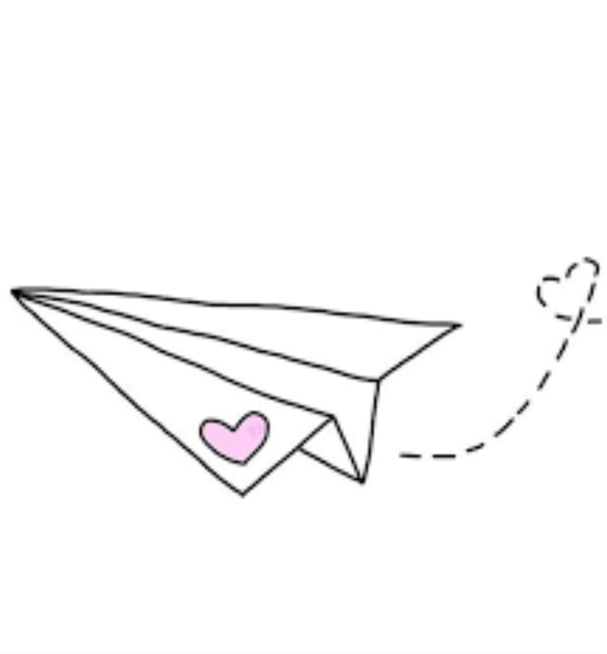
NETS 213: CROWDSOURCING AND HUMAN COMPUTATION

# Introduction to Python

Professor Callison-Burch

Penn Engineering
UNIVERSITY of PENNSYLVANIA

# For Loops



# For Loops

- ```
  for <item> in <collection>:
    <statements>
  ```

- If you’ve got an existing list, this iterates each item in it.
- You can generate a list with **Range:**

  `list(range(5))` **returns [0,1,2,3,4]**

  So we can say:

  ```
  for x in range(5):
    print(x)
  ```

- **&lt;item&gt; can be more complex than a single variable name.**

  ```
  for (x, y) in [('a',1), ('b',2), ('c',3), ('d',4)]:
    print(x)
  ```
  ```

# Iterators

# Iterator Objects

- **Iterable objects can be used in a `for` loop because they have an `__iter__` magic method, which converts them to iterator objects:**

```
>>> k = [1,2,3]

>>> k.__iter__()
<list_iterator object at 0x104f8ca50>

>>> iter(k)
<list_iterator object at 0x104f8ca10>
```

# Iterators

- Iterators are objects with a `__next__()` method:

```
>>> i = iter(k)
>>> next(i)
1
>>> i.__next__()
2
>>> i.next()
3
>>> i.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- Python iterators do not have a `hasnext()` method!
- Just catch the `StopIteration` exception

# Iterators: The truth about for... in...

- **for <item> in <iterable>:**
  **<statements>**

- **First line is just syntactic sugar for:**
  1. Initialize: Call **<iterable>.__iter__()** to create an *iterator*
- Each iteration:
  2. Call **iterator.__next__()** and bind **<item>**
  2a. Catch **StopIteration** exceptions

- **To be iterable: has __iter__ method**
  which returns an iterator obj
- **To be iterator: has __next__ method**
  which throws **StopIteration** when done

# An Iterator Class

```
class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

    def __iter__(self):
        return self

>>> for char in Reverse('spam'):
        print(char)
```

```
m
a
p
s
```

# Iterators use memory efficiently

Eg: File Objects

```
>>> for line in open("script.py"):   # returns iterator
...     print(line.upper())
...
IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(2 ** 3)
```

instead of

```
>>> for line in open("script.py").readlines(): #returns list
...     print(line.upper())
...
```

# Generators

# Generators: using `yield`

- Generators are iterators (with `__next()__` method)
- Creating Generators: `yield`

  Functions that contain the **`yield`** keyword ***automatically*** return a generator when called

```
>>> def f(n):
...     yield n
...     yield n+1
...
>>>
>>> type(f)
<class 'function'>
>>> type(f(5))
<class 'generator'>
>>> [i for i in f(6)]
[6, 7]
```

# Generators: What does `yield` do?

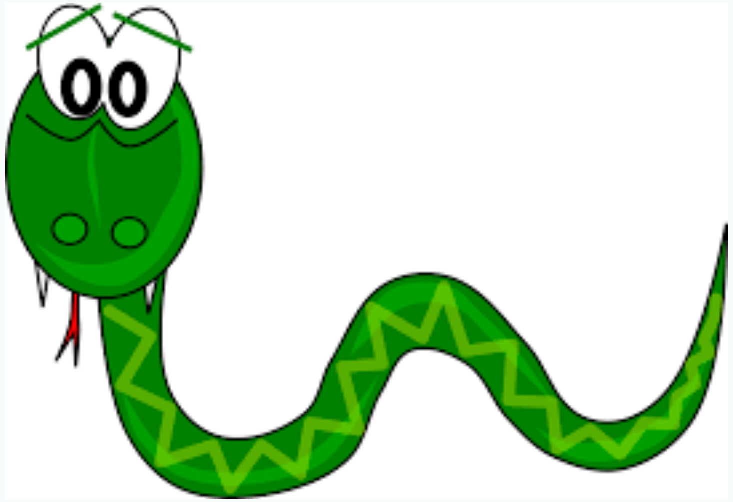
- Each time we call the `__next__` method of the generator, the method runs until it encounters a `yield` statement, and then it stops and returns the value that was yielded. Next time, it resumes where it left off.

```
>>> gen = f(5) # no need to say f(5).__iter__()
>>> gen
<generator object f at 0x1008cc9b0>
>>> gen.__next__()
5
>>> next(gen)
6
>>> gen.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

# Generators

- Benefits of using generators
    - Less code than writing a standard iterator
    - Maintains local state automatically
    - Values are computed one at a time, as they're needed
    - Avoids storing the entire sequence in memory
    - Good for aggregating (summing, counting) items. One pass.
    - Crucial for infinite sequences
    - Bad if you need to inspect the individual values

# Imports

# Import Modules and Files

```
>>> import math
>>> math.sqrt(9)
3.0
```

# Not as good to do this:

```
>>> from math import *
>>> sqrt(9)  # unclear where function defined

>>> import queue as Q
>>> q = Q.PriorityQueue()
>>> q.put(10)
>>> q.put(1)
>>> q.put(5)
>>> while not q.empty():
      print q.get(),
1, 5, 10
```

**Hint: Super useful for search algorithms**

# Import and pip

- pip is the The Python Package Installer
- It allows you to install a huge range of external libraries that have been packaged up and that are listed in the Python Package Index
- You run it from the command line:

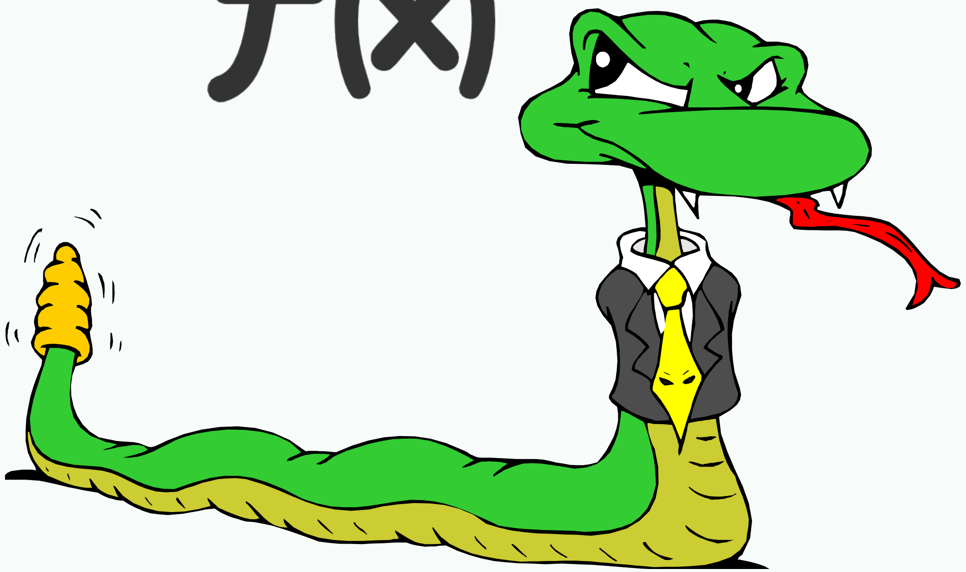
  pip install package_name
- In Google Colab, you can run command line arguments in the Python notebook by prefacing the commands with !:

  !pip install nltk

**Tip: if you ever get a** ModuleNotFoundError **then try** pip install **module name**

# Functions

$f(x)$



# Defining Functions

Function definition begins with def.

Function name and its arguments.

```
def get_final_answer(filename):
    """Documentation String"""
    line1
    line2
    return total_counter
...
```

First line with less indentation is considered to be outside of the function definition.

'return' indicates the value to be sent back to the caller.

**No declaration of <u>types</u> of arguments or result.**

# Function overloading? No.

- Python doesn't allow function overloading like Java deos
    - Unlike Java, a Python function is specified by its name alone
    - Two different functions can't have the same name, even if they have different numbers, order, or names of arguments
    - *But **operator** overloading – overloading +, ==, -, etc. – is possible using special methods on various classes*

- There is partial support in Python 3, but I don't recommend it
    - Python 3 – Function Overloading with singledispatch

# Default Values for Arguments

- You can provide default values for a function's arguments
- These arguments are optional when the function is called

```
>>> def myfun(b, c=3, d="hello"):
        return b + c

>>> myfun(5,3,"bob")
8
>>> myfun(5,3)
8
>>> myfun(5)
8
```

- Non-default argument should always precede default arguments; otherwise, it reports `SyntaxError`

# Keyword Arguments

- Functions can be called with arguments out of order
- These arguments are specified in the call
- Keyword arguments can be used after all other arguments.

```
>>> def myfun(a, b, c):
      return a - b

>>> myfun(2, 1, 43)           # 1
>>> myfun(c=43, b=1, a=2)     # 1
>>> myfun(2, c=43, b=1) # 1
>>> myfun(a=2, b=3, 5)
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

# *args

- Suppose you want to accept a variable number of non-keyword arguments to your function.

```
def print_everything(*args):
    """args is a tuple of arguments passed to the fn"""
    for count, thing in enumerate(args):
        print('{0}. {1}'.format(count, thing))

>>> lst = ['a', 'b', 'c']
>>> print_everything('a', ’b', 'c')
0. a
1. b
2. c
>>> print_everything(*lst) # Same results as above
```

# **kwargs

- Suppose you want to accept a variable number of keyword arguments to your function.

```
def print_keyword_args(**kwargs):
    # kwargs is a dict of the keyword args passed to the fn
    for key, value in kwargs.items(): #.items() is list
        print("%s = %s" % (key, value))
>>> kwargs = {'first_name': 'Bobby', 'last_name': 'Smith'}
>>> print_keyword_args(**kwargs)
first_name = Bobby
last_name = Smith
>>> print_keyword_args(first_name="John", last_name="Doe")
first_name = John
last_name = Doe
```

# Python uses dynamic scope

- Function sees the most current value of variables

```
>>> i = 10
>>> def add(x):
        return x + i

>>> add(5)
15
>>> i = 20
>>> add(5)
25
```

# Default Arguments & Memoization

- *Default parameter values are evaluated only when the* `def` *statement they belong to is first executed.*
- The function uses the same default object each call

```
def fib(n, fibs={}):
    if n in fibs:
        print('n = %d exists' % n)
        return fibs[n]
    if n <= 1:
        fibs[n] = n      # Changes fibs!!
    else:
        fibs[n] = fib(n-1) + fib(n-2)
    return fibs[n]
```

```
>>> fib(3)
n = 1 exists
2
```

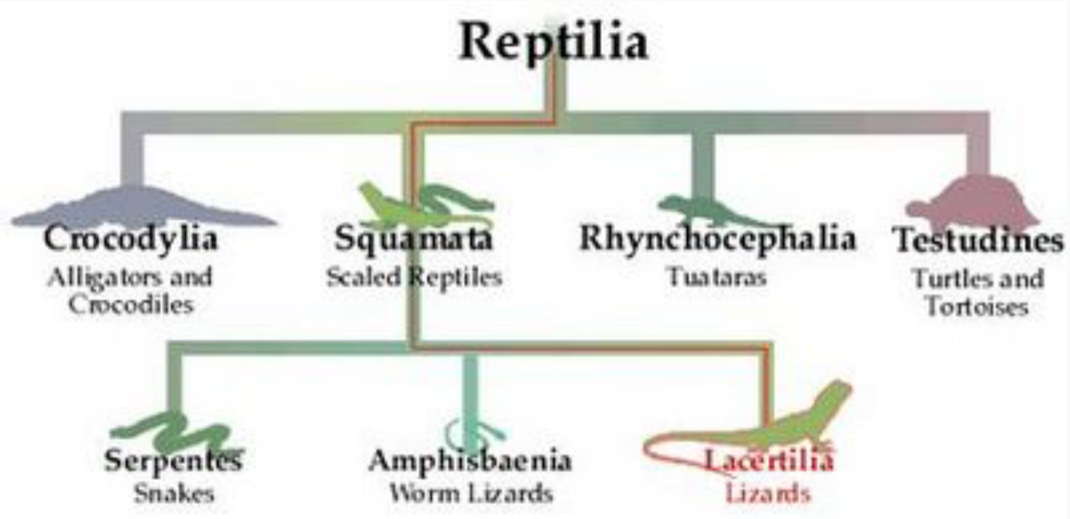
# Functions are “first-class” objects

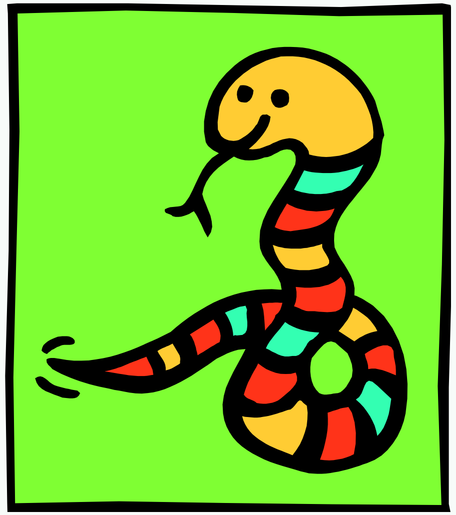
- First class object
  - An entity that can be dynamically created, destroyed, passed to a function, returned as a value, and have all the rights as other variables in the programming language have
- Functions are “first-class citizens”
  - Pass functions as arguments to other functions
  - Return functions as the values from other functions
  - Assign functions to variables or store them in data structures
- Higher order functions: take functions as input

```
def compose (f, g, x):
    return f(g(x))
```

```
>>> compose(str, sum, [1, 2, 3])
'6'
```

# Classes and Inheritance



# Creating a class

```
class Student:
    univ = "upenn" # class attribute

    def __init__(self, name, dept):
        self.student_name = name
        self.student_dept = dept

    def print_details(self):
        print("Name: " + self.student_name)
        print("Dept: " + self.student_dept)

student1 = Student("julie", "cis")
student1.print_details()
Student.print_details(student1)
Student.univ
```

Called when an object is instantiated

Every method begins with the variable **self**

Another member method

Creating an instance, note no **self**

Calling methods of an object

# Subclasses

- A class can *extend* the definition of another class
  - Allows use (or extension) of methods and attributes already defined in the previous one.
  - New class: *subclass*. Original: *parent*, *ancestor* or *superclass*

- To define a subclass, put the name of the superclass in parentheses after the subclass's name on the first line of the definition.

```
class Nets213Student(Student):
```

- Python has no 'extends' keyword like Java.
- Multiple inheritance is supported.

# Constructors: __init__

- Very similar to Java
- Commonly, the ancestor's `__init__` method is executed in addition to new commands
- *Must be done explicitly*
- You'll often see something like this in the `__init__` method of subclasses:

```
parentClass.__init__(self, x, y)
```

where parentClass is the name of the parent's class

```
Student.__init__(self, x, y)
```

# Redefining Methods

- Very similar to over-riding methods in Java

- To *redefine a method* of the parent class, include a new definition using the same name in the subclass.

  The old code in the parent class won't get executed.

- To execute the method in the parent class *in addition to* new code for some method, explicitly call the parent's version of the method.

```
parentClass.methodName(self, a, b, c)
```

**The only time you ever explicitly pass `self` as an argument is when calling a method of an ancestor.**

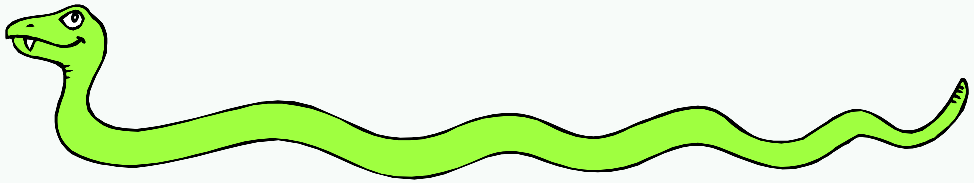
So use `myOwnSubClass.methodName(a,b,c)`

# Multiple Inheritance can be tricky

```
class A(object):
    def foo(self):
        print('Foo!')
class B(object):
    def foo(self):
        print('Foo?')
    def bar(self):
        print('Bar!')

class C(A, B):
    def foobar(self):
        super().foo()  # Foo!
        super().bar()  # Bar!
```

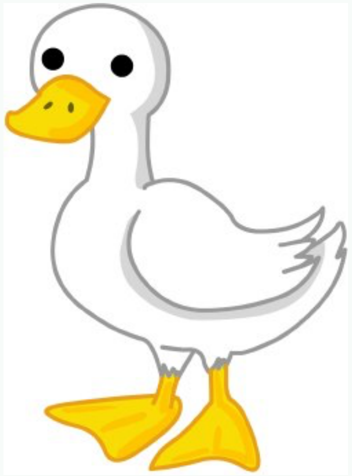
# Special Built-In Methods and Attributes



# Magic Methods and Duck Typing

- *Magic Methods* allow user-defined classes to behave like built in types

- *Duck typing* establishes suitability of an object by determining presence of methods
  - Does it swim like a duck and quack like a duck? It's a duck
  - Not to be confused with 'rubber duck debugging'



# Magic Methods and Duck Typing

```
class Duck:
    def fly(self):
        print("Duck flying")

class Airplane:
    def fly(self):
        print("Airplane flying")

class Whale:
    def swim(self):
        print("Whale swimming")

def lift_off(entity):
    entity.fly()

duck = Duck()
airplane = Airplane()
whale = Whale()

lift_off(duck) # prints `Duck flying`
lift_off(airplane) # prints `Airplane flying`
lift_off(whale) # Throws the error `'Whale' object has no attribute 'fly'`
```

# Example Magic Method

```
class Student:
   def __init__(self, full_name, age):
       self.full_name = full_name
       self.age = age

   def __str__(self):
       return "I'm named " + self.full_name + " - age: " + str(self.age)
   ...

>>> f = Student("Bob Smith", 23)

>>> print(f)
I'm named Bob Smith - age: 23
```

# Other "Magic" Methods

- Used to implement operator overloading

  Most operators trigger a special method, dependent on class

  **`__init__`**: The constructor for the class.
  **`__len__`** : Define how **`len(`** obj **`)`** works.
  **`__copy__`**: Define how to copy a class.
  **`__cmp__`** : Define how **`==`** works for class.
  **`__add__`** : Define how **`+`** works for class
  **`__neg__`** : Define how unary negation works for class

- Other built-in methods allow you to give a class the ability to use [ ] notation like an array or ( ) notation like a function call.

# Pandas

pandas

$$y_i t = \beta' x_{it} + \mu_i + \epsilon_{it}$$

# Other Resources

# Tons of good resources on YouTube


https://www.youtube.com/watch?v=_uQrJ0TkZlc