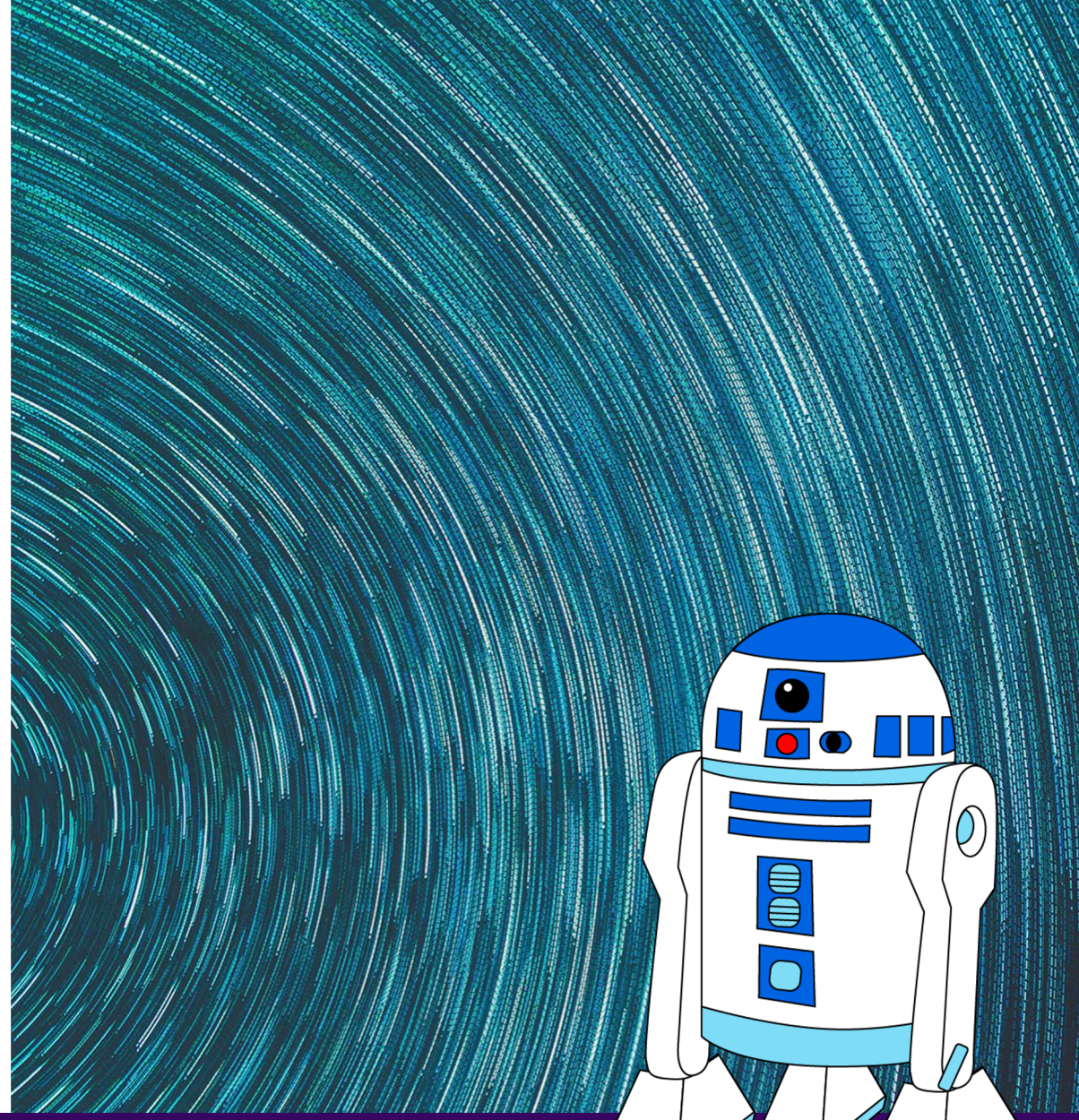
CIS 421/521:
ARTIFICIAL INTELLIGENCE

Introduction to Python

Professor Chris Callison-Burch



Penn Engineering
University of Pennsylvania

Python

- **Developed by Guido van Rossum in 1989**
  - Originally Dutch, in USA since 1995.
  - Benevolent Dictator for Life (now retired)
- **Python inspired by ABC language**
- **Van Rossum submitted a DARPA proposal "Computer Programming for Everybody"**
  - An easy and intuitive language just as powerful as major competitors
  - Open source, so anyone can contribute to its development
  - Code that is as understandable as plain English
  - Suitability for everyday tasks, allowing for short development times
- **Named after the Monty Python comedy group**

Some Positive Features of Python

- **Fast development:**
  - Concise, intuitive syntax that is whitespace delimited
  - Garbage collected
- **Portable:**
  - Programs run on major platforms without change
- **Various built-in types:**
  - lists, dictionaries, sets: useful for AI
- **Large collection of support libraries:**
  - NumPy for Matlab like programming
  - Pandas for data analysis
  - Sklearn for machine learning
  - Pytorch and TensorFlow for deep learning

PEP8: Python Style Guide

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.

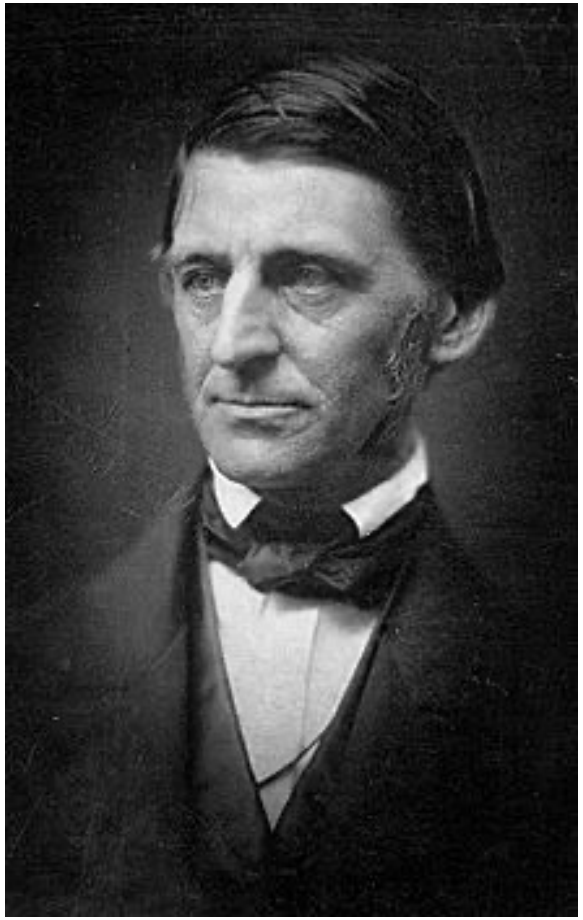
Many projects have their own coding style guidelines. In the event of any conflicts, such project-specific guides take precedence for that project.

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 “python3” in your terminal
  - Convenient for testing

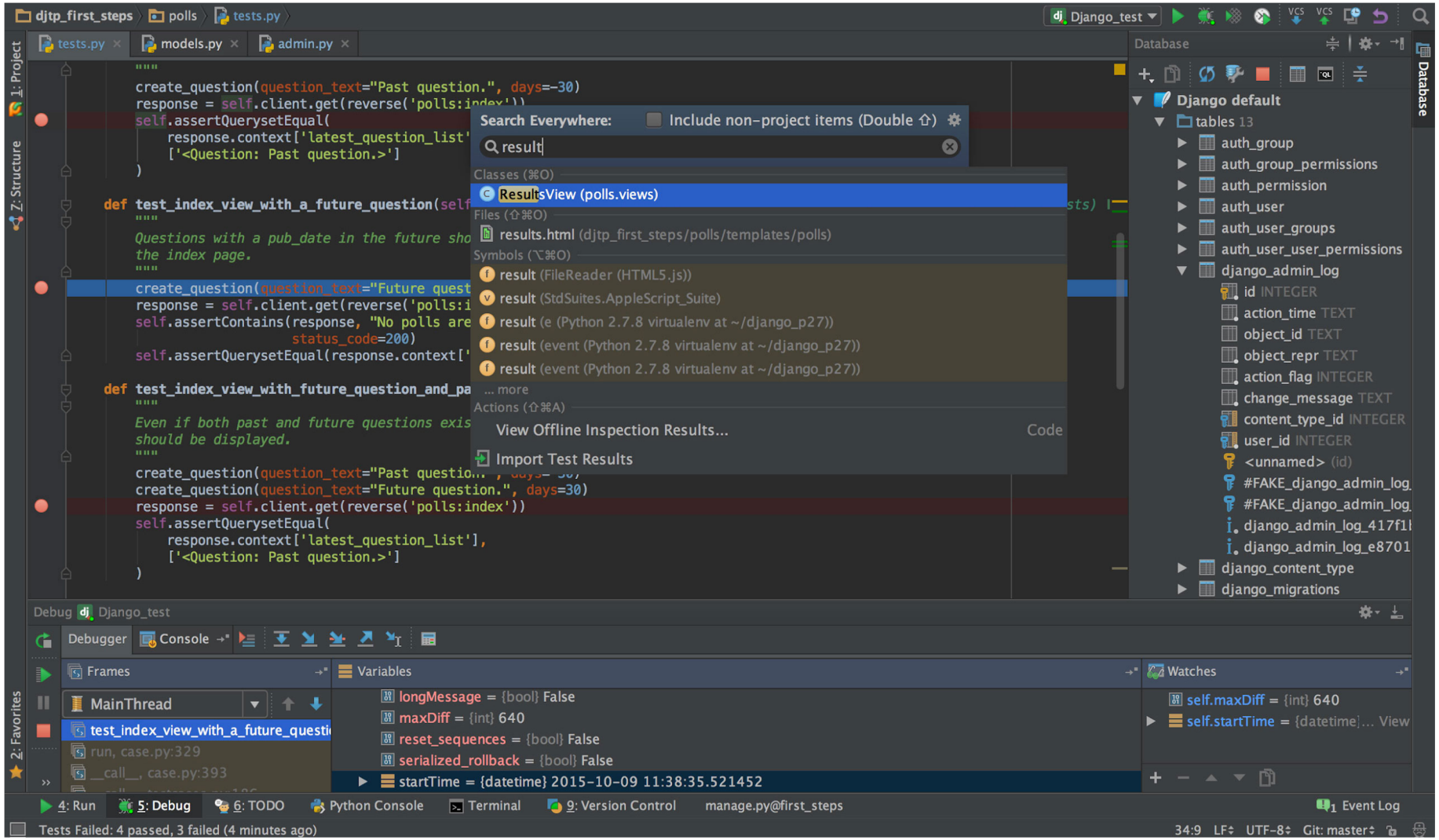
```
cis521x@eniac:~> python3
Python 3.4.6 (default, Mar 22 2017, 12:26:13) [GCC] on linux
Type “help”, “copyright”, “credits” or license for more information.
>>> print('Hello World!')
Hello World!
>>> 'Hello World!'
‘Hello World!'
>>> [2*i for i in range(10)]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>>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

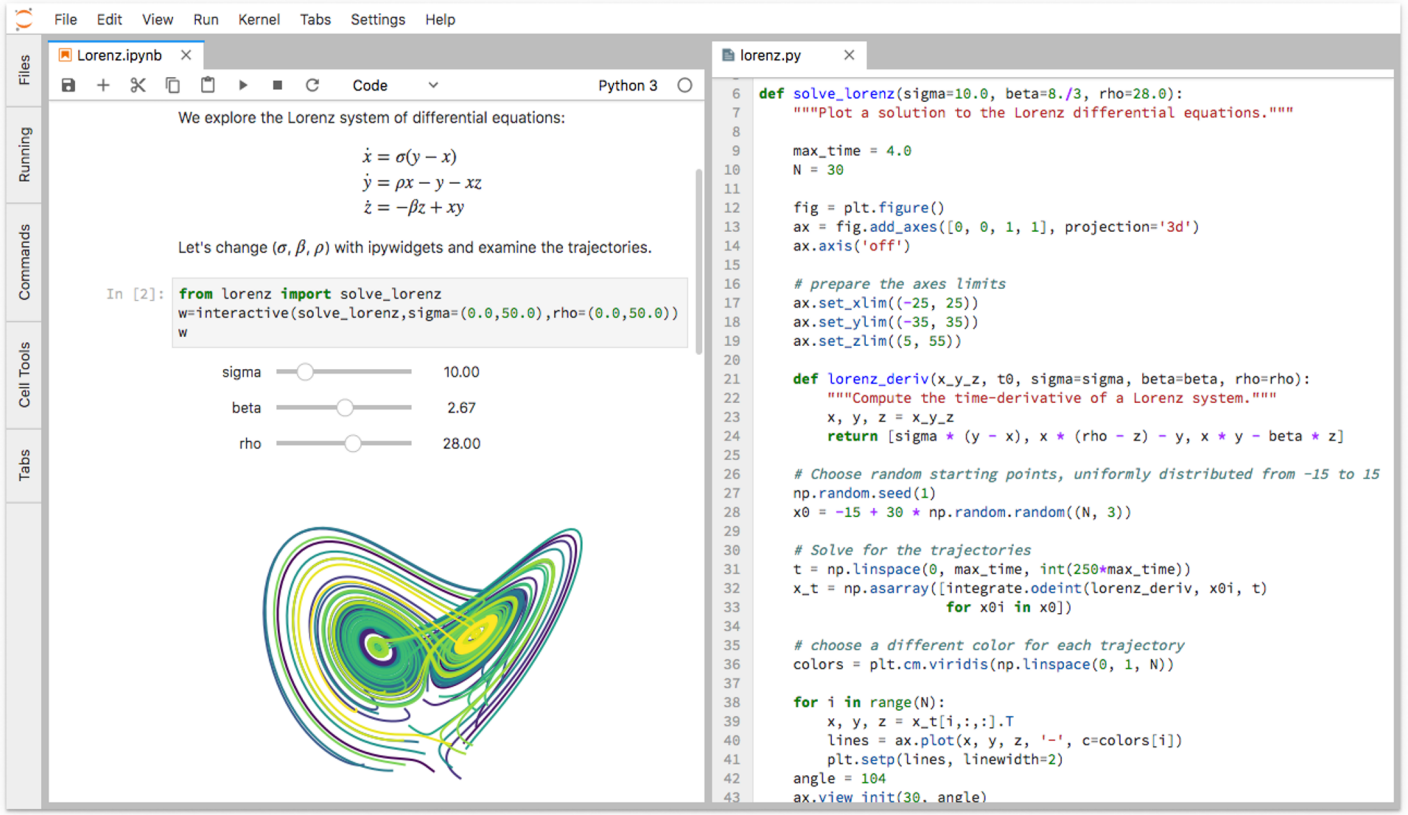
```
cis521x@eniac:~> cat foo.py
import random
def rand_fn():
    """outputs list of 10 random floats between [0.0, 1.0)"""
    return [“%.2f % random.random() for i in range(10)]

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 Django_test
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**



Simple Programs in Java and Python

Java

```java
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Must be saved in a file named **HelloWorld.java**

Python

```python
print("Hello World!")
```

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.)

Conditionals in Java and Python

Java

```java
class HelloWorld {
    public static void main(String[] args) {
        boolean isPandemicOver = true;
        System.out.println("Hello, World!");
         if (isPandemicOver) {
            System.out.println("Lovely to see
  you!");
        } else {
            System.out.println("I miss you!");
        }
    }
}
```

Java delineates code blocks with curly brackets.

Python

```python
pandemic_is_over = True
if pandemic_is_over:
    print("Hello World! Lovely to see you again!")
else:
    print("Hello World! I miss you!")
pandemic_is_over = False
```

Java delineates a code blocks with a colon and indentation.

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
    `int x = 2`
    `String y = "foo"`
  - Methods use type signatures to enforce contracts
    `public static void main(String[] args)`
- **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, 2e9
  - Boolean: True, False
- **Operations**
  - Arithmetic: + - * /
  - Power: **
  - Modulo: %
  - 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  '
```

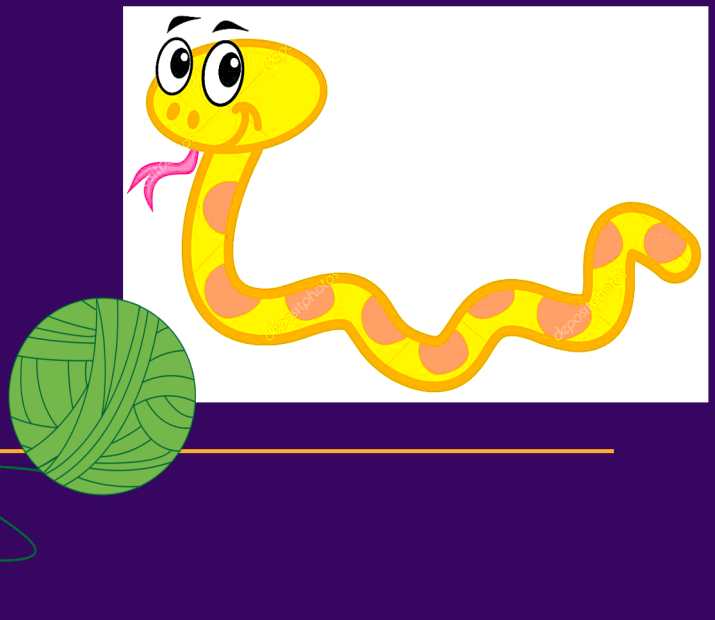
- strings are immutable
- == checks whether variables point to objects of the same value
- is checks whether variables point to the same object

```
>>> 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*
  - 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

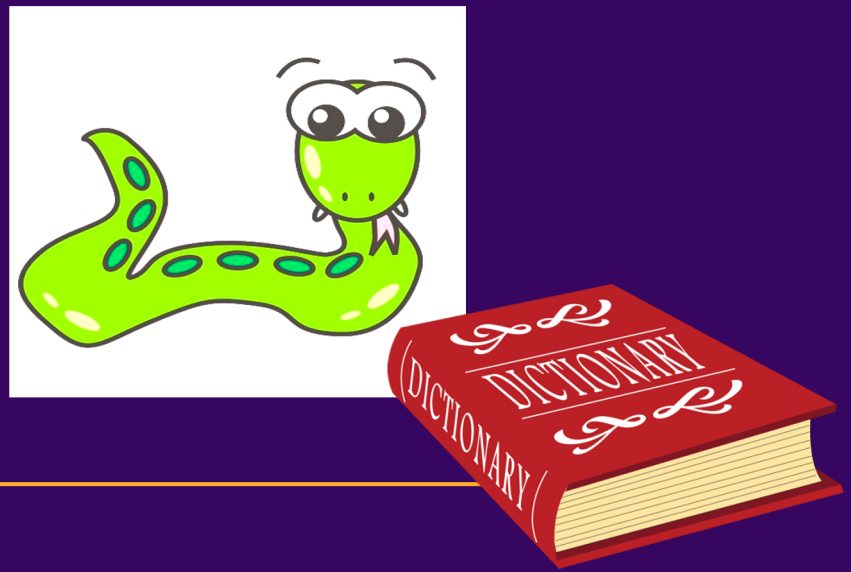
```
>>> s = "Pretend this sentence makes sense."
>>> words = s.split(" ")
>>> words
['Pretend', 'this', 'sentence', 'makes', 'sense.']
>>> "_".join(words) #join method of obj "_"
'Pretend_this_sentence_makes_sense.'

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

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

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' : 'R2D2', 'pswd':  1234}
>>> d['user']
'R2D2'
>>> d['R2D2']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'R2D2'

>>> d['pswd'] = 'thx1138'  # Assigning to an existing key   replaces its value.
>>> d
{'user': 'R2D2', 'pswd': 'thx1138'}
```

Dict: Useful Methods

```
>>> d = {'user':'R2D2', 'p':1234, 'i':34}
>>> d.keys()           # List of current keys
dict_keys(['user', 'p', 'i'])

>>> d.values()         # List of current values.
dict_values(['R2D2', 1234, 34])

>>> d.items()      # List of item tuples.
dict_items([('user', 'R2D2'), ('p', 1234), ('i', 34)])
```

Default Dictionaries and Counters

- `defaultdict` **automatically initializes nonexistent dictionary values**

```
from collections import defaultdict
d = defaultdict(str)
d['a']
"""
from collections import Counter
d = Counter()
d['a']
0
d['dog'] += 10
d['dog']
10
```

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

...
```

The first line with less indentation is outside of the function definition.

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

No declaration of types of arguments or result.

Multiple Return Values

- In Java, the only way to have a function return multiple values is using an Object that you design for the purpose.
- Python allows you to multiple values like this:

```
def describe_data(data):
    mean = ...
    median = ...
    mode = ...
    return mean, median, mode
```

- The return type is a **tuple**

No Function Overloading

- Java differentiates methods by their signature, which includes the method name and the types of its argument.
  - Java class classes can have multiple methods with the same name

    add(int, int)
    add(int, int, int)
    add(float, float)
- Python doesn't allow function overloading like Java does
  - 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 when you implement a class

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"""
    print(args)

print_everything('a', 'b', 'c’)
('a', 'b', 'c’)

lst = ['a', 'b', 'c’]
print_everything(*lst)
('a', 'b', 'c')
```

**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
    print(kwargs)

print_keyword_args(first_name="John", last_name="Doe")
{'first_name': 'John', 'last_name': 'Doe’}

my_dict = {'first_name': 'Wei', 'last_name': 'Xu’}
print_keyword_args(**my_dict)
{'first_name': 'Wei', 'last_name': 'Xu'}
```

*args and **kwargs

```
def myfun(positional, *args, **kwargs):
    print(positional)
    if args:
        print(args)
    if kwargs:
        print(kwargs)

myfun("hello", 1, 2, 3, a="hi", b="bye", c="ciao")
hello
(1, 2, 3)
{'a': 'hi', 'b': 'bye', 'c': 'ciao'}

myfun("hi", "the", "best", "food", "is", "tacos", shell="soft", meat="beef")
hi
('the', 'best', 'food', 'is', 'tacos')
{'shell': 'soft', 'meat': 'beef'}
```

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
```

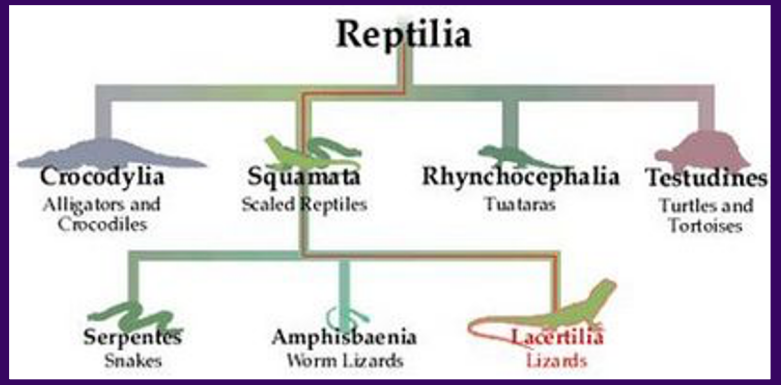
Functions are “first-class” objects

- 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)
```

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 AI_Student(Student):
    fav_class = "CIS 521”
```

- 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

```
class AI_Student(Student):
  def  __init__(self, name, dept):
      Student.__init__(self, name, dept)
```

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!
```

Magic Methods and Duck Typing

Magic Methods

```
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)

s = Student("Wei Xu", 23)
print(s)
I’m named Wei Xu – age: 23
```

Other “Magic” Methods

- *Magic Methods* allow user-defined classes to behave like built in types
- You can implement operator overloading with magic methods
  operators trigger a magic method, defined in a 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.

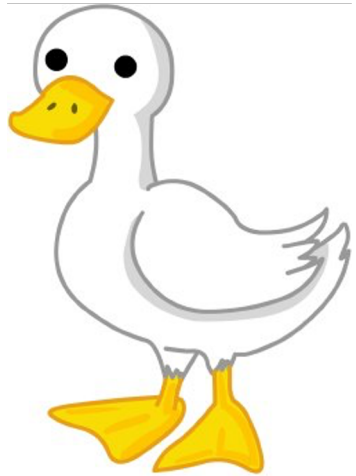
Python's main method is __main__

- In addition to magic methods, Python also has special variables.
- One is __name__ that is used with Python scripts.
- __name__ is a built-in variable which evaluates to the name of the current module (for instance, the name of the Python script).
- The interpreter sets the __name__ variable to have a value "__main__" for the source file that is being executed on the command line.
- Python doesn't have a built-in main method, so there is no `def main().`
- Instead, it uses this syntax to define a main method:

```
if __name__ == '__main__':
        # code block to be executed
```

Duck Typing

- We can call the Python function **len()** on any class that implements **`__len__`**. We don't need to implement a specific interface like we would need to do in Java.
- *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'



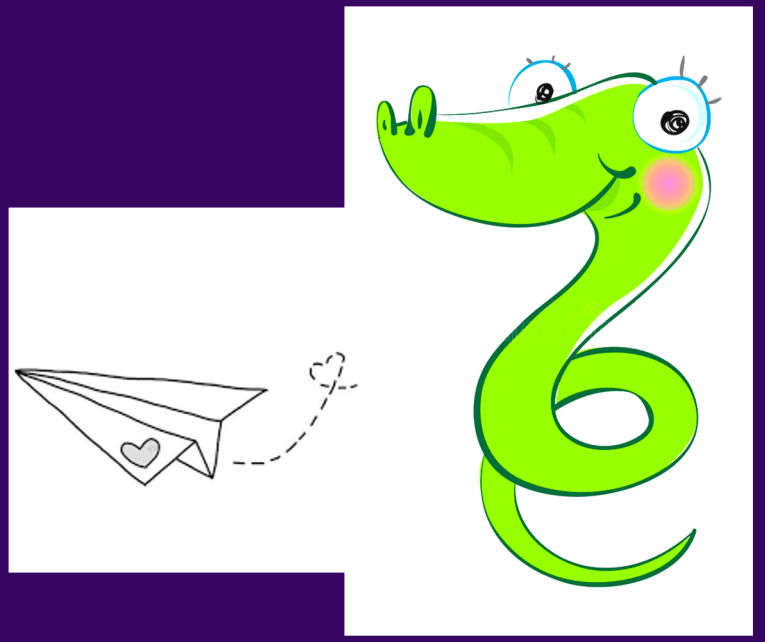
Duck Typing

```
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()
plane = Airplane()
whale = Whale()

lift_off(duck)
Duck flying
lift_off(plane)
Airplane flying
lift_off(whale)
AttributeError: 'Whale' object
  has no attribute 'fly'
```

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)
  ```
  ```

List Comprehensions replace loops!

[expression for name in list]

```
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:
    squares.append(x*x)
print(squares)
```

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

List Comprehensions replace loops!

[expression for name in list]

```
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]
```

Filtered List Comprehensions

[expression for name in list if filter]

```
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] = vs = 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

Using generators: merging sequences

- Problem: merge two sorted lists, using the output as a stream (i.e. not storing it).

```
def merge(l, r):
    llen = len(l)
    rlen = len(r)
    i = 0
    j = 0
    while i < llen or j < rlen:
        if j == rlen or (i < llen and l[i] < r[j]):
            yield l[i]
            i += 1
        else:
            yield r[j]
            j += 1
```

Using generators

```
g = merge([2,4], [1, 3, 5]) #g is an iterator
while True:
    print(g.__next__())
```

```
1
2
3
4
5
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
StopIteration
```

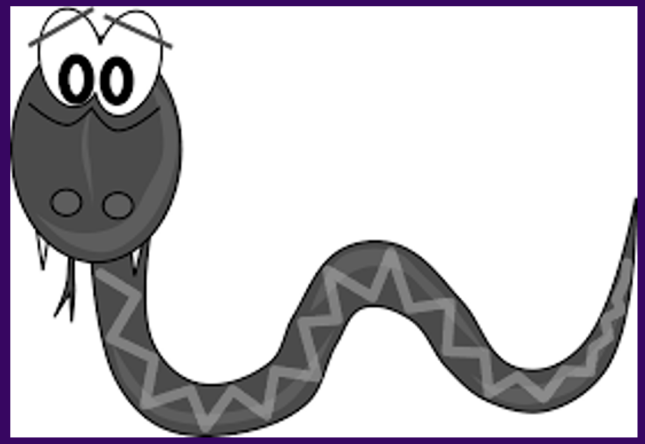
```
[x for x in merge([1,3,5],[2,4])]
[1, 2, 3, 4, 5]
```

Generators and exceptions

```
g = merge([2,4], [1, 3, 5])
while True:
    try:
        print(g.__next__())
    except StopIteration:
        print('Done’)
        break
```

```
1
2
3
4
5
Done
```

Imports



Import Modules and Files

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

from math import sqrt
sqrt(9)

# Not as good to do this:
from math import *
sqrt(9) # unclear where function defined

# create alias or nickname
import numpy as np
```

Import Modules and Files

Hint: Super useful for search algorithms

```
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
```

```
import queue as Q
q = Q.PriorityQueue()
q.put((10, "Prepare to die."))
q.put((1, "Hello,"))
q.put((5, "Diego Montoya."))
q.put((2, "My name is"))
while not q.empty():
    print(q.get()[1])
Hello,
My name is
Diego Montoya.
Prepare to die.
```

Import Modules and Files

```
# homework1.py
def concatenate(seqs):
    return [seq for seq in seqs] # This is wrong

# run python interactive interpreter (REPL) in directory of homework1.py
>>> import homework1
>>> assert homework1.concatenate([[1, 2], [3, 4]]) == \
        [1, 2, 3, 4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> import importlib          #after fixing homework1
>>> importlib.reload(homework1)
```

Tip: **importlib** is useful for reloading code from a file.

Import and pip

- **pip** is the the Package Installer for Python
- 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

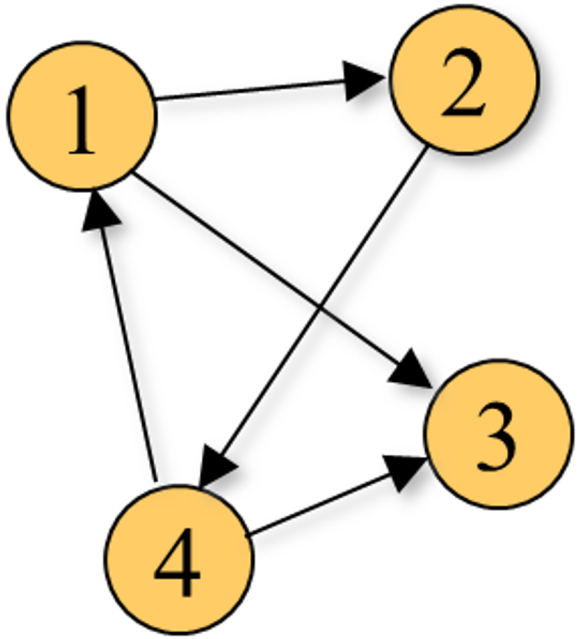
Tip: if you get a **ModuleNotFoundError**, try running **pip install module**

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

A worked example

A directed graph class

```
d = DiGraph([(1,2),(1,3),(2,4),(4,3),(4,1)])
```



Create a digraph using tuples to represent directed edges between two nodes in the graph

The DiGraph constructor

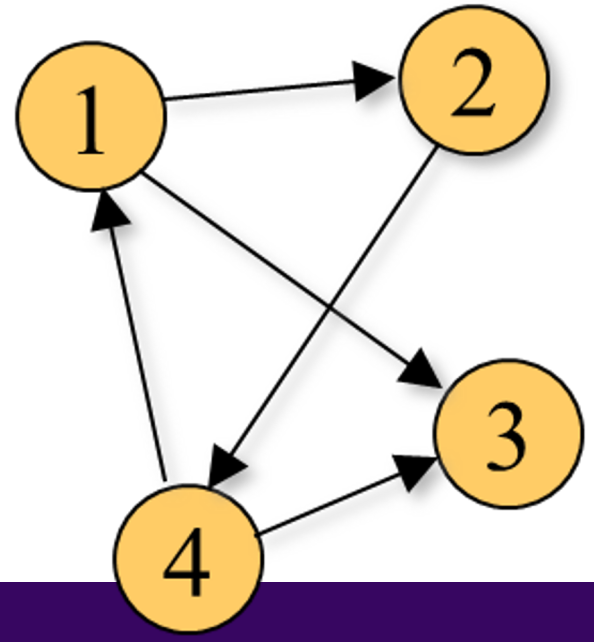
```
class DiGraph:
    def __init__(self, edges):
        self.adj = {}
        for u, v in edges:
            if u not in self.adj:
                self.adj[u] = [v]
            else:
                self.adj[u].append(v)
```

Define a class

Dictionary stores nodes as keys, value are a list of its connections

Iterate over a list

```
d = DiGraph([(1,2),(1,3),(2,4),(4,3),(4,1)])
print(d.adj)
{1: [2, 3], 2: [4], 4: [3, 1]}
```



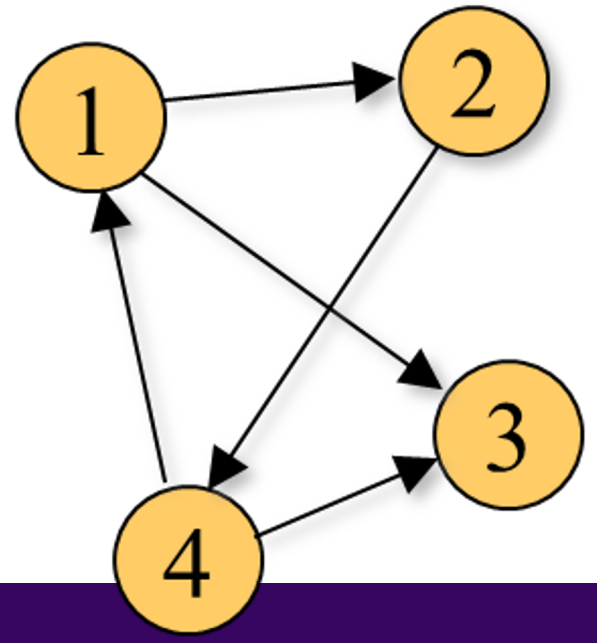
String magic method

```
class DiGraph:
    def __str__(self):
        return '\n'.join(['%s -> %s'% (u,v) \
                          for u in self.adj \
                          for v in self.adj[u]])

d = DiGraph([(1,2),(1,3),(2,4),(4,3),(4,1)])
print(d)
1 -> 2
1 -> 3
2 -> 4
4 -> 3
4 -> 1
```

Define the magic method

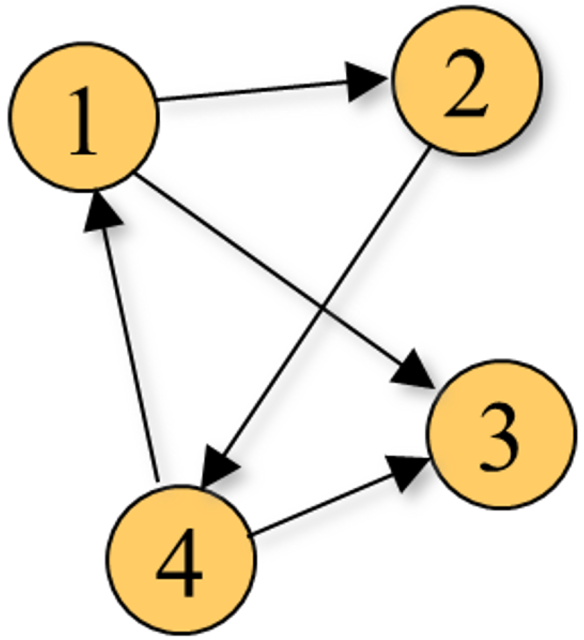
List Comprehension



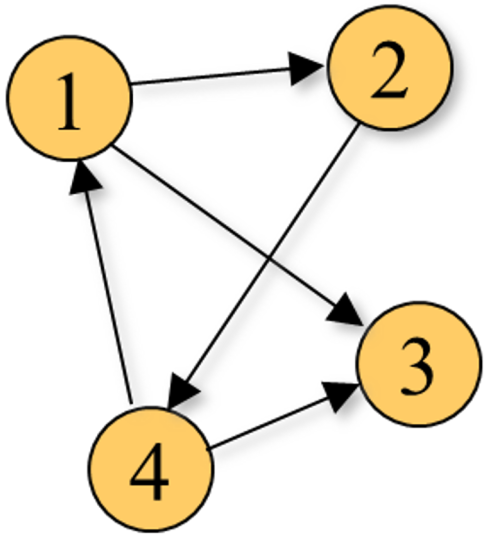
Searching a directed graph

Write a **search** function that takes in a starting node and explores all of nodes that can be reached from that node via directed edges.

- Follow each arc from tail to head
- Don’t expand any node more than once
- Return an iterator over nodes that can be reached
- Use a generator in case the graph is large



The search function




```
class DiGraph:
    def search(self, u, visited):
        # If we haven't visited this node...
        if u not in visited:
            yield u # yield it
            visited.add(u) # and remember we've visited it now.
            # Then, if there are any adjacent nodes...
            if u in self.adj:
                for v in self.adj[u]: # for each adjacent node...
                    # search for all nodes reachable from *it*...
                    for w in self.search(v, visited):
                        # and yield each one.
                        yield w
```

Use a generator

Searching a directed graph

```
d = DiGraph([(1,2),(1,3),(2,4),(4,3),(4,1)])
[v for v in d.search(1, set())]
[1, 2, 4, 3]

[v for v in d.search(4, set())]
[4, 3, 1, 2]

[v for v in d.search(2, set())]
[2, 4, 3, 1]

[v for v in d.search(3, set())]
[3]
```