NETS 213: CROWDSOURCING AND HUMAN COMPUTATION

Introduction to Python

Professor Callison-Burch

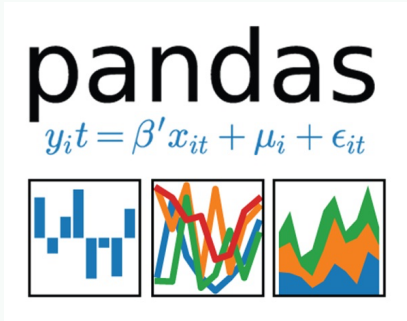
Penn Engineering
University of Pennsylvania

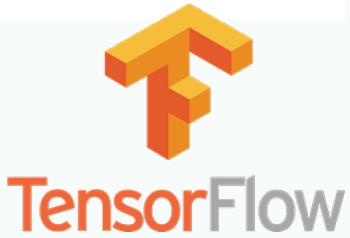
Python

- **Developed by Guido van Rossum in the early 90s**
  Originally Dutch, in USA since 1995.
  Benevolent Dictator for Life (now retired)
- **Named after the Monty Python comedy group**
- **Download from python.org**

Some Positive Features of Python

- **Fast development:**
  - Concise, intuitive syntax
    - Whitespace delimited
  - Garbage collected
- **Portable:**
  - Programs run on major platforms without change
  - cpython: common Python implementation in C.
- **Various built-in types:**
  - lists, dictionaries, sets: useful for AI
- **Large collection of support libraries:**
  - NumPy for Matlab like programming
  - Sklearn for machine learning
  - Pandas for data analysis





Recommended Reading

- **Python Overview**
  The Official Python Tutorial (https://docs.python.org/3/tutorial/index.html)
  Slides for CIS192
  (https://cis192.github.io/schedule/)
- **PEPs – Python Enhancement Proposals**
  PEP 8 - Official Style Guide for Python Code (Guido et al)
  - Style is about consistency. 4 space indents, < 80 char lines
  - Naming convention for functions and variables: lower_w_under
  - Use the automatic pep8 checker!
- PEP 20 – The Zen of Python (Tim Peters) (try: *import this*)
  Beautiful is better than ugly; simple is better than complex
  There should be one obvious way to do it
  That way may not be obvious at first unless you're Dutch
  Readability counts

PEP 8 — the Style Guide for Python Code

This stylized presentation of the well-established PEP 8 was created by Kenneth Reitz (for humans).

Introduction

This document gives coding conventions for the Python code comprising the standard library in the main Python distribution. Please see the companion informational PEP describing style guidelines for the C code in the C implementation of Python [1].

This document and PEP 257 (Docstring Conventions) were adapted from Guido's original Python Style Guide essay, with some additions from Barry's style guide [2].

This style guide evolves over time as additional conventions are identified and past conventions are rendered obsolete by changes in the language itself.

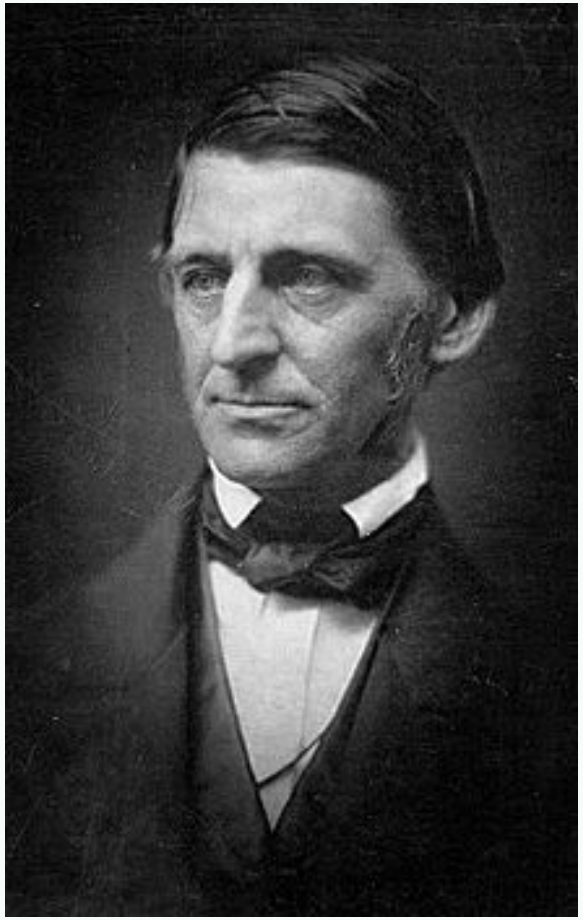
Many projects have their own coding style guidelines. In the event of any conflicts, such project-specific guides take precedence for that project.

A Foolish Consistency is the Hobgoblin of Little Minds

One of Guido's key insights is that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. As PEP 20 says, "Readability counts".

A style guide is about consistency. Consistency with this style guide is important.

Ralph Waldo Emerson



“A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines. With consistency a great soul has simply nothing to do. He may as well concern himself with his shadow on the wall. Speak what you think now in hard words, and tomorrow speak what tomorrow thinks in hard words again, though it contradict everything you said today. —'Ah, so you shall be sure to be misunderstood.'— Is it so bad, then, to be misunderstood? Pythagoras was misunderstood, and Socrates, and Jesus, and Luther, and Copernicus, and Galileo, and Newton, and every pure and wise spirit that ever took flesh. To be great is to be misunderstood.”

Python REPL Environment

- **REPL**
  - Read-Evaluate-Print Loop
  - Type “python” at the terminal
  - Convenient for testing
  - If you’d like syntax highlighting in REPL try bpython

You should use Python version 3.7 or version 3.8.

```
cis521x@eniac:~> python3
Python 3.4.6 (default, Mar 22 2017, 12:26:13) [GCC] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello World!')
Hello World!
>>> 'Hello World!'
'Hello World!'
>>> [2*i for i in range(10)]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> exit()
cis521x@eniac:~>
```

Python Scripts

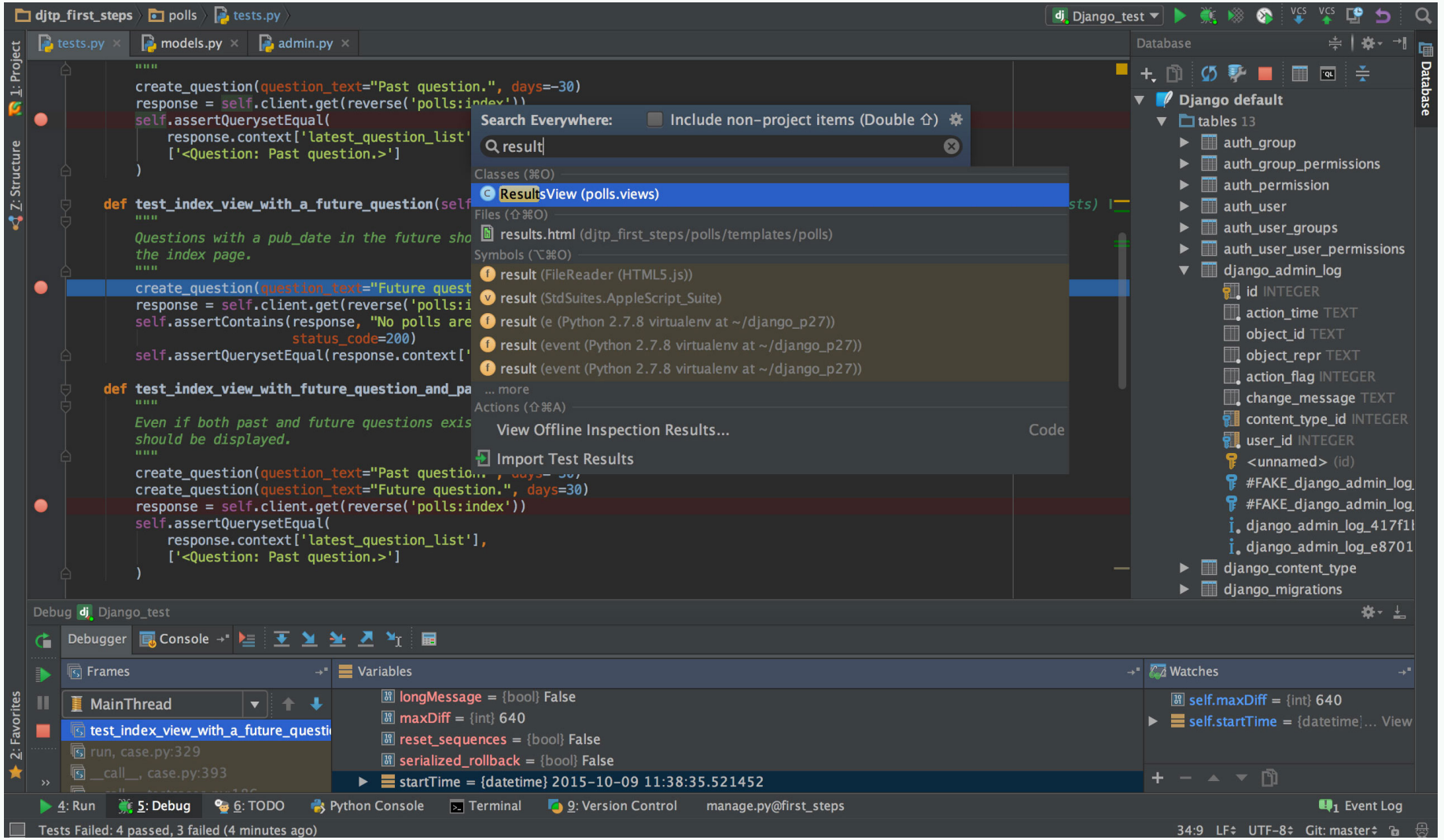
- **Scripts**
  - Create a file with your favorite text editor (like Sublime)
  - Type “python3 script_name.py” at the terminal to run
  - Not REPL, so you need to explicitly print

```
cis521x@eniac:~> cat foo.py
import random
def rand_fn():
    """outputs list of 10 random floats between [0.0, 1.0)"""
    return ["%.2f" % random.random() for i in range(10)]

print('1/2 = ', 1/2)
if __name__ == '__main__':
    rand_fn()
    print(rand_fn())

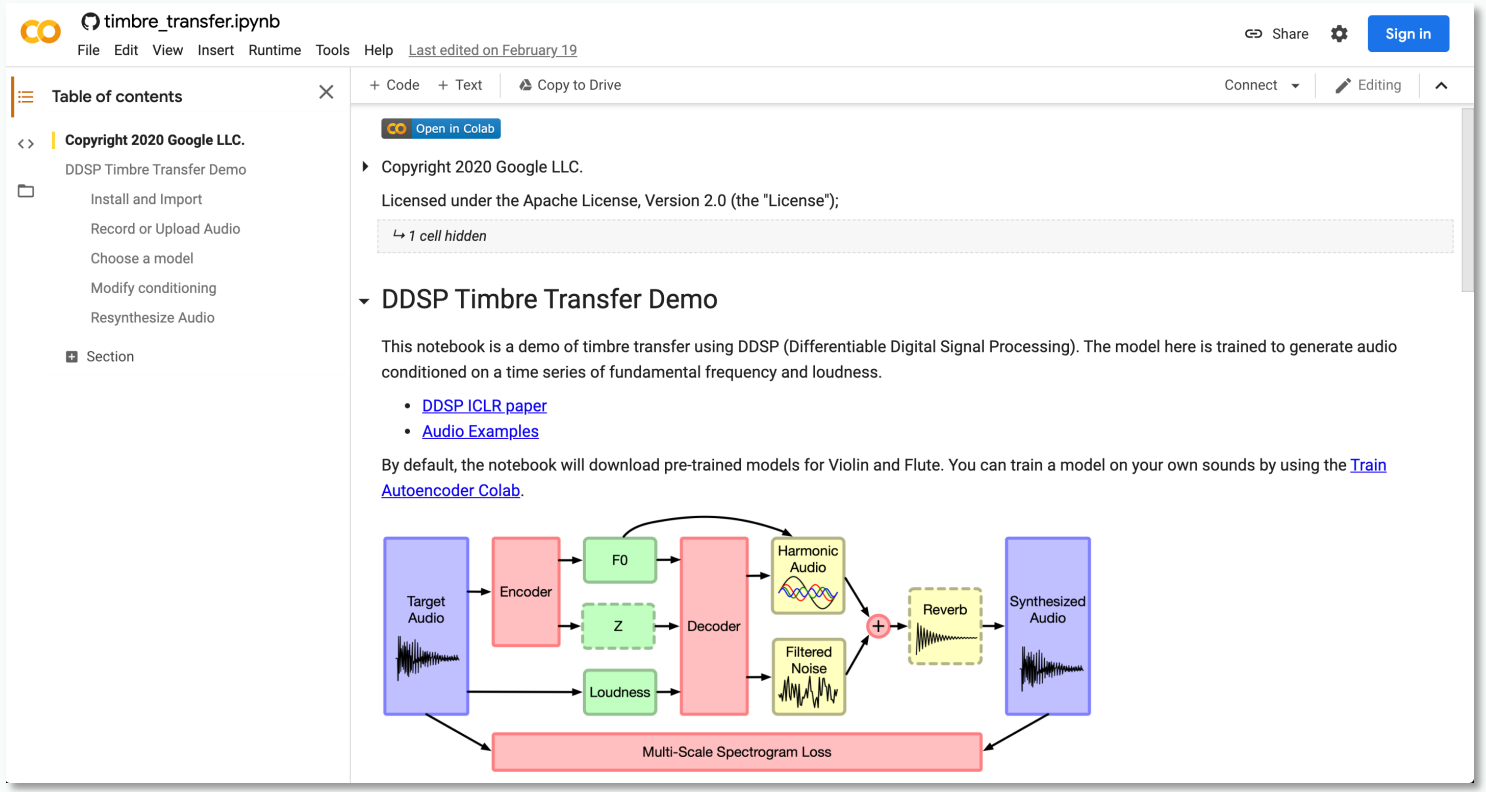
cis521x@eniac:~> python3 foo.py
1/2 =  0.5
['0.08', '0.10', '0.84', '0.01', '0.00', '0.59', '0.67', '0.88', '0.58', '0.81']
cis521x@eniac:~>
```

PyCharm IDE


djtp_first_steps
polls
tests.py
Django_test
tests.py
models.py
admin.py
Search Everywhere:
Include non-project items (Double ⇧)
result
Classes (⌘O)
ResultsView (polls.views)
Files (⇧⌘O)
results.html (djtp_first_steps/polls/templates/polls)
Symbols (⌥⌘O)
result (FileReader (HTML5.js))
result (StdSuites.AppleScript_Suite)
result (e (Python 2.7.8 virtualenv at ~/django_p27))
result (event (Python 2.7.8 virtualenv at ~/django_p27))
result (event (Python 2.7.8 virtualenv at ~/django_p27))
... more
Actions (⇧⌘A)
View Offline Inspection Results...
Code
Import Test Results
Database
Django default
tables 13
auth_group
auth_group_permissions
auth_permission
auth_user
auth_user_groups
auth_user_user_permissions
django_admin_log
id INTEGER
action_time TEXT
object_id TEXT
object_repr TEXT
action_flag INTEGER
change_message TEXT
content_type_id INTEGER
user_id INTEGER
django_content_type
django_migrations
Debug
Debugger
Console
Frames
Variables
Watches
MainThread
longMessage = {bool} False
maxDiff = {int} 640
reset_sequences = {bool} False
serialized_rollback = {bool} False
startTime = {datetime} 2015-10-09 11:38:35.521452
self.maxDiff = {int} 640
4: Run
5: Debug
6: TODO
Python Console
Terminal
9: Version Control
manage.py@first_steps
Event Log
Tests Failed: 4 passed, 3 failed (4 minutes ago)
34:9
LF
UTF-8
Git: master

Python Notebooks

- Jupyter Notebooks allow you to interactively run Python code in your web browser and share it with others in places like Google Colab
- They are popular for tutorials since you can include inline text and images



Structure of Python File

- **Whitespace is meaningful in Python**
- **Use a newline to end a line of code.**

  Use \ when must go to next line prematurely.
- **Block structure is indicated by indentation**

  The first line with less indentation is outside of the block.

  The first line with more indentation starts a nested block.

  Often a colon appears at the end of the line of a start of a new block. (E.g. for function and class definitions.)

A Simple Code Sample

```
x = 34 - 23                 # A comment.
y = 'Hello'                 # Another one.
z = 3.45
if z == 3.45 or y == 'Hello':
    x = x + 1
    y = y + ' World'        # String concat.
print(x)
print(y)
```

Objects and Types

- **All data treated as objects**
  An object is deleted (by garbage collection) once unreachable.
- **Strong Typing**
  Every object has a fixed type, interpreter doesn't allow things incompatible with that type (eg. "foo" + 2)
  `type(object)`
  `isinstance(object, type)`
- **Examples of Types:**
  int, float
  str, tuple, dict, list
  bool: True, False
  None, generator, function

Static vs Dynamic Typing

- **Java: *static* typing**
  - Variables can only refer to objects of a declared type
  - Methods use type signatures to enforce contracts
- **Python: *dynamic* typing**
  - Variables come into existence when first assigned.

```
>>> x = "foo”
>>> x = 2
```

  - type(var) automatically determined
  - If assigned again, type(var) is updated
  - *Functions have no type signatures*
  - Drawback: type errors are only caught at runtime

Math Basics

- **Literals**
  - Integers: 1, 2
  - Floats: 1.0, 2e10
  - Boolean: True, False
- **Operations**
  - Arithmetic: + - * /
  - Power: **
  - Modulus: %
  - Comparison: , <=, >=, ==, !=
  - Logic: (and, or, not) *not symbols*
- **Assignment Operators**
  - += *= /= &= ...
  - No ++ or --

Strings

- **Creation**
  - Can use either single or double quotes
  - Triple quote for multiline string and docstring
- **Concatenating strings**
  - By separating string literals with whitespace
  - Special use of ‘+’
- **Prefixing with r means raw.**
  - No need to escape special characters: r’\n’
- **String formatting**
  - Special use of ‘%’ (as in printf in C)
  - print("%s can speak %d languages" % ("C3PO", 6000000))
- **Immutable**

References and Mutability

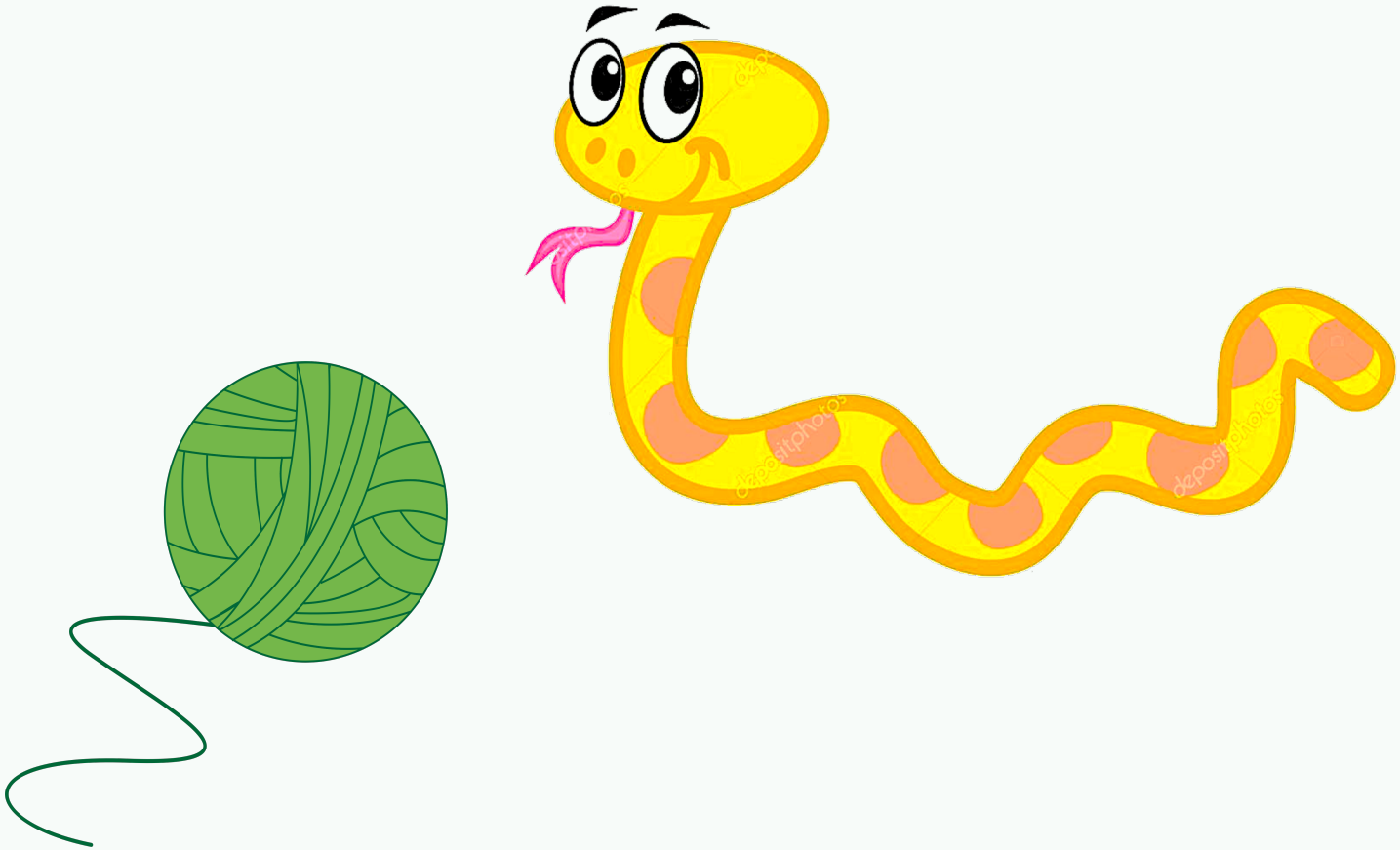
```
>>> x = 'foo  '
>>> y = x
>>> x = x.strip() # new obj
>>> x
'foo'
>>> y
'foo  '
```

- strings are immutable
- == checks whether variables point to objects of the same value
- is checks whether variables point to the same object

```
>>> x = [1, 2, 3, 4]
>>> y = x
>>> x.append(5) #same obj
>>> y
[1, 2, 3, 4, 5]
>>> x
[1, 2, 3, 4, 5]
```

- lists are mutable
- use y = x[:] to get a (shallow) copy of any sequence, ie. a new object of the same value

Sequence types: Tuples, Lists, and Strings



Sequence Types

- **Tuple**
  - A simple *immutable* ordered sequence of items
  - *Immutable*: a tuple cannot be modified once created
  - Items can be of mixed types, including collection types
- **Strings**
  - *Immutable*
  - Very much like a tuple with different syntax
  - Regular strings are Unicode and use 2-byte characters (Regular strings in Python 2 use 8-bit characters)
- **List**
  - *Mutable* ordered sequence of items of mixed types

Sequence Types

- **The three sequence types share much of the same syntax and functionality.**

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def') # tuple

>>> li = ['abc', 34, 4.34, 23] # list

>>> st = "Hello World"; st = 'Hello World' # strings

>>> tu[1]  # Accessing second item in the tuple.
'abc'

>>> tu[-3]  #negative lookup from right, from -1
4.56
```

Slicing: Return Copy of a Subsequence

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')

>>> t[1:4] #slicing ends before last index
('abc', 4.56, (2,3))

>>> t[1:-1] #using negative index
('abc', 4.56, (2,3))

>>> t[1:-1:2] # selection of every nth item.
('abc', (2,3))

>>> t[:2] # copy from beginning of sequence
(23, 'abc')

>>> t[2:] # copy to the very end of the sequence
(4.56, (2,3), 'def')
```

Operations on Lists

```
>>> li = [1, 11, 3, 4, 5]
>>> li.append('a')  # Note the method syntax
>>> li
[1, 11, 3, 4, 5, 'a']
>>> li.insert(2, 'i')
>>> li
[1, 11, 'i', 3, 4, 5, 'a']
>>> li = ['a', 'b', 'c', 'b']
>>> li.index('b')     # index of first occurrence
1
>>> li.count('b')     # number of occurrences
2
>>> li.remove('b')    # remove first occurrence
>>> li
['a', 'c', 'b']
```

Operations on Lists 2

```
>>> li = [5, 2, 6, 8]
>>> li.reverse()    # reverse the list *in place* (modify)
>>> li
[8, 6, 2, 5]

>>> li.sort()       # sort the list *in place*
>>> li
[2, 5, 6, 8]

>>> li.sort(some_function)
# sort in place using user-defined comparison

>>> sorted(li)  #return a *copy* sorted
```

Operations on Strings

```
>>> s = "Pretend this sentence makes sense."
>>> words = s.split(" ")
>>> words
['Pretend', 'this', 'sentence', 'makes', 'sense.']
>>> "_".join(words) #join method of obj "_"
'Pretend_this_sentence_makes_sense.'

>>> s = 'dog'
>>> s.capitalize()
'Dog'
>>> s.upper()
'DOG'
>>> ' hi --'.strip(' –')
'hi'
```

https://docs.python.org/3.7/library/string.html

Tuples

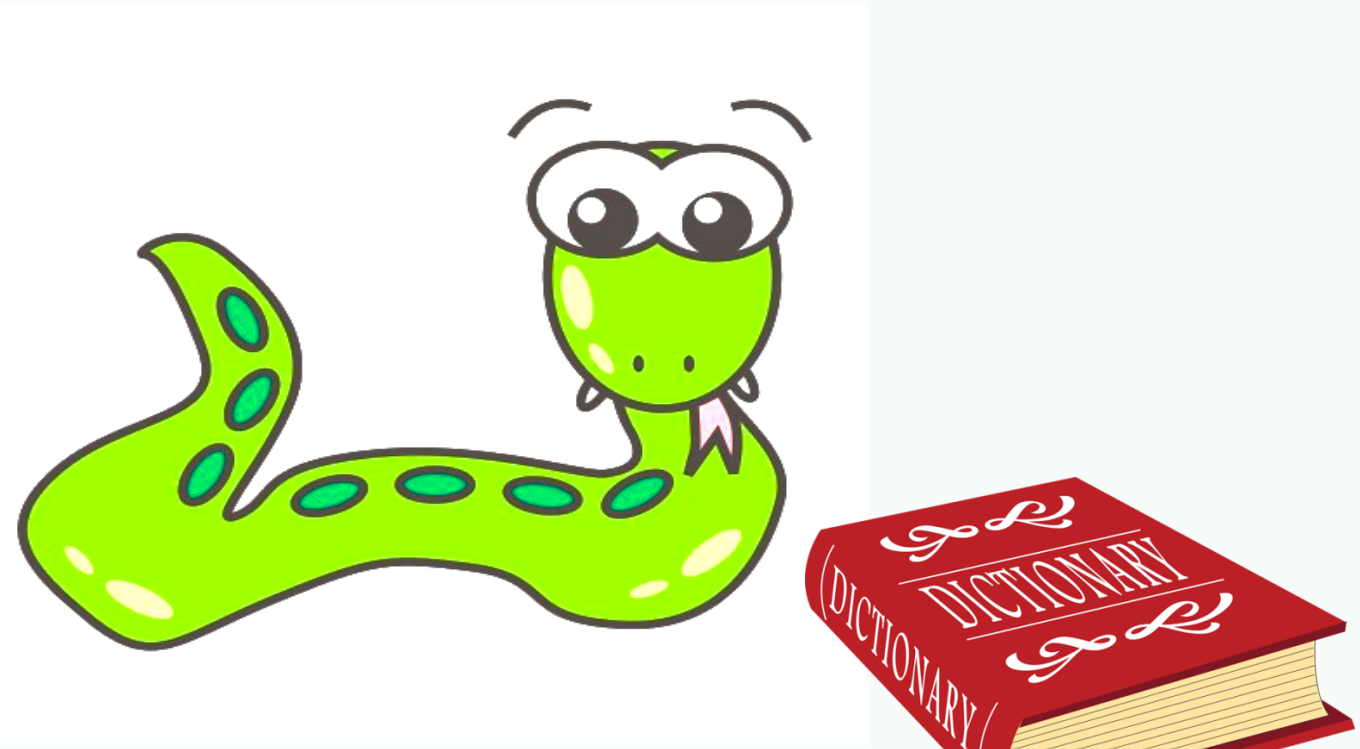
```
>>> a = ["apple", "orange", "banana"]
>>> for (index, fruit) in enumerate(a):
...         print(str(index) + ": " + fruit)
...
0: apple
1: orange
2: banana

>>> a = [1, 2, 3]
>>> b = ['a', 'b', 'c', 'd']
>>> list(zip(a, b))
[(1, 'a'), (2, 'b'), (3, 'c')]

>>> list(zip("foo", "bar"))
[('f', 'b'), ('o', 'a'), ('o', 'r')]

>>> x, y, z = 'a', 'b', 'c'
```

Dictionaries: a *mapping* collection type



Dict: Create, Access, Update

- **Dictionaries are unordered & work by hashing, so keys must be immutable**
- **Constant average time add, lookup, update**

```
>>> d = {'user' : 'bozo', 'pswd':  1234}

>>> d['user']
'bozo'

>>> d['bozo']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'bozo'

>>> d['user'] = 'clown'  # Assigning to an existing key   replaces its value.

>>> d
{'user': 'clown', 'pswd': 1234}
```

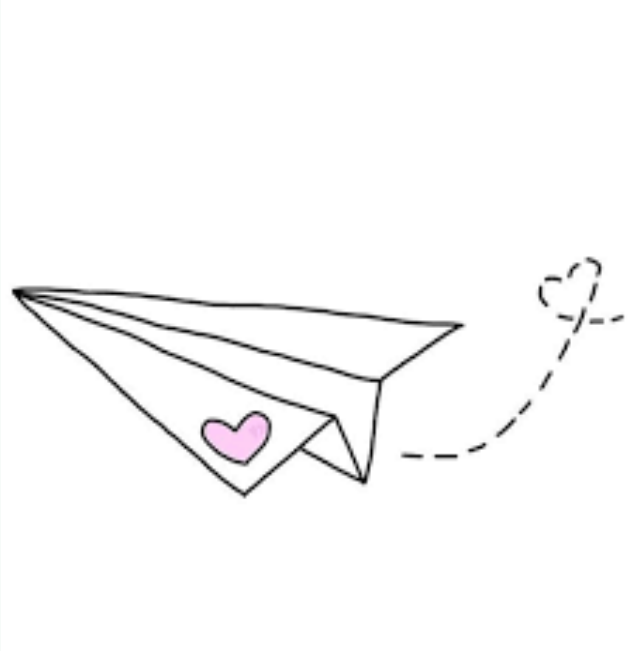
Dict: Useful Methods

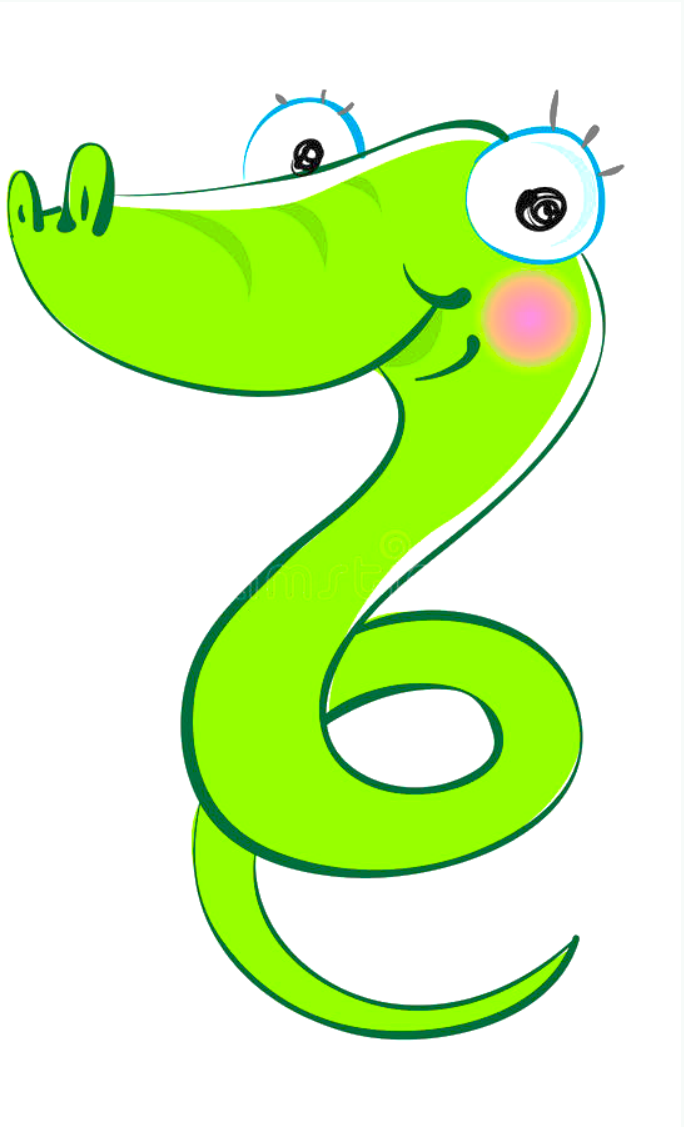
```
>>> d = {'user':'bozo', 'p':1234, 'i':34}
>>> d.keys()              # List of current keys
dict_keys(['user', 'p', 'i'])
>>> d.values()            # List of current values.
dict_values(['bozo', 1234, 34])
>>> d.items()       # List of item tuples.
dict_items([('user', 'bozo'), ('p', 1234), ('i', 34)])

>>> from collections import defaultdict
>>> d = defaultdict(int)
>>> d['a']
0
```

- defaultdict **automatically initializes nonexistent dictionary values**

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)
  ```

- **<item> can be more complex than a single variable name.**

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

[expression for name in list]

List Comprehensions replace loops!

```
nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
#  I want 'n*n' for each 'n' in nums
squares = []
for n in nums:
  for n in nums:
    squares.append(x*x)
print(squares)
```

```
squares = [x*x for x in nums]
print(squares)
```

[expression for name in list]

List Comprehensions replace loops!

```
>>> li = [3, 6, 2, 7]
>>> [elem * 2 for elem in li]
[6, 12, 4, 14]
```

```
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [n * 3 for (x, n) in li]
[3, 6, 21]
```

[expression for name in list if filter]

Filtered List Comprehensions

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem * 2 for elem in li if elem > 4]
[12, 14, 18]
```

- Only 6, 7, and 9 satisfy the filter condition.
- So, only 12, 14, and 18 are produced.

Dictionary, Set Comprehensions

```
lst1 = [('a', 1), ('b', 2), ('c', 'hi')]
lst2 = ['x', 'a', 6]

d = {k: v for k,v in lst1}
s = {x for x in lst2}

d = dict() # translation
for k, v in lst1:
    d[k] = v
s = set() # translation
for x in lst:
    s.add(x)

# Both value of d: {'a': 1, 'b': 2, 'c': 'hi'}
# Both value of d: {'x', 'a', 6}
```

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?

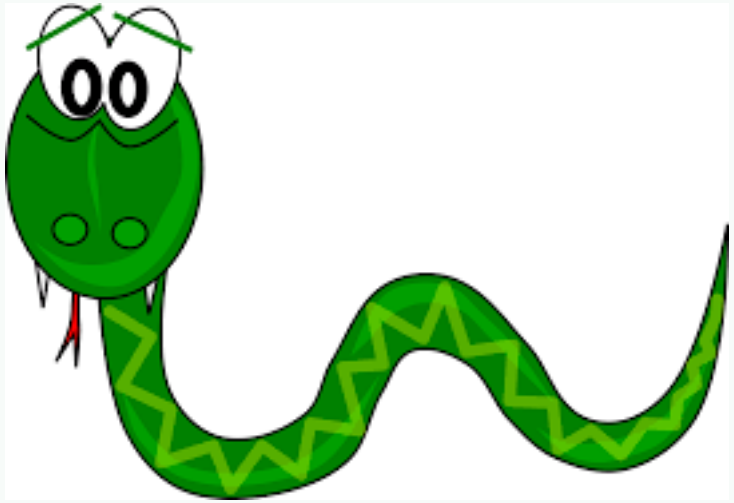
- Each time we call the `__next__` method of the generator, the method runs until it encounters a `yield` statement, and then it stops and returns the value that was yielded. Next time, it resumes where it left off.

```
>>> 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

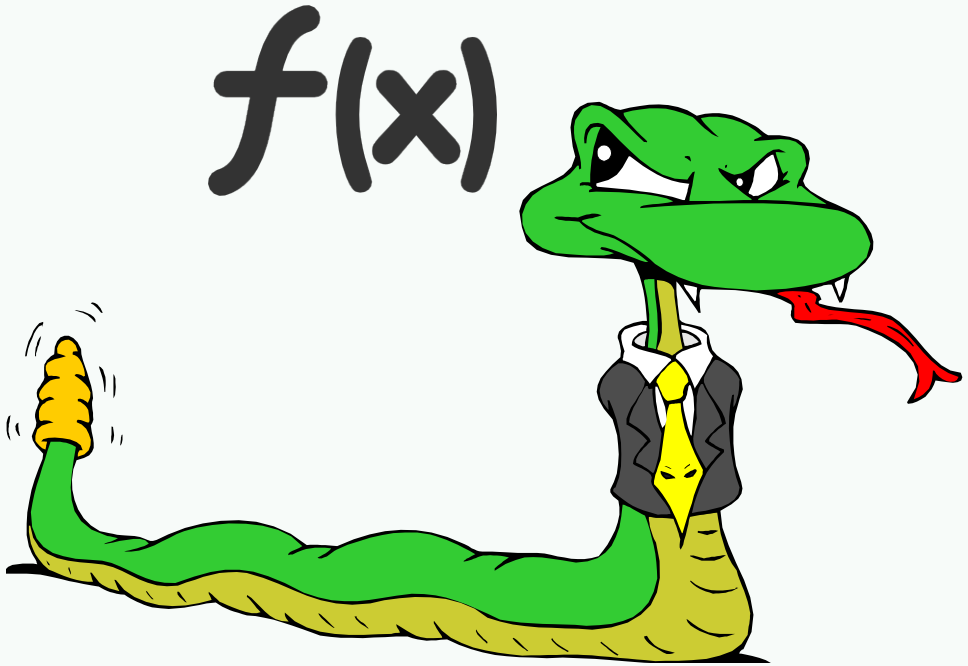
- pip is the The Python Package Installer
- It allows you to install a huge range of external libraries that have been packaged up and that are listed in the Python Package Index
- You run it from the command line:

  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



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 types 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
```

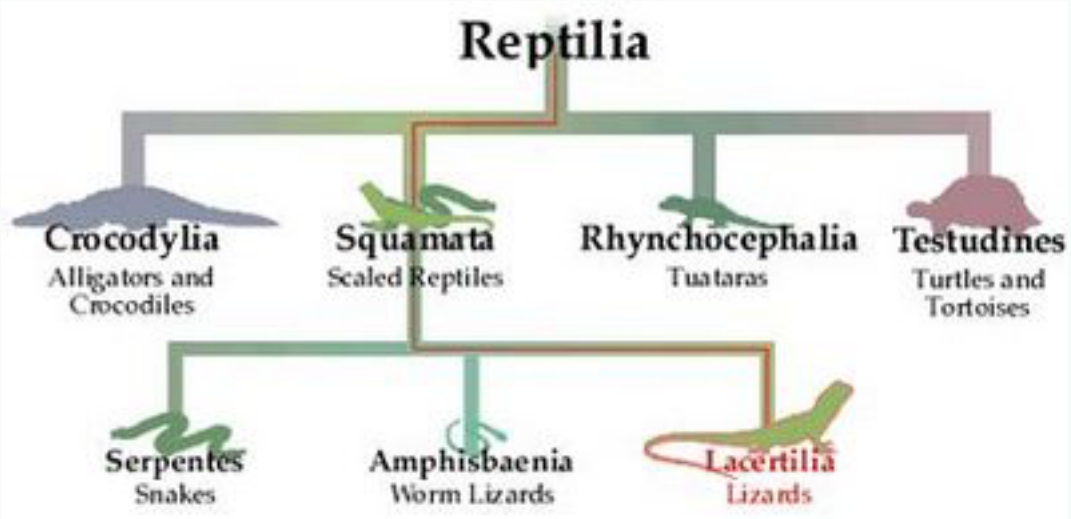
Functions are “first-class” objects

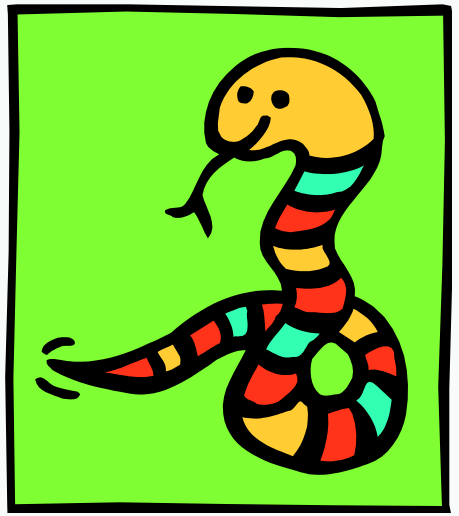
- First class object
  - An entity that can be dynamically created, destroyed, passed to a function, returned as a value, and have all the rights as other variables in the programming language have
- Functions are “first-class citizens”
  - Pass functions as arguments to other functions
  - Return functions 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.

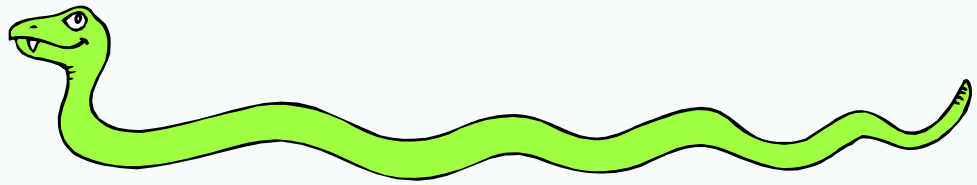
So use `myOwnSubClass.methodName(a,b,c)`

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!
```

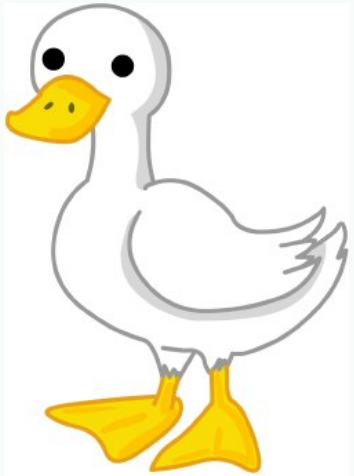
Special Built-In Methods and Attributes



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.

Other Resources

Tons of good resources on YouTube

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