Python Crash Course

Why Python?

- Python is a widely used, general purpose programming language
- Easy to start working with
- Great for scripting simple applications
- Scientific computation functionality similar to Matlab
- Plenty of libraries fully supported by the community
- Used by major deep learning frameworks such as PyTorch and Tensorflow
- Jupyter notebooks
- We will use it to program Spark applications

Common Operations

```
x = 10

y = 3

x + y

x - y

x ** y

x / y

x / float(y)

str(x) + ‘‘ + ‘‘ + str(y)
```

Common Operations

```
x = 10

y = 3

x + y                           >> 13

x - y                           >> 7

x ** y                          >> 1000

x / y                           >> 3

x / float(y)                    >> 3.33...

str(x) + ‘‘ + ‘‘ + str(y)       >> "10 + 3"
```

Common Operations

```
# Declaring two integer variable
x = 10
# Comments start with the hash symbol
y = 3
# Addition
x + y                         >> 13
# Subtraction
x - y                         >> 7
# Exponentiation
x ** y                        >> 1000
# Dividing two integers
x / y                         >> 3
# Type casting for float division
x / float(y)                  >> 3.33...
# Casting and string concatenation
str(x) + “ + “ + str(y)       >> "10 + 3"
```

Built-in Values

```
True, False

None


x = None

array = [1,2,None]


def func():
  return None


if [1,2] != [3,4]:
  print('Error!')
```

Built-in Values

```
True, False           # Usual true/false values

None                  # Represents the absence of something

                      # A valid object -- can be used like one

x = None              # Variables can be None

array = [1,2,None]    # Lists can contain None


def func():           # Functions can return None

  return None


if [1,2] != [3,4]:    # Can check for equality

  print(‘Error!’)
```

Indentation

- Code blocks are created using indents
- Indents can be tabs, 2 or 4 spaces, but should be consistent throughout the code

```
def fib(n):
    if n <= 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Indentation

- Code blocks are created using indents
- Indents can be tabs, 2 or 4 spaces, but should be consistent throughout the code

```
def fib(n):
    if n <= 1:                      # Indent level 1: function body
        return 1                    # Indent level 2: if statement body
    else:
        return fib(n-1) + fib(n-2) # Indent level 2: else statement
```

Characteristics

- **Multi-paradigm**: supports structured, object oriented and functional programming
- **Interpreted**: each statement is translated into a sequence of one or more subroutines, and then into another language (often machine code)
- **Strongly typed**: whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function
- **Dynamically typed**: the majority of its type checking is performed at run-time as opposed to at compile-time

Characteristics

- **Multi-paradigm**: you can write all programs you were used to write up to now (mostly)
- **Interpreted**: Python is first interpreted into bytecode (.pyc) and then compiled by a VM implementation into machine instructions (e.g., C)
- **Strongly typed**: Types will not be coerced silently as it happens with implicit casting in C/C++, no pointer arithmetic

- **Dynamically typed**: Variables are names for values or object references. Variables can be reassigned to values of a different type

Lists

```
# Lists are mutable arrays (think std::vector)

names = [‘Zach’, ‘Jay’]

names[0] == ‘Zach’

names.append(‘Richard’)

len(names) == 3

print(names) >> [‘Zach’, ‘Jay’, ‘Richard’]

names.extend([‘Abi’, ‘Kevin’])

print(names) >> [‘Zach’, ‘Jay’, ‘Richard’, ‘Abi’, ‘Kevin’]

names = [] # Creates an empty list

names = list() # Also creates an empty list

stuff = [1, [‘hi’,’bye’], -0.12, None] # Types can be mixed
```

List Slicing

```
# Convenient access to list elements

# Basic format: some_list[start_index:end_index]

numbers = [0, 1, 2, 3, 4, 5, 6]

numbers[0:3] == numbers[:3] == [0, 1, 2]

numbers[5:] == numbers[5:7] == [5, 6]

numbers[:] == numbers = [0, 1, 2, 3, 4, 5, 6]

numbers[-1] == 6 # Negative index wraps around

numbers[-3:] == [4, 5, 6]

numbers[3:-2] == [3, 4] # Can mix and match
```

Tuples

```
# Tuples are immutable arrays
names = (‘Zach’, ‘Jay’)  # Note the parentheses
names[0] == ‘Zach’
len(names) == 2
print(names) >> (‘Zach’, ‘Jay’)
names[0] = ‘Richard’
>> TypeError: 'tuple' object does not support item assignment
empty = tuple() # Empty tuple
single = (10,) # Single-element tuple. Comma matters!
```

Dictionaries

```
# Dictionaries are hash maps

phonebook = dict()                  # Empty dictionary

phonebook = {‘Zach’: ‘12-37’}       # Dictionary with one item

phonebook[‘Jay’] = ‘34-23’          # Add another item

print(‘Zach’ in phonebook) >> True

print(‘Kevin’ in phonebook) >> False

print(phonebook[‘Jay’]) >> ‘34-23’

del phonebook[‘Zach’]               # Delete an item

print(phonebook) >> {‘Jay’ : ‘34-23’}

for name, number in phonebook.iteritems():

    print(name, number) >> Jay 34-23
```

Loops (I)

```
for name in ['Zack', 'Jay', 'Richard']:
    print('Hi ' + name + '!')
>> Hi Zack!
>> Hi Jay!
>> Hi Richard!


while True:
    print('We are stuck in a loop...')
    break # Break out of the while loop
>> We're stuck in a loop...
```

Loops (II)

```
# What about for (i = 0; i < 10; i++)?
# Use range():
for i in range(10):          # Want an index also?
    print('Line ' + str(i)) # Look at enumerate()!


# Looping over a list, unpacking tuples:
for x, y in [(1,10), (2,20), (3,30)]:
    print(x, y)
>> 1 10
>> 2 20
>> 3 30
```

Functions

```
def my_func(param1='default'):                # No explicit return type
                                              # Parameters can have default values
    """                                       # Opening triple double quotes
    Docstring goes here.                      # Function documentation
    """                                       # Closing triple double quotes
    print(param1)                             # No value returned

my_func() >> default                          # Invocation with default parameter
my_func('new param') >> new param             # Invocation with custom parameter
my_func(param1='new param') >> new param      # Invocation with custom named parameter

def square(x):
    return x**2                               # Value returned
out = square(2)
print(out) >> 4
```

Lambda Functions

- A **lambda function** is an anonymous function that can have more than an argument but only one expression
- While normal function are defined using the `def` keyword, anonymous functions are defined using the `lambda` keyword
- **Syntax**: `lambda argument: expression`

```
my_double = lambda x: x * 2
print(my_double(5))
>> 10
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(filter(lambda x: x % 2 == 0, my_list)
print(new_list)
>> [4, 6, 8, 12]
```

Iterators (I)

- **Iterators** are like sequences (list, tuples) but...
- The entire sequence is **not materialized**
- Items are produced **one at a time** as needed
- The sequence can be unlimited
- Python **generators** create iterators
- Write a regular function and instead of calling `return` to produce a value, call `yield` instead
- When another value is needed, the generator function picks up where it left off
- Call `return` when you are done (or raise an `Exception`)

Iterators (II)

```
def g():
    x = 2
    y = 3
    yield x, y, x + y
    z = 12
    yield z/x
    yield z/y
    return
```

```
f = g
g.next()
>> 2, 3, 5
g.next()
>> 6
g.next()
>> 4
g.next()
>> Traceback (most recent call last)
    File "<stdin>", line 1, in <module>
StopIteration
```

Classes

```
class Animal(object):
    def __init__(self, species, age): # Constructor 'a = Animal(‘bird’, 10)'
        self.species = species        # Refer to instance with `self`
        self.age = age                # All instance variables are public

    def isPerson(self):               # Invoked with 'a.isPerson()'
        return self.species == ‘‘Homo Sapiens’’

    def ageOneYear(self):
        self.age += 1

class Dog(Animal):                    # Inherits Animal’s methods
    def ageOneYear(self):             # Override for dog years
        self.age += 7
```

Useful Stuff

```
st = 'hello my name is Sam'

print(st.lower()) >> 'hello my name is sam'

print(st.upper()) >> 'HELLO MY NAME IS SAM'

print(st.split()) >> ['hello', 'my', 'name', 'is', 'Sam']


tweet = 'Go Sports! #Sports'

print(tweet.split('#')) >> ['Go Sports! ', 'Sports']

print(tweet.split('#')[1]) >> 'Sports'


d = {'key1':'item1','key2':'item2'}

print(d.keys()) >> dict_keys(['key2', 'key1'])

print(d.items()) >> dict_items([('key2', 'item2'), ('key1', 'item1')])


lst = [1,2,3]

print(lst.pop()) >> 3

print(lst) >> [1, 2]


print('x' in [1,2,3]) >> False

print('x' in ['x','y','z']) >> True
```

Importing Modules

Install packages in terminal using `pip3 install [package_name]`

```
# Import ‘os’ and ‘time’ modules
import os, time

# Import under an alias
import numpy as np
np.dot(x, y) # Access components with pkg.fn

# Import specific submodules/functions
from numpy import linalg as la, dot as matrix_multiply
# Not really recommended because namespace collisions...
```

Working with files

Reading a file and storing its lines

```
filename = 'pippo.txt'
with open(filename) as file_object:
    lines = file_object.readlines()
for line in lines:
    print(line)
```

Writing to a file

```
filename = 'pippo.txt'
with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
```

Numpy

- **Optimized** library for **matrix** and **vector** computations
- Makes use of C/C++ subroutines and memory-efficient data structures
- Lots of computation can be efficiently represented as vectors

Main data type: `np.ndarray`

- This is the data type that you will use to represent matrix/vector computations.
- Note: constructor function is `np.array()`

np.ndarray

```
x = np.array([1, 2, 3])          >> [1  2  3]
y = np.array([[3, 4, 5]])        >> [[3  4  5]]
z = np.array([[6, 7], [8, 9]])   >> [[6 7]
                                     [8 9]]

print(x, y, z)

print(x.shape) >> (3,)    # a list of scalars
print(y.shape) >> (1, 3)  # a row vector
print(z.shape) >> (2, 2)  # a matrix
```

Note: Shapes `(N,) != (N, 1)`

`np.ndarray` Operations

Reductions: `np.max`, `np.min`, `np.amax`, `np.sum`, `np.mean`, ...

Always reduces along an axis! (Or will reduce along all axes if not specified)

```
x = np.array([[1, 2], [3, 4]])
print(np.max(x, axis = 1)) >> [2 4]
print(np.max(x, axis = 1, keepdims = True)) >> [[2]
                                                [4]]
```

Matrix Operations: `np.dot`, `np.linalg.norm`, `.T`, `+`, `-`, `*`, ...

Infix operators (i.e. `+`, `-`, `*`, `**`, `/`) are **element-wise**

Matrix multiplication is done with `np.dot(x, W)` or `x.dot(W)`

Transpose with `x.T`

```
print(np.array([1,2,3]).T) >> [1 2 3]
np.sum(np.array([1,2,3]), axis = 1) >> Error!
```

Indexing

```
x = np.random.random((3, 4)) # Random (3,4) matrix

x[:]                         # Selects everything in x
x[np.array([0, 2]), :]       # Selects the 0th and 2nd rows
x[1, 1:3]                    # Selects 1st row as 1-D vector
                             # and 1st through 2nd elements
x[x > 0.5]                   # Boolean indexing
```

Selecting with an `ndarray` or range will preserve the dimensions of the selection

Broadcasting

```
x = np.random.random((3, 4))          # Random (3, 4) matrix
y = np.random.random((3, 1))          # Random (3, 1) matrix
z = np.random.random((1, 4))            # Random (3,) vector

x + y                            # Adds y to each column of x
x * z         # Multiplies z element-wise with each row of x
print((y + y.T).shape)         # Can give unexpected results!
```

If you're getting an error, print the shapes of the matrices and investigate from there

Avoid cycles

Avoid explicit for-loops over indices/axes at all costs

For-loops will dramatically slow down your code (10x – 100x)

```
for i in range(x.shape[0]):
    for j in range(x.shape[1]):
        x[i,j] **= 2

for i in range(100, 1000):
    for j in range(x.shape[1]):
        x[i, j] += 5
```

Avoid cycles

Avoid explicit for-loops over indices/axes at all costs

For-loops will dramatically slow down your code (10x – 100x)

```
for i in range(x.shape[0]):
    for j in range(x.shape[1]):
        x[i,j] **= 2
```

```
for i in range(100, 1000):
    for j in range(x.shape[1]):
        x[i, j] += 5
```

Avoid cycles

Avoid explicit for-loops over indices/axes at all costs

For-loops will dramatically slow down your code (10x – 100x)

```
for i in range(x.shape[0]):
    for j in range(x.shape[1]):
        x[i,j] **= 2
```

```
x **= 2
```

```
for i in range(100, 1000):
    for j in range(x.shape[1]):
        x[i, j] += 5
```

```
x[np.arange(100,1000), :] += 5
```

List comprehension

- Similar to `map()` from functional programming languages
- Can improve readability & make the code succinct
- Format: `[func(x) for x in some_list]`
- Following are equivalent:
    - ```
      squares = []
      for i in range(10):
          squares.append(i**2)
      ```
    - `squares = [i**2 for i in range(10)]`
- Can be conditional:
    - `odds = [i**2 for i in range(10) if i % 2 == 1]`
      ```

Pythonic Syntax

- Multiple assignment / unpacking iterables
  - `x, y, z = [‘Tensorflow’, ‘PyTorch’, ‘Chainer’]`
  - `age, name, pets = 20, ‘Joy’, [‘cat’]`
- Returning multiple items from a function
  - ```
    defsome_func():
        return 10, 1
    ten, one = some_func()
    ```
- Joining list of strings with a delimiter
  - `“, ”.join([1, 2, 3]) == ‘1, 2, 3’`
- String literals with both single and double quotes
  - `message = ‘I like “single” quotes.’`
  - `reply = “I prefer ‘double’ quotes.”`
- Traversing lists in parallel with zip():
  - `s1 = {2, 3, 1}`
  - `s2 = {'b', 'a', 'c'}`
  - `print(list(zip(s1, s2))) >> [(1, 'a'), (2, 'c'), (3, 'b')]`
    ```

Interactive shell

Python has an **interactive shell** where you can execute arbitrary code

- Confused by syntax? Just try it in the shell!

```
$ python3
Python 3.6.9 (default, Nov  7 2019, 10:44:02)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 ** 5 / 2
16
>>> 2 ** (5 / 2)
4
```

- Can import any module (even custom ones in the current directory)
- Try small test cases in the shell

Online Resources

An overview of the basics of Python including variables, lists, dictionaries, functions, classes, and more can be downloaded from here:

`https://bit.ly/2RL5mjp`