CSE5AIF

Artificial Intelligence Fundamentals

Lab 1: Introduction to Python

Lab Objectives

The aim of this lab is to introduce and outline the key points of the Python programming language. You are not expected to have prior programming knowledge with python.

Upon completing this lab, you should be able to create your own python environments, develop code, and understand the basic datatypes and data structures, using this knowledge as reference for future labs.

Introduction to Python

Python is a high-level object-oriented programming language. The ideology behind python was to create a programming language focused on code readability and syntax, leading to easy to write code which reads almost like pseudo-code. Unlike many other coding languages, python code does not need to be compiled and is executed on runtime; this means that runtime errors are expected.

As a result of python's runtime nature, code typically takes longer to execute, as the interpreter must sequentially interpret code into low level machine language (as compared to other languages where this is done via a compiler before running). However, it's high readability and ease of use, has led to widespread adoption in both research and industry.

An outcome of this is a constantly growing community of developers, who create and distribute python packages. These packages can contain any variety of code for any task; they are also commonly used to interface with code written in other, faster running programming languages, providing all of the upside of python's syntax with the optimized performance of precompiled code bases.

Further information/documentation about the python programming language can be found at: https://www.python.org.

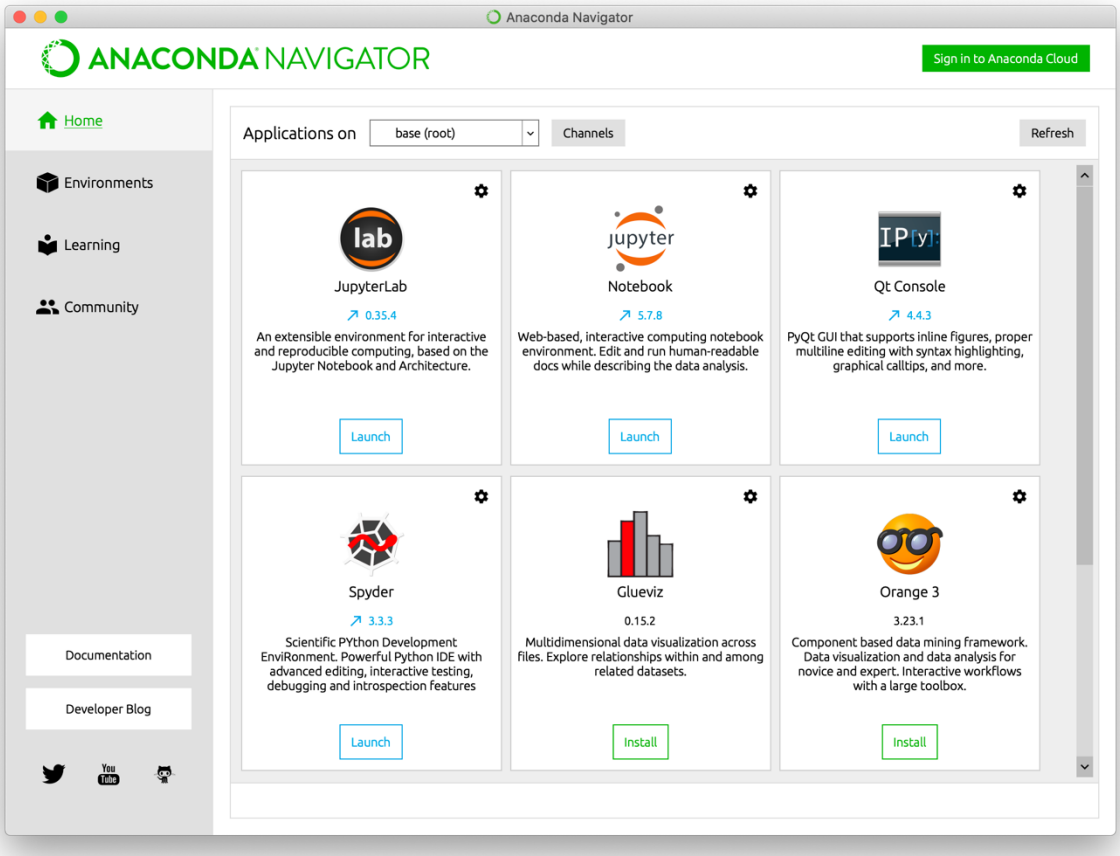
Version Control & Anaconda

Python as a language is constantly being improved and developed; compounding this with constantly updating packages, makes controlling your python and package versions paramount to ensure your code will continue to run as expected. One tool which allows for easy version control is called Anaconda.

Anaconda includes hundreds of popular data science packages, the conda package and virtual environment manager for Windows, Linux and MacOS. Conda makes it quick and easy to install, run and upgrade complex data science and machine learning environments like scikit-learn, TensorFlow and SciPy. Anaconda Distribution is the foundation of millions of data science projects as well as Amazon Web Services' Machine Learning AMIs and Anaconda for Microsoft on Azure and Windows.

Installation

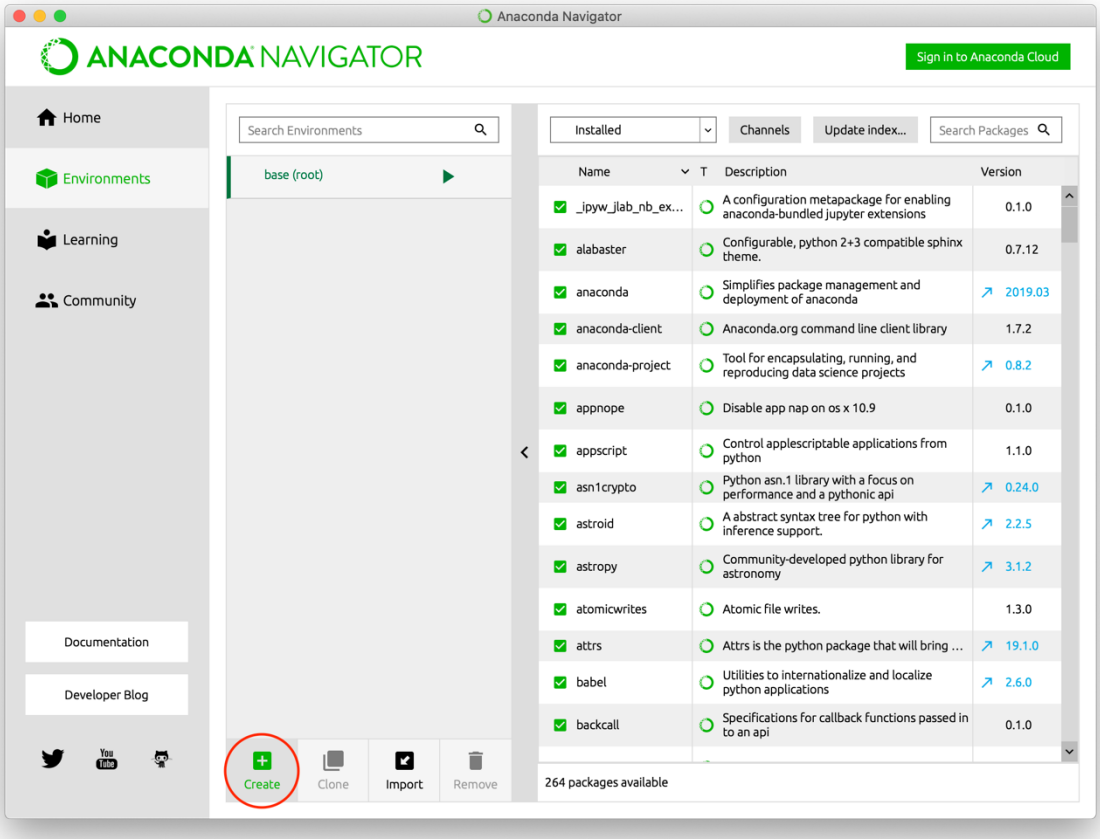
Step 1: Download Anaconda Distribution Python 3.7 64-Bit version (Windows, MacOS or Linux): https://www.anaconda.com/distribution/
Step 2: Install Anaconda with the default settings.
Step 3: Open Anaconda Navigator, you will see similar GUI as below, this means you have successfully installed Anaconda.



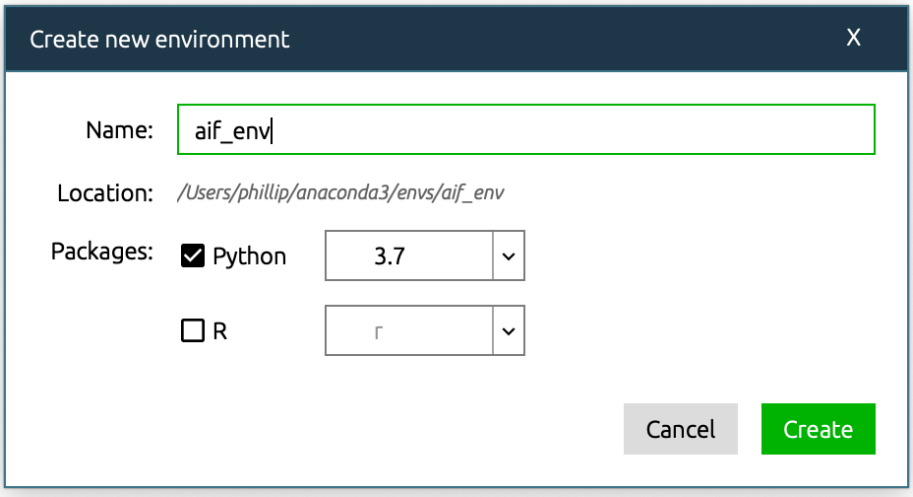
Creating an environment

Step 1: Open the **Environments** tab in the left navigation menu of Anaconda Navigator.

Step 2: Click on **Create** at the bottom of the window.



Step 3: A window will appear showing the environment creation options, enter a name for your new environment, set the python version to 3.7, then click **create**.

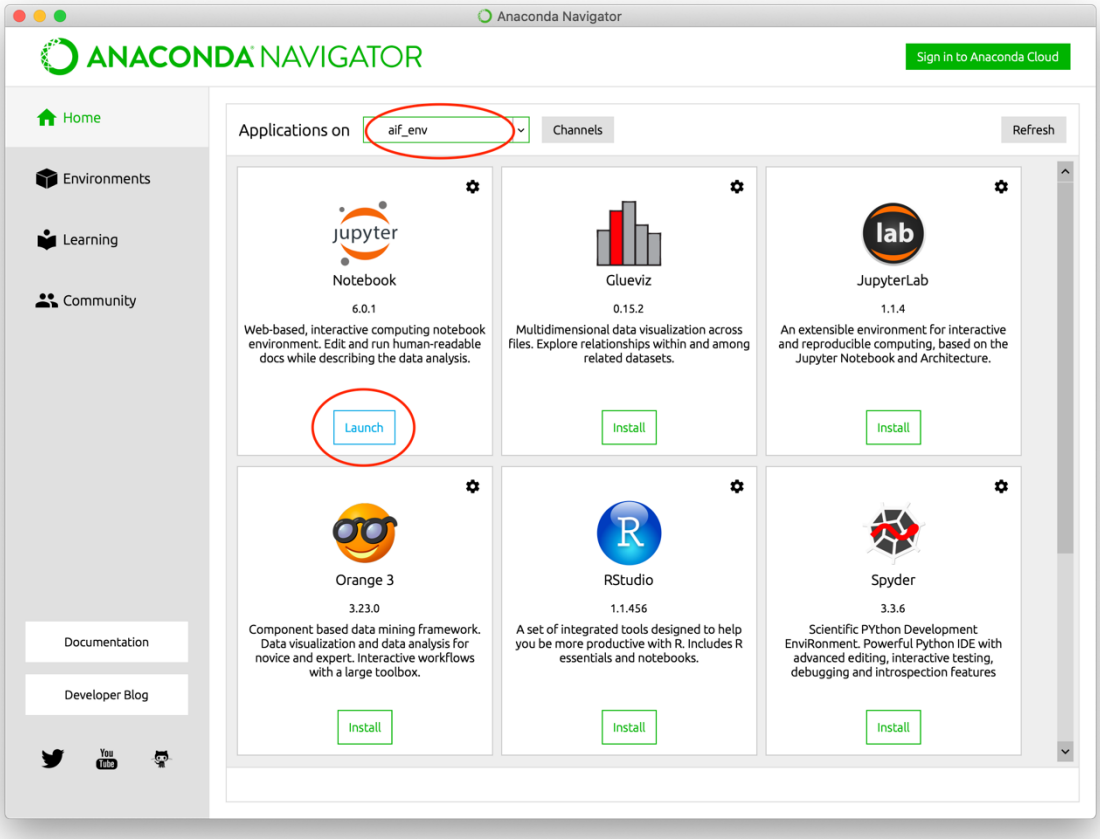


Note: Anaconda can also be utilized via command prompt using “conda” commands. These commands and a user guide can be found here.

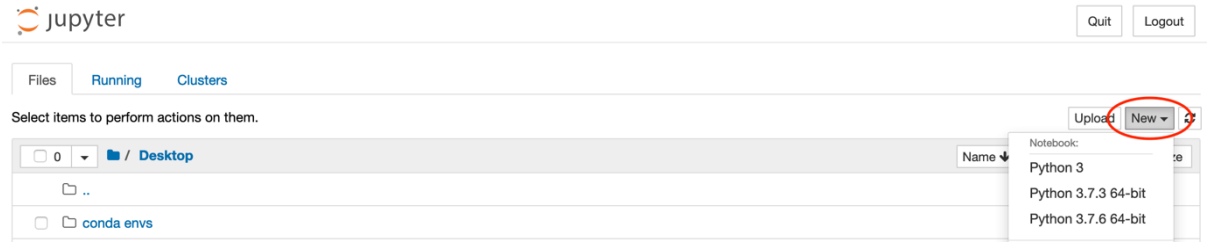
Environments and Jupyter Notebook

For this lab we will be utilizing an application called Jupyter Notebook, if you are experienced with python feel free to use any other IDE/development platform you like.

Return to the **Home** tab of the Anaconda navigator. At the top of the page click on the environment selector menu and select your environment. Now click on the **Jupyter Notebook Launch** button to begin a session.



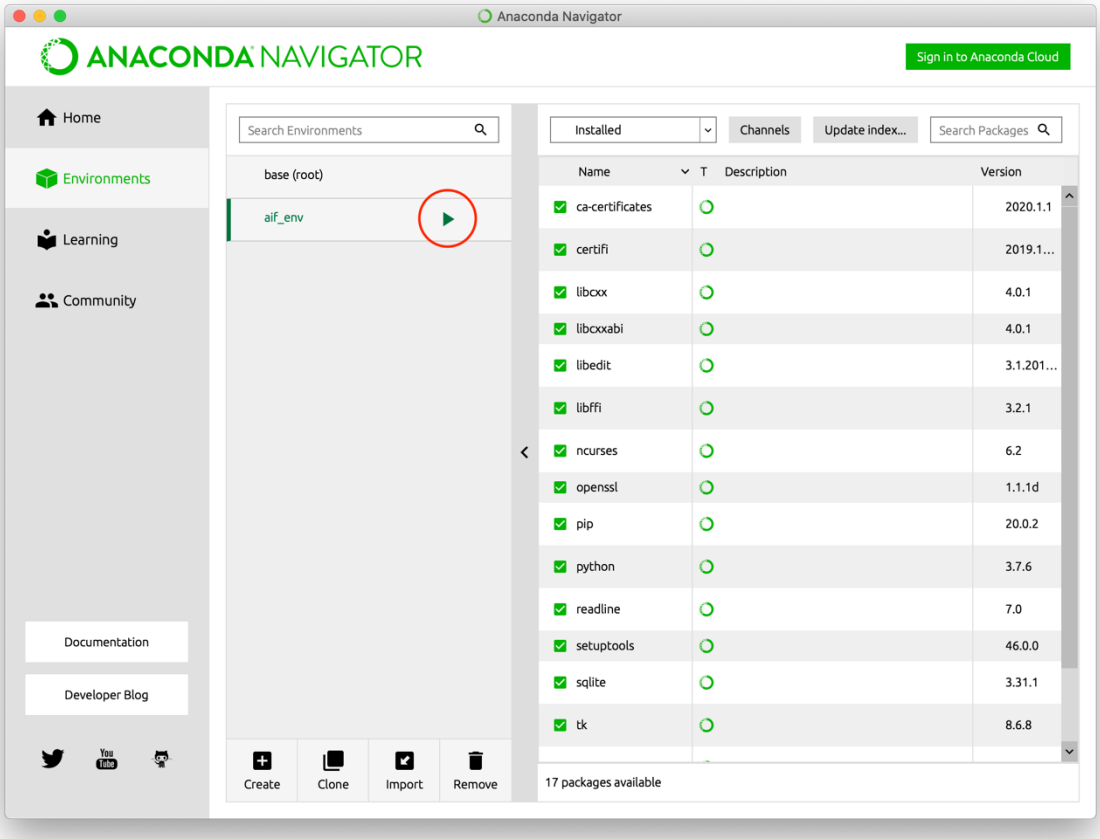
This will now start a webpage in your default browser. Select your working directory and then create a new notebook by clicking **new** and selecting your python version from the list.



Rename your notebook to **“lab_1”** by selecting **File > Rename…** in the menu bar. Jupyter Notebook consists of interactive cells, where each cell can be run individually. Outputs of code run in a cell are displayed inline allowing for easy testing and development. In the first cell enter: **print(“Hello World!”)** then select **Cell > Run Cells** in the menu bar.

All basic controls are located in the menu bar of the notebook. Shortcuts and further instructions regarding Jupyter Notebooks use can be found in the documentation.

Note: You can also access the command prompt within your Anaconda environment by clicking on the play button next to the environment label you just created and selecting “Open Command Prompt”.



Python Programming

Syntax and Indentation

Unlike many other programming languages python does not require a semicolon to denote the end of a line of code. Likewise, python also doesn't require curly braces {} to denote a block of code, instead this is achieved through indentation levels.

A code block (such as the body of a function, loop etc.) starts with indentation and ends with the first unindented line. Indentation level and use of tabs/spaces is entirely up to you but must remain consistent throughout your code.

Generally, four whitespaces are used for indentation and are preferred over tabs:

Data Types

Python contains most basic data types found in other languages; these include:

- Integers
- Floats
- Booleans
- Strings

Integers and Floats:

Note that unlike many languages, Python does **not** have unary increment (x++) or decrement (x--) operators.

```
x=3                                 # Assigns "3" to variable "x"
print(type(x))                      # Prints "<class 'int'>"
print(x)                            # Prints "3"
print(x + 1)                        # Addition; prints "4"
print(x - 1)                        # Subtraction; prints "2"
print(x * 2)                        # Multiplication; prints "6"
print(x ** 2)                       # Exponentiation; prints "9"
x += 1
print(x)                            # Prints "4"
x *= 2
print(x)                            # Prints "8"
y = 2.5
print(type(y))                      # Prints "<class 'float'>"
print(y, y + 1, y * 2, y ** 2)      # Prints "2.5 3.5 5.0 6.25"
```

Python also has built-in types for complex numbers; you can find all the details in the documentation.

Booleans:

Python implements all the usual operators for Boolean logic, but uses English words rather than symbols (&&, ||, etc.):

```
t = True
f = False
print(type(t))                    # Prints "<class 'bool'>"
print(t and f)                    # Logical AND; prints "False" # Logical OR; prints
print(t or f)                     "True"
print(not t)                      # Logical NOT; prints "False" # Logical XOR;
print(t != f)                     prints "True"
```

Strings:

Python has great support for strings:

```
hello = 'hello'                           # String literals can use single quotes
world = "world"                           # or double quotes; it does not matter.
print(hello)                              # Prints "hello"
print(len(hello))                         # String length; prints "5"
hw = hello + ' ' + world                  # String concatenation
print(hw)                                 # Prints "hello world"
hw12 = '%s %s %d' % (hello, world, 12)    # sprintf style string formatting
print(hw12)                               # prints "hello world 12"
```

You can find a list of all string methods in the documentation.

Containers

Python includes several built-in container types: lists, dictionaries, sets, and tuples.

Lists:

A list is the Python equivalent of an array, but is resizable and can contain elements of different types:

Note: python starts index from “0”, for example, xs = [3,1,2], this first element of xs is xs[0] that is number 3.

```
xs=[3,1,2]          # Create a list
print(xs, xs[2])    # Prints "[3, 1, 2] 2"
print(xs[-1])       # Negative indices count from the end of the list; prints "2"
xs[2] = 'foo'       # Lists can contain elements of different types
print(xs)           # Prints "[3, 1, 'foo']"
xs.append('bar')    # Add a new element to the end of the list
print(xs)           # Prints "[3, 1, 'foo', 'bar']"
x = xs.pop()        # Remove and return the last element of the list
print(x, xs)        # Prints "bar [3, 1, 'foo']"
```

More information on lists in the documentation.

Slicing:
In addition to accessing list elements one at a time, Python provides concise syntax to access sub-lists; this is known as slicing:

```
nums = list(range(5))  # range is a built-in function that creates a list of integers
print(nums)            # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])       # Get a slice from index 2 to 4 (exclusive); prints "[2, 3]"
print(nums[2:])        # Get a slice from index 2 to the end; prints "[2, 3, 4]"
print(nums[:2])        # Get a slice from the start to index 2 (exclusive); prints
                       "[0, 1]"
print(nums[:])         # Get a slice of the whole list; prints "[0, 1, 2, 3, 4]" #
print(nums[:-1])       Slice indices can be negative; prints "[0, 1, 2, 3]"
                       # Assign a new sublist to a slice
nums[2:4] = [8, 9]     # Prints "[0, 1, 8, 9, 4]"
print(nums)
```

Loops:
You can loop over the elements of a list like this:

```
animals = ['cat', 'dog', 'monkey'] for animal in animals:
    print(animal)
# Prints "cat", "dog", "monkey", each on its own line.
```

If you want access to the index of each element within the body of a loop, use the built-in **enumerate** function:

```
Animals = ['cat', 'dog', 'monkey'] for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: cat", "#2: dog", "#3: monkey", each on its own line
```

List comprehensions:
When programming, frequently we want to transform one type of data into another. As a simple example, consider the following code that computes square numbers:

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)              #Prints [0, 1, 4, 9, 16]
```

You can make this code simpler using a **list comprehension**:

```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares) # Prints [0, 1, 4, 9, 16]
```

List comprehensions can also contain conditions:

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares) # Prints "[0, 4, 16]"
```

Dictionaries:

A dictionary stores (key, value) pairs, like a Map in Java or an object in Javascript. You can use it like this:

```
d = {'cat': 'cute', 'dog': 'furry'}    # Create a new dictionary with some data
print(d['cat'])                        # Get an entry from a dictionary; prints "cute"
print('cat' in d)                      # Check if a dictionary has a given key; prints
                                       "True"
d['fish'] = 'wet'                      # Set an entry in a dictionary
print(d['fish'])                       # Prints "wet"
# print(d['monkey'])                   # KeyError: 'monkey' not a key of d
print(d.get('monkey', 'N/A'))          # Get an element with a default; prints "N/A"
print(d.get('fish', 'N/A'))            # Get an element with a default; prints "wet"
del d['fish']                          # Remove an element from a dictionary
print(d.get('fish', 'N/A')             # "fish" is no longer a key; prints "N/A"
```

You can find all you need to know about dictionaries in the documentation.

Loops:

It is easy to iterate over the keys in a dictionary:

```
d = {'person': 2, 'cat': 4, 'spider': 8} for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

If you want access to keys and their corresponding values, use the items method:

```
d = {'person': 2, 'cat': 4, 'spider': 8} for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

Dictionary comprehensions:
These are similar to list comprehensions but allow you to easily construct dictionaries. For example:

```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square) # Prints "{0: 0, 2: 4, 4: 16}"
```

Sets

A set is an unordered collection of distinct elements. As a simple example, consider the following:

```
animals = {'cat', 'dog'}
print('cat' in animals)    # Check if an element is in a set; prints "True"
print('fish' in animals)   # prints "False"
animals.add('fish')        # Add an element to a set
print('fish' in animals)   # Prints "True"
print(len(animals))        # Number of elements in a set; prints "3"
animals.add('cat')         # Adding an element that is already in the set does
                           nothing
print(len(animals))        # Prints "3"
animals.remove('cat')      # Remove an element from a set
print(len(animals))        # Prints "2"
```

As usual, everything you want to know about sets can be found in the documentation.

Loops:
Iterating over a set has the same syntax as iterating over a list; however, since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set:

```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: fish", "#2: dog", "#3: cat"
```

Set comprehensions:

Like lists and dictionaries, we can easily construct sets using set comprehensions:

```
from math import sqrt
nums = {int(sqrt(x)) for x in range(30)}
print(nums) # Prints "{0, 1, 2, 3, 4, 5}"
```

Tuples

A tuple is an (immutable) ordered list of values. A tuple is in many ways similar to a list; one of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot. Here is a trivial example:

```
d = {(x, x + 1): x for x in range(10)} # Create a dictionary with tuple keys t =
(5, 6)           # Create a tuple
print(type(t))    # Prints "<class 'tuple'>"
print(d[t])       # Prints "5"
print(d[(1, 2)]). # Prints "1"
```

The documentation contains more information about tuples.

Functions

Python functions are defined using the def keyword. For example:

```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'
for x in [-1, 0, 1]:
    print(sign(x))
# Prints "negative", "zero", "positive"
```

We will often define functions to take optional keyword arguments, like this:

```
def hello(name, loud=False):
    if loud:
        print('HELLO, %s!' % name.upper())
    else:
        print('Hello, %s' % name)

hello('Bob') # Prints "Hello, Bob"
hello('Fred', loud=True) # Prints "HELLO, FRED!"
```

There is a lot more information about Python functions in the documentation.

Classes

The syntax for defining classes in Python is straightforward:

```
class Greeter(object):

    # Constructor
    def __init__(self, name):
        self.name = name # Create an instance variable

    # Instance method
    def greet(self, loud=False):
        if loud:
            print('HELLO, %s!' % self.name.upper())
        else:
            print('Hello, %s' % self.name)

g = Greeter('Fred')   # Construct an instance of the Greeter class
g.greet()             # Call an instance method; prints "Hello, Fred"
g.greet(loud=True).   # Call an instance method; prints "HELLO, FRED!"
```

You can read a lot more about Python classes in the documentation.

Practice Exercises

If you are just starting out with Python and you would like to try out some of the knowledge you've learnt from this lab you can try to complete some of the following exercises.

1. Calculator

Create a simple function which enables users to enter two input numbers and an operator (+, -, *, /). Print the full equation and the result.

2. Fibonacci Sequence

Create a function which solves the Fibonacci sequence using recursion. To do this you must call the function from within itself.

3. Basic Classes

- Create a class, Triangle. It's **__init__()** method should take self, angle1, angle2, and angle3 as arguments. Make sure to set these appropriately in the body of the __init__()method.
- Create a variable named **number_of_sides** and set it equal to 3.
- Create a method named **check_angles**. The sum of a triangle's three angles is It should return True if the sum of self.angle1, self.angle2, and self.angle3 is equal 180, and False otherwise.
- Create a variable named **my_triangle** and set it equal to a new instance of your Triangle class. Pass it three angles that sum to 180 (e.g. 90, 30, 60).
- Print out my_triangle.number_of_sides and print out my_triangle.check_angles().