == 0:
            return n
        else:
            return d
    return frac
```

We can change the basic implementation

We change this…

```
def numer(r):
    return r[0]
```

…to this.

```
def numer(r):
    return r(0)
```

We can change the basic implementation

We change this…

```
def denom(r):
    return r[1]
```

…to this.

```
def denom(r):
    return r(1)
```

And everything works as before.

Another example: Point on the earth's surface

Constructor

- Takes what format?
  - Multiple options?
- Stores in what format?

Selectors

- Latitude, longitude
  - In what format?
- Direction?

Another example: Point on the earth's surface

Methods

- Move a point in some direction
- Check if two points are equal
- Distance between two points
  - What degree of accuracy?

Data can have local state

- Lists
- Dictionaries

We can make functions have local state too

If we want this:

```
>>>a = newGiftCard(100)
>>>a(20)
80
>>>a(45)
35
>>>a(50)
"Insufficient funds"
>>>a(20)
15
```

Create a gift card

```
def newGiftCard(balance):
    def spend(amount)
        nonlocal balance
        if amount > balance:
            return "Insufficient funds"
        balance = balance - amount
        return balance
    return spend
```

What if we also want to be able to add money to the card?

```
def newGiftCard(balance):
    def spend(amount, task):
        nonlocal balance
        if task == "spend":
            if amount > balance:
                return "Insufficient funds"
            balance = balance - amount
            return balance
        elif task == "add"
            balance = balance + amount
            return balance
    return spend
```

Now works like this:

```
>>>a = newGiftCard(100)
>>>a(20, "spend")
80
>>>a(45, "spend")
35
>>>a(50, "spend")
"Insufficient funds"
>>>a(20, "add")
55
>>>a(50, "spend")
5
```

This methodology is "object-oriented programming"

- Python has built-in tools for this

```
class GiftCard:
    def __init__(self, amount):
        self.balance = amount
    def addMoney(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def spend(self, amount):
        if amount > self.balance:
            return "Insufficient funds"
        self.balance = self.balance - amount
        return self.balance
```

Now works like this:

```
>>>a = GiftCard(100)
>>>a.spend(20)
80
>>>a.spend(45)
35
>>>a.spend(50)
"Insufficient funds"
>>>a.addMoney(20)
55
>>>a.spend(50)
5
```

```
class Account:
    intRate = .02
    def __init__(self, amount):
        self.balance = amount
    def deposit(self, amount):
        self.balance += amount
    def payInterest(self):
        self.balance *= 1 + intRate
```

```
class Account:
    intRate = .02
    def __init__(self, amount):
        self.balance = amount
    def deposit(self, amount):
        self.balance += amount
    def payInterest(self):
        self.balance *= 1 + intRate
```

```
>>>a = Account(100)
```

Rules for evaluating

- Class used as function
  - Create new object of that class
  - run `__init__` with the new object as the first argument (`self`) and the other arguments the same

```
class Account:
    intRate = .02
    def __init__(self, amount):
        self.balance = amount
    def deposit(self, amount):
        self.balance += amount
    def payInterest(self):
        self.balance *= 1 + intRate
```

```
>>>a = Account(100)
>>>a.balance
100
```

Rules for evaluating

- `<object>.<variable>`
  - Look for the value of that variable associated with that object

```
class Account:
    intRate = .02
    def __init__(self, amount):
        self.balance = amount
    def deposit(self, amount):
        self.balance += amount
    def payInterest(self):
        self.balance *= 1 + intRate
```

```
>>>a = Account(100)
>>>a.balance
100
>>>a.intRate
.02
```

Rules for evaluating

- `<object>.<variable>`
  - Look for the value of that variable associated with that object
  - If not found, look for a value of that variable associated with the class

```
class Account:
    intRate = .02
    def __init__(self, amount):
        self.balance = amount
    def deposit(self, amount):
        self.balance += amount
    def payInterest(self):
        self.balance *= 1 + intRate
```

```
>>>a = Account(100)
>>>a.balance
100
>>>a.intRate
.02
>>>a.deposit(50)
>>>a.balance
150
```

Rules for evaluating

- `<object>.<variable>`
  - Look for the value of that variable associated with that object
  - If not found, look for a value of that variable associated with the class
  - If it is a function, convert it to a method
    - Function that takes one less argument, uses `<object>` as the first argument

```
class Account:
    intRate = .02
    def __init__(self, amount):
        self.balance = amount
    def deposit(self, amount):
        self.balance += amount
    def payInterest(self):
        self.balance *= 1 + intRate
```

```
>>>a = Account(100)
>>>a.balance
100
>>>a.intRate
.02
>>>a.deposit(50)
>>>a.balance
150
>>>Account.intRate
.02
```

Rules for evaluating

- `<class>.<variable>`
  - Look for the value of that variable associated with that object

```
class Account:
    intRate = .02
    def __init__(self, amount):
        self.balance = amount
    def deposit(self, amount):
        self.balance += amount
    def payInterest(self):
        self.balance *= 1 + intRate
```

```
...
>>>Account.intRate
.02
>>>Account.deposit(a, 25)
>>>a.balance
175
```

Rules for evaluating

- `<class>.<variable>`
  - Look for the value of that variable associated with that object
  - Treat functions like other values

We can now create our own types for objects

- With general information about the type
- With specific information for each object
- With behavior

Objects *both* store values and do things

- Many languages are entirely object-oriented
- For us, it's a design decision

Let's make a school

- What objects?
- What data?
- What methods?
- What relationships?

```
class Account:
    intRate = .02
    def __init__(self, amount):
        self.balance = amount
    def deposit(self, amount):
        self.balance += amount
    def payInterest(self):
        self.balance *= 1 + self.intRate
    def withdraw(self, amount):
        self.balance -= amount

class SavingsAct(Account):
    intRate = .04
    withdrawFee = 1
    def withdraw(self, amount):
        Account.withdraw(self, amount + self.withdrawFee)
```

```
>>>a = SavingsAct(100)
>>>a.balance
100
>>>a.payInterest()
>>>a.balance
104.0
>>>a.withdraw(20)
83.0
```

Rule:

When evaluating a variable/method, if it is not found in the class, look at the parent class.

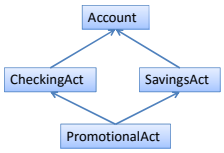
```
class Account:
    intRate = .02
    def __init__(self, amount):
        self.balance = amount
    def deposit(self, amount):
        self.balance += amount
    def payInterest(self):
        self.balance *= 1 + self.intRate
    def withdraw(self, amount):
        self.balance -= amount

class CheckingAcct(Account):
    minBalance = 1000
    def payInterest(self):
        if self.balance >= self.minBalance:
            Account.payInterest(self)
```

```
>>>a = CheckingAct(1000)
>>>a.balance
1000
>>>a.payInterest()
>>>a.balance
1040.0
>>>a.withdraw(50)
>>>a.balance
990.0
>>>a.payInterest()
>>>a.balance
990.0
```

New promotional account: all the features of both a checking account and a savings account. And you get $20 for opening an account!

```
class PromotionalAct(CheckingAct, SavingsAct):
    def __init__(self, amount):
        self.balance = amount + 20
```

What happens?




```
>>>a = CheckingAct(1000)
>>>a.balance
1000
>>>a is CheckingAct
True
>>>a is SavingsAct
False
>>>a is Account
False
>>>isinstance(a, CheckingAct)
True
>>>isinstance(a, SavingsAct)
False
>>>isinstance(a, Account)
True
>>>[c.__name__ for c in PromotionalAct.mro()]
['PromoitionalAct', 'CheckingAct', 'SavingsAct',
'Account', 'object']
```

Interfaces

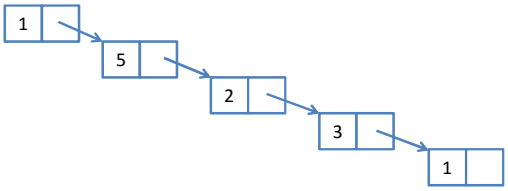
- Set of object methods and attributes, and properties thereof
- Like a parent class, but (in Python) not part of the programming language
- Doesn't actually implement anything

```
employeeActs = [a, b, c]
def payday(empacts):
    for a in empacts:
        a.deposit(100)
```

This could be using an "account" interface

- Must be a `deposit` method

Linked lists: Let's represent 1, 5, 2, 3, 1



Component: node

- Value
- Pointer to next node

Node also represents the linked list that starts at that node.

```
class LinkedList:
    def __init__(self, value, next=None):
        self.value = value
        self.rest = next
```

Can now make a list:

```
a = LinkedList(1)
a = LinkedList(3, a)
a = LinkedList(2, a)
a = LinkedList(5, a)
a = LinkedList(1, a)
```

What other methods should linked lists have?

```
class LinkedList:
    def __init__(self, value, next=None):
        self.value = value
        self.rest = next
    def append(self, value):
        if self.rest == None:
            self.rest = LinkedList(value)
        else:
            self.rest.append(value)
    def length(self):
        if self.rest == None:
            return 1
        else:
            return 1 + self.rest.length()
```

```
class LinkedList:
    def __init__(self, value, next=None):
        self.value = value
        self.rest = next
    def append(self, value):
        if self.rest == None:
            self.rest = LinkedList(value)
        else:
            self.rest.append(value)
    def length(self):
        if self.rest == None:
            return 1
        else:
            return 1 + self.rest.length()
    def getItem(self, index):
        if index == 0:
            return self.value
        else:
            return self.rest.getItem(index-1)
    def insert(self, index, value):
        ?
```

Let's try a new way…

```
class LLNode:
    def __init__(self, value, next=None):
        self.value = value
        self.rest = next
```

```
class LinkedList:
    def __init__(self):
        self.head = None
```

```
class LLNode:
    def __init__(self, value, next=None):
        self.value = value
        self.rest = next
```

```
class LinkedList:
    def __init__(self):
        self.head = None
    def append(self, value):
        if self.head == None:
            self.head = LLNode(value)
        else:
            curr = head
            while curr.rest != None:
                curr = curr.rest
            curr.rest = LLNode(value)
```

```
class LLNode:
    def __init__(self, value, next=None):
        self.value = value
        self.rest = next
    def append(self, value):
        if self.rest == None:
            self.rest = LLNode(value)
        else:
            self.rest.append(value)
```

```
class LinkedList:
    def __init__(self):
        self.head = None
    def append(self, value):
        if self.head == None:
            self.head = LLNode(value)
        else:
            self.head.append(value)
```

```
class LLNode:
    def __init__(self, value, next=None):
        self.value = value
        self.rest = next
    def append(self, value):
        if self.rest == None:
            self.rest = LLNode(value)
        else:
            self.rest.append(value)
```

```
class LinkedList:
    def __init__(self):
        self.head = None
    def append(self, value):
        if self.head == None:
            self.head = LLNode(value)
        else:
            self.head.append(value)
    def isEmpty(self):
        return self.head == None
```

```
class LLNode:
    def __init__(self, value, next=None):
        self.value = value
        self.rest = next
    def append(self, value):
        if self.rest == None:
            self.rest = LLNode(value)
        else:
            self.rest.append(value)
```

```
class LinkedList:
    def __init__(self):
        self.head = None
    def append(self, value):
        if self.head == None:
            self.head = LLNode(value)
        else:
            self.head.append(value)
    def isEmpty(self):
        return self.head == None
    def getLength(self):
        if self.head == None:
            return 0
        else:
            curr = self.head
            count = 1
            while curr.rest != None:
                curr = curr.rest
                count += 1
            return count
```

```
class LLNode:
    def __init__(self, value, next=None):
        self.value = value
        self.rest = next
    def append(self, value):
        if self.rest == None:
            self.rest = LLNode(value)
        else:
            self.rest.append(value)
    def getLength(self):
        if self.rest == None:
            return 1
        else:
            return 1 + self.rest.length()
```

```
class LinkedList:
    def __init__(self):
        self.head = None
    def append(self, value):
        if self.head == None:
            self.head = LLNode(value)
        else:
            self.head.append(value)
    def isEmpty(self):
        return self.head == None
    def getLength(self):
        if self.head == None:
            return 0
        else:
            return self.head.length()
```

How long does this take?

```
class LLNode:
    def __init__(self, value, next=None):
        self.value = value
        self.rest = next
    def append(self, value):
        if self.rest == None:
            self.rest = LLNode(value)
        else:
            self.rest.append(value)
```

```
class LinkedList:
    def __init__(self):
        self.head = None
        self.length = 0
    def append(self, value):
        self.length += 1
        if self.head == None:
            self.head = LLNode(value)
        else:
            self.head.append(value)
    def isEmpty(self):
        return self.head == None
    def getLength(self):
        return self.length
```

Is this faster?

```
class LLNode:
    def __init__(self, value, next=None):
        self.value = value
        self.rest = next
    def append(self, value):
        if self.rest == None:
            self.rest = LLNode(value)
        else:
            self.rest.append(value)
```

```
class LinkedList:
    def __init__(self):
        self.head = None
        self.length = 0
    def append(self, value):
        self.length += 1
        if self.head == None:
            self.head = LLNode(value)
        else:
            self.head.append(value)
    def isEmpty(self):
        return self.head == None
    def getLength(self):
        return self.length
    def insert(self, index, value):
        self.length += 1
        if index == 0:
            self.head = LLNode(value,se
        else:
            curr = self.head
            count = 1
            while count < index:
                curr = curr.rest
                count += 1
            curr.rest = LLNode(value,
                                curr.rest)
```

```
class LLNode:
    def __init__(self, value, next=None):
        self.value = value
        self.rest = next
    def append(self, value):
        if self.rest == None:
            self.rest = LLNode(value)
        else:
            self.rest.append(value)
```

```
class LinkedList:
    def __init__(self):
        self.head = None
        self.length = 0
    def append(self, value):
        self.length += 1
        if self.head == None:
            self.head = LLNode(value)
        else:
            self.head.append(value)
    def isEmpty(self):
        return self.head == None
    def __len__(self):
        return self.length
    def insert(self, index, value):
        self.length += 1
        if index == 0:
            self.head = LLNode(value,se
        else:
            curr = self.head
            count = 1
            while count < index:
                curr = curr.rest
                count += 1
            curr.rest = LLNode(value,
                                curr.rest)
```

```
class LLNode:
    def __init__(self, value, next=None):
        self.value = value
        self.rest = next
    def append(self, value):
        if self.rest == None:
            self.rest = LLNode(value)
        else:
            self.rest.append(value)
```

```
class LinkedList:
    def __init__(self):
        self.head = None
        self.length = 0
    def append(self, value):
        self.length += 1
        if self.head == None:
            self.head = LLNode(value)
        else:
            self.head.append(value)
    …
    def __len__(self):
        return self.length
    …
```

```
class LLNode:
    def __init__(self, value, next=None):
        self.value = value
        self.rest = next
    def append(self, value):
        if self.rest == None:
            self.rest = LLNode(value)
        else:
            self.rest.append(value)
    def getItem(self, index):
        if index == 0:
            return self.value
        else:
            return self.rest.getItem(index-1)
```

```
class LinkedList:
    def __init__(self):
        self.head = None
        self.length = 0
    def append(self, value):
        self.length += 1
        if self.head == None:
            self.head = LLNode(value)
        else:
            self.head.append(value)
    …
    def __len__(self):
        return self.length
    …
    def __getitem__(self, index):
        return self.head.getItem(index)
```

Now we can do this:

```
a = LinkedList()
a.append(4)
a.append(6)
a.insert(1, 7)
print("length:", len(a))
print("second:", a[2])
```

And get this:

```
length: 3
second: 6
```

Running time

Which algorithm is faster depends on what the input is.

- If we know what inputs we'll have, just time them.
- Usually it's time on the big/hard inputs that matters.

We consider efficiency

- in the worst case
- for large inputs

Two algorithm run times are **asymptotically equal** if *for big inputs* which algorithm is faster *depends on the relative speed of the computers.*

Example:

Algorithm A has a worst-case running time on n-bit inputs of $n^3 - 4n^2$ steps.

Algorithm B has a worst-case running time on n-bit inputs of $10n^3 + 15$ steps.

If A and B are run on equal-speed machines, A is faster.

If B is run on a machine that is 100 times faster, B is faster (for large inputs).

Two algorithm run times are **asymptotically equal** if *for big inputs* which algorithm is faster *depends on the relative speed of the computers.*

$f(n) \in \theta(g(n))$ means:

There exist positive constants c_1, c_2, n_0 such that

$$c_1 f(n) < g(n) < c_2 f(n) \quad \text{for all } n > n_0.$$

Searching

Given a sorted list and a value v, does the list include v?

Option 1:

```
def search(ls, v):
    for i in ls:
        if i == v:
            return True
    return False
```

Searching

Given a sorted list and a value v, does the list include v?

Option 2:

```
def search_help(ls, v, start, end):
    mid = (start + end) // 2
    if ls[mid] == v:
        return True
    elif start >= end:
        return False
    elif ls[mid] > v:
        return search(ls, v, start, mid-1)
    else:
        return search(ls, v, mid+1, end)

def search(ls, v):
    return search_help(ls, v, 0, len(ls)-1)
```

Sorting

We want to put a list in order.

Ideas?

Bubble Sort

```
def bubblesort(aList):
   for i in range(len(aList)):
      for k in range(len(aList)-1):
         if aList[k] > aList[k+1]:
            aList[k], aList[k+1] = aList[k+1], aList[k]
   return aList
```

Bubble Sort

```
def bubblesort(aList):
   for i in range(len(aList)):
      for k in range(len(aList)-1-i):
         if aList[k] > aList[k+1]:
            aList[k], aList[k+1] = aList[k+1], aList[k]
   return aList
```

Bubble Sort

```
def bubblesort(aList):
   for i in range(len(aList)):
      finished = True
      for k in range(len(aList)-1-i):
         if aList[k] > aList[k+1]:
            finished = False
            aList[k], aList[k+1] = aList[k+1], aList[k]
      if finished:
         break
   return aList
```

Insertion Sort

```
def insertionsort(aList):
   for i in range( 1, len( aList ) ):
      tmp = aList[i]
      k = i
      while k > 0 and tmp < aList[k - 1]:
         aList[k] = aList[k - 1]
         k -= 1
      aList[k] = tmp
   return aList
```

Merge Sort

```
def merge(a, b):
   c = []
   i, j = 0, 0
   while i + j < len(a) + len(b):
      if i >= len(a) or b[j] <= a[i]:
         c.append(b[j])
         j += 1
      elif j >= len(b) or a[i] <= b[j]:
         c.append(a[i])
         i += 1
   return c

def mergesort(aList):
   if len(aList) <= 1:
      return aList
   else:
      mid = len(aList) // 2
      return merge(mergesort(aList[:mid]),mergesort(aList[mid:]))
```

Aside: new problem

Input: a list of n bits, half 0s and half 1s in some order

Goal: output an index that points to a 1

One option:

```
def findOne(ls):
    for i in range(len(ls)):
        if ls[i] == 1:
            return i
```

What is the running time?

```
def findOne(ls):
    for i in range(len(ls)):
        if ls[i] == 1:
            return i
```

Another option:

```
def findOne(ls):
    while True:
        guess = randint(0, len(ls)-1)
        if ls[guess] == 1:
            return guess
```

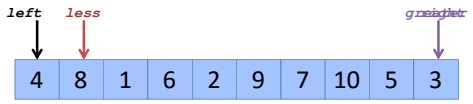
Running time?

Randomness can help!

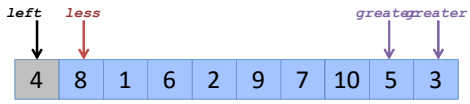
Quicksort: randomized sort

General idea:

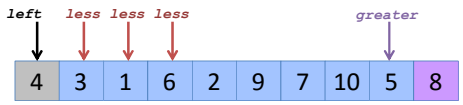
- Split the list into two parts, with everything in the first part less than everything in the second part
- Recurse on each part



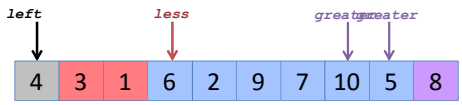

```
def partition(ls, left, right):
    less = left + 1
    greater = right
    while less <= greater:
        if ls[less] < ls[left]:
            less = less + 1
        else:
            ls[less],ls[greater]= ls[greater],ls[less]
            greater = greater - 1
    ls[left], ls[less - 1] = ls[less - 1], ls[left]
    return less - 1
```




```
def partition(ls, left, right):
    less = left + 1
    greater = right
    while less <= greater:
        if ls[less] < ls[left]:
            less = less + 1
        else:
            ls[less],ls[greater]= ls[greater],ls[less]
            greater = greater - 1
    ls[left], ls[less - 1] = ls[less - 1], ls[left]
    return less - 1
```



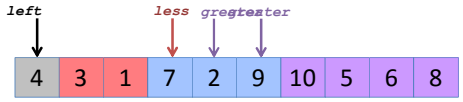

```
def partition(ls, left, right):
    less = left + 1
    greater = right
    while less <= greater:
        if ls[less] < ls[left]:
            less = less + 1
        else:
            ls[less],ls[greater]= ls[greater],ls[less]
            greater = greater - 1
    ls[left], ls[less - 1] = ls[less - 1], ls[left]
    return less - 1
```



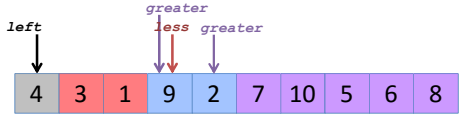

```
def partition(ls, left, right):
    less = left + 1
    greater = right
    while less <= greater:
        if ls[less] < ls[left]:
            less = less + 1
        else:
            ls[less],ls[greater]= ls[greater],ls[less]
            greater = greater - 1
    ls[left], ls[less - 1] = ls[less - 1], ls[left]
    return less - 1
```


```
def partition(ls, left, right):
    less = left + 1
    greater = right
    while less <= greater:
        if ls[less] < ls[left]:
            less = less + 1
        else:
            ls[less],ls[greater]= ls[greater],ls[less]
            greater = greater - 1
    ls[left], ls[less - 1] = ls[less - 1], ls[left]
    return less - 1
```

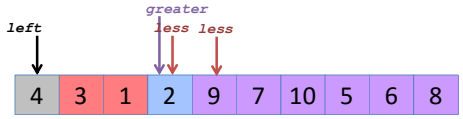

```
def partition(ls, left, right):
    less = left + 1
    greater = right
    while less <= greater:
        if ls[less] < ls[left]:
            less = less + 1
        else:
            ls[less],ls[greater]= ls[greater],ls[less]
            greater = greater - 1
    ls[left], ls[less - 1] = ls[less - 1], ls[left]
    return less - 1
```



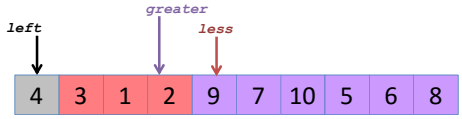

```
def partition(ls, left, right):
    less = left + 1
    greater = right
    while less <= greater:
        if ls[less] < ls[left]:
            less = less + 1
        else:
            ls[less],ls[greater]= ls[greater],ls[less]
            greater = greater - 1
    ls[left], ls[less - 1] = ls[less - 1], ls[left]
    return less - 1
```




```
def partition(ls, left, right):
    less = left + 1
    greater = right
    while less <= greater:
        if ls[less] < ls[left]:
            less = less + 1
        else:
            ls[less],ls[greater]= ls[greater],ls[less]
            greater = greater - 1
    ls[left], ls[less - 1] = ls[less - 1], ls[left]
    return less - 1
```




```
def partition(ls, left, right):
    less = left + 1
    greater = right
    while less <= greater:
        if ls[less] < ls[left]:
            less = less + 1
        else:
            ls[less],ls[greater]= ls[greater],ls[less]
            greater = greater - 1
    ls[left], ls[less - 1] = ls[less - 1], ls[left]
    return less - 1
```




```
def partition(ls, left, right):
    less = left + 1
    greater = right
    while less <= greater:
        if ls[less] < ls[left]:
            less = less + 1
        else:
            ls[less],ls[greater]= ls[greater],ls[less]
            greater = greater - 1
    ls[left], ls[less - 1] = ls[less - 1], ls[left]
    return less - 1
```

```
def partition(ls, left, right):
    less = left + 1
    greater = right
    while less <= greater:
        if ls[less] < ls[left]:
            less = less + 1
        else:
            ls[less],ls[greater]= ls[greater],ls[less]
            greater = greater - 1
    ls[left], ls[less - 1] = ls[less - 1], ls[left]
    return less - 1

def qshelp(ls, first, last):
    if first < last:
        pivot = partition(ls, first, last)
        qshelp(list, first, pivot-1)
        qshelp(list, pivot+1, last)

def quicksort(ls):
    qshelp(ls, 0, len(ls)-1)
```

Now that we know objects…

…let's go back to for statements.

We learned them as stepping through lists, but they actually can handle a wider set of things.

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

What they really require is an "iterable" object.

```
for a in b:
    <block>
```

Evaluation order:

- Call `b.__iter__` to get an "iterator" `it`
- Repeat:
  - Set `a = it.__next__`
  - Run the block
- If there is a `StopIteration` exception, stop repeating

Why not have the object itself be the iterator?

Why have iterators at all?

How `range` works:

`range(start, stop, step)` – sequence of integers starting with `start`, increasing by `step` each time, and ending *before* `stop`

`range(start, stop)` – sequence of integers starting with `start`, increasing by 1 each time, and ending *before* `stop`

`range(stop)` – sequence of integers starting with 0, increasing by 1 each time, and ending *before* `stop`

But it doesn't return a list!

- Take `a = range(1, 7, 2)`
- `a.__iter__`, or `iter(a)`, returns an iterator
- That iterator has a `__next__` method that steps through each value

Can make a list with `list(range(1, 7, 2))`

Error-handling

- Sometimes easier to catch errors than to prevent them

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Not a number. Try again.")
```

Multiple types of errors:

```
while True:
    try:
        x = int(input("Please enter a number: "))
        print(1/x)
        break
    except ValueError:
        print("Not a number. Try again.")
    except ZeroDivisionError:
        print("Can't invert zero. Try again")
```

Rules for executing try-except blocks:

- Run the try block until there is an error
- If there is an error, stop immediately and check each except block until one is found that handles that error
  - Run that block (and no others)

Errors are handled by the "most recent" try block

Errors are handled by the "most recent" try block.

```
def getNum():
    while True:
        try:
            x = int(input("Please enter a number: "))
            return x
        except ValueError:
            print("Not a number. Try again.")

try:
    y = getNum()
    print(1/y)
except ZeroDivisionError:
    print("Can't invert zero.")
except ValueError:
    print("The other ValueError-catcher.")
```

But they do handle errors raised within called functions.

```
def getNum():
    x = int(input("Please enter a number: "))

try:
    y = getNum()
    print(1/y)
except ZeroDivisionError:
    print("Can't invert zero.")
except ValueError:
    print("The other ValueError-catcher.")
```

Except clauses can list multiple types.

```
def getNum():
    x = int(input("Please enter a number: "))

try:
    y = getNum()
    print(1/y)
except (ZeroDivisionError, ValueError):
    print("Something went wrong.")
```

Except clauses can just handle everything.

```
def getNum():
    x = int(input("Please enter a number: "))

try:
    y = getNum()
    print(1/y)
except:
    print("Something went wrong.")
```

You can re-raise the error

```
try:
    <complex code>
except:
    print("Something went wrong.")
    print("Current values:")
    print(x, y, z)
    raise
```

You can also raise your own specified exceptions

```
def myDiv(x, y):
    if y == 0:
        raise ZeroDivisionError
    count = 0
    while x > y:
        x = x - y
        count += 1
    return count
```

You can also create your own types of exception

- Exceptions are objects in classes that inherit from `Exception`

You can also use assert statements

- Raises `AssertionError` if specified condition is not true

```
def myDiv(x, y):
    assert y > 0 and x >= 0
    count = 0
    while x > y:
        x = x - y
        count += 1
    return count
```

Why create errors?

- Can catch them
- Easier to find bugs if there's an error quickly

You can actually also add `else` and `finally` clauses to `try` blocks.

Rules for executing `try` blocks:

- Run the `try` block until there is an error
- If there is an error, stop immediately and check each except block until one is found that handles that error
  - Run that block (and no others)
- If there is no error, run the `else` block (if it exists)
- Run the `finally` block
- Report any error not handled

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")

>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Files let you
- Save data while program is not running
- Send data to others
- Work with large amounts of data

In Python, files are objects
- But use a special `open` constructor

```
f = open("testing.txt", "w")
f.write("let's write some words")
f.write("in a file")
f.write("and see what")
f.write("happens")
f.close()
```

Why do you need to close files?
- Buffering

```
f = open("testing.txt", "w")
f.write("let's write some words")
f.write("in a file")
f.write("and see what")
f.write("happens")
f.close()
```

Creates the following file:

```
let's write some wordsin a fileand see whathappens
```

Be careful about:
- Exact formatting (spaces, etc.)
- Line breaks, `\n`

```
f = open("testing.txt", "w")
f.write("Let's write some words ")
f.write("in a file\n")
f.write("and see what ")
f.write("happens.")
f.close()
```

Creates the following file:

```
Let's write some words in a file
and see what happens.
```

The `write` method only accepts strings
- Use `str()` to convert

Line endings are platform-specific

```
f = open("testing.txt", "w")
f.write("Let's write some words ")
f.write("in a file\n")
f.write("and see what ")
f.write("happens.")
f.close()
```

In order to make sure the file gets closed:
- try-except blocks
- with statement

```
with open("testing.txt", "w") as f:
    f.write("Let's write some words ")
    f.write("in a file\n")
    f.write("and see what ")
    f.write("happens.")
```

```
f = open("testing.txt", "w")
```

Details of the open statement:

File name:
- Relative to current directory
- Absolute

Mode:
- "w" – writing
  - Will replace an existing file!
- "a" – appending
  - Will add data to end of the file
- "r" – reading
  - Get data from the file
- "r+" – both reading and writing

Text modes

Given this file:

```
Let's write some words in a file
and see what happens.
```

Try this:

```
with open("testing.txt", "r") as f:
    x = f.readline()
    y = f.readline()
print(x)
print(y)
print("done")
```

And you get:

```
Let's write some words in a file

and see what happens.
done
```

Given this file:
```
Let's write some words in a file
and see what happens.
```

Try this:
```
with open("testing.txt", "r") as f:
    x = f.readline()
    y = f.readline()
    z = f.readline()
print(x)
print(y)
print(z)
print("done")
```

And you get:
```
Let's write some words in a file

and see what happens.

done
```

Given this file:
```
Let's write some words in a file
and see what happens.
```

Try this:
```
with open("testing.txt", "r") as f:
    x = f.readlines()
print(x)
```

And you get:
```
["Let's write some words in a file\n", 'and see what
happens.']
```

Given this file:
```
Let's write some words in a file
and see what happens.
```

Try this:
```
with open("testing.txt", "r") as f:
    for x in f:
        print(len(x))
```

And you get:
```
33
21
```

Given this file:
```
Let's write some words in a file
and see what happens.
```

Try this:
```
with open("testing.txt", "r") as f:
    f.seek(3)
    x = f.read(4)
    y = f.tell()
print(x)
print(y)
```

And you get:
```
's w
7
```

Warning:
`read()` reads the rest of the file.

Given this file:
```
Let's write some words in a file
and see what happens.
```

Try this:
```
with open("testing.txt", "r+") as f:
    f.seek(3)
    x = f.read(4)
    f.write("and then this")
print(x)
```

And you get:
```
's w
```

And the file is:
```
Let's write some words in a file
and see what happens.and then this
```