Programming and Python in a Nutshell

In this chapter we will go over a brief overview of programming languages and Python. While important, not having a full grasp of this chapter should not hinder your ability to program using Python.

What is a programming language?

Programming is, in essence, writing a series of instructions for the computer to execute. This is done using a programming language, which can be understood or translated to a form that can be understood by the computer.

At the lowest level the language of computers is called machine code. This language is used to communicate with the computer's CPU through binary or hexadecimal instructions. Machine code is dependent on the computer hardware being used and is not easy to understand as humans. A step up from this is an assembly language, which uses some human language, but is still difficult to understand and dependent on the computer architecture [I6][I4].

Hardware dependence could be a big problem. Programs written for one computer would not necessarily work on another computer, they would have to be translated (by a human) first. To bridge this difference between hardware specifications high level programming languages were developed.

Most of the programming languages you are likely to use these days are high-level programming languages. Besides being CPU independent, these languages are designed to be readable by humans. At some level these languages will need to be compiled (translated [I5]) to machine code [I6].

What is a Script?

A script is a text file containing source code (instructions for the computer) written in a programming language. Programs can be composed of a single script, or many scripts working together.

Generally scripts for a particular language are given a specific file suffix. Relevant to us, Python scripts end with a `.py`.

Python, a Dynamic Programming Language.

Python is a high level language and thus needs to eventually be compiled down.

Many high level programming languages' source code is compiled to machine code once, and then can be executed in this form. These are called static programming languages (C is an example).

Dynamic programming languages are languages where operations that would normally take place at compile-time (when the code is compiled) can instead be done at run-time (when the program is executed) [I2]. Python is a dynamic programming language.

When you run a python script it is compiled to byte code (if it hasn't been already). This byte code is a lower-level, platform independent representation of your source code [I3][I1].

Byte code is similar to the CPU specific assembly code, but is instead executed by software called a virtual machine (which simulates a CPU environment). [I3]. The Python Virtual Machine is always present in the Python system and is the last step of the Python Interpreter [I1].

References

[I1](1,2)
Compiling and linking in python — Net-informations.com. [Online; accessed 10-February-2020]. URL: http://net-informations.com/python/iq/linking.htm.

[I2]
Dynaimc programming language — MDN Web Docs. [Online; accessed 10-February-2020]. URL: https://developer.mozilla.org/en-US/docs/Glossary/Dynamic_programming_language.

[I3](1,2)
Ned Batchelder. Is python interpreted or compiled? yes. — Ned Batchelder. [Online; accessed 10-February-2020]. URL: https://nedbatchelder.com/blog/201803/is_python_interpreted_or_compiled_yes.html.

[I4]
Wikipedia contributors. Assembly language — Wikipedia, the free encyclopedia. [Online; accessed 10-February-2020]. URL: https://en.wikipedia.org/wiki/Assembly_language.

[I5]
Wikipedia contributors. Compiler — Wikipedia, the free encyclopedia. [Online; accessed 10-February-2020]. URL: https://en.wikipedia.org/wiki/Compiler.

[I6](1,2)
Wikipedia contributors. Machine code — Wikipedia, the free encyclopedia. [Online; accessed 10-February-2020]. URL: https://en.wikipedia.org/wiki/Machine_code.

Part

The Python Standard Library

Python Basics

In this chapter we shall discuss some of the basics of programming in Python, namely variables, operations and using functions.

Variables

Python receives information by means of variables. A variable is a dedicated piece of computer memory that holds some information. For example,

```
a = 5
```

tells python to assign 5 (an integer number) to the variable with the variable name `a`. Note that the `=` here is used for variable assignment, it does not have the same meaning as the mathematical symbol (“assign-variable-to” rather than “is-equal-to”).

If we wanted to access the value that our variable `a` holds, we can refer to it by it’s name. For example, if we want to print the value to terminal:

```
print(a)
```

```
5
```

Data Types

The information stored in memory needs to be interprated if it’s to be of any use to us. To achieve this Python (and many other programming languages) uses variable types.

In the example above we used an integer or `int` type. In order to check what type a variable has, we can use the `type()` function:

```
print(type(a))
```

```
<class 'int'>
```

The other basic variable types we will be working with are floating point numbers and strings.

Floating point numbers (or `float`) are numbers with decimal parts, for example:

```
print(type(5.2))
```

```
<class 'float'>
```

Strings (or `str`), are a collection of unicode characters (letters, numbers, symbols, ect). Basically, the contents of any text file can be seen as a string. Strings are represented using parenthesis:

```
print(type('This is a string.'))
```

```
<class 'str'>
```

You are not limited to single quotes. For single line strings you can use double quotes as well:

```
print('String using single quotes, " does not break the string.')
print("String using double quotes, ' doesn't break the string.")
```

```
String using single quotes, " does not break the string.
String using double quotes, ' doesn't break the string.
```

For strings containing line breaks, you can use `'''` or `"""`:

```
print(
'''String with a
line break'''
)

print(
"""Another string with a
line break"""
)
```

```
String with a
line break
Another string with a
line break
```

You could also insert line breaks using a the special character `'\n`

Variable Names

So far we have been using single letters (`a`, `b`, `c`, `d`, ...) as variable names, but this approach can be confusing for long segments of codes. Variable names should be as clear and descriptive as possible (describing what they are used for), while still being short enough to type out efficiently.

To this end we should delve into some of the restrictions on the character sequences that make up variable names:

- The characters must all be letters, digits, or underscores (_) , and must start with a letter. In particular, punctuation and blanks are not allowed.
- There are some words that are reserved for special use in Python. You may not use these words as your own identifiers. This is the full list:

`False`	`class`	`finally`	`is`	`return`
`None`	`continue`	`for`	`lambda`	`try`
`True`	`def`	`from`	`nonlocal`	`while`
`and`	`del`	`global`	`not`	`with`
`as`	`if`	`elif`	`or`	`yield`
`assert`	`else`	`import`	`pass`	
`break`	`except`	`in`	`raise`	

- Python is case sensitive: The variable names `last`, `LAST`, and `LaSt` are all different.

Now, you may want to use a variable that is more than one word long, for example `price at opening`, but blanks are illegal! One poor option is just leaving out the blanks, like `priceatopening`. Then it may be hard to figure out where words split. Two practical options are:

- Underscore separated: putting underscores (which are legal) in place of the blanks, like `price_at_opening`.
- Using camel-case: omitting spaces and using all lowercase, except capitalizing all words after the first, like `priceAtOpening`.
- Using Pascal-case: similar to camel-case but capitalising the first word, `PriceAtOpening`.

The standard in Python is to use underscore seperations for variable and function names.

Assigning Variables to other Variables

You can assign the value of one variable to another:

```
var1 = 3
var2 = var1

print('Variable 1 is', var1)
print('Variable 2 is', var2)
```

```
Variable 1 is 3
Variable 2 is 3
```

When you assign assign a variable using another variable, in most cases it is only the value of the variable that is assigned:

```python
var1 = 3
var2 = var1

print('Variable 1 is', var1)
print('Variable 2 is', var2)

var1 = 2

print('')
print('Variable 1 is', var1)
print('Variable 2 is', var2)
```

```
Variable 1 is 3
Variable 2 is 3

Variable 1 is 2
Variable 2 is 3
```

Notice how, even though we change the value of `var1`, the value of `var2` remains the same.

Comments

Comments make it possible to write messages in our scripts that are not to be read by the computer, but fellow humans.

In Python you can write an in-line comment by using the # symbol. Everything after this symbol until the end of the line will be considered a part of the comment and the computer will not read this as code. For example:

```
print('Not a comment') # This is a comment print('Part of the comment')
```

```
Not a comment
```

Comments can be useful for explaining what a script/section of a script does or why you've made the choices you have made in a particular line. It is not normally necessary to explain what each line of code does, as it should be easy enough to read the actual code to determine this.

Commenting on a Line of Code

If you want to comment about a particular line of code it is common practice to put the comment at the end of that line of code:

```
print('some code') #comment on code
```

```
some code
```

If the comment is too long to fit on the line, you can write the comment on a separate line above the code:

```
#Comment line that is too long to fit on the end of the line of code
print('some code')
```

```
some code
```

Commenting Out Portions of Code

Sometimes you may want to comment out code to temporarily remove it from the program without deleting it. It is especially useful when you want to isolate code snippets during debugging or print statements used in debugging during normal runtime. For example:

```
var1 = 3
var2 = var1

#print('Variable 1 is', var1)
#print('Variable 2 is', var2)

var1 = 2

#print('')
print('Variable 1 is', var1)
print('Variable 2 is', var2)
```

```
Variable 1 is 2
Variable 2 is 3
```

Type Conversion

So far we have looked at three variable types: integers, floats and strings; and how to check what type a variable is.

Sometimes we want to convert between different variable types. To do this we can use the `int`, `float` and `str` functions:

```
int_var = 1
print(int_var, type(int_var))

float_var = float(int_var)
print(float_var, type(float_var))
```

```
1 <class 'int'>
1.0 <class 'float'>
```

```
float_var = 5.7
print(float_var, type(float_var))

int_var = int(float_var)
print(int_var, type(int_var))
```

```
5.7 <class 'float'>
5 <class 'int'>
```

Note that when you convert a float to an integer Python does simply discards the decimal part (if you wish to round-off a float you can use the `round` function).

```
str_var = '1.43'
print(str_var, type(str_var))

float_var = float(str_var)
print(float_var, type(float_var))
```

```
1.43 <class 'str'>
1.43 <class 'float'>
```

```
str_var = '12'
print(str_var, type(str_var))

int_var = int(str_var)
print(int_var, type(int_var))
```

```
12 <class 'str'>
12 <class 'int'>
```

Note that anything other than a number cannot be converted from a string to a float or int:

```
str_var = 'not a number'
print(str_var, type(str_var))

float_var = float(str_var)
print(float_var, type(float_var))
```

```
not a number <class 'str'>
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-5-4224f055b9d6> in <module>
      2 print(str_var, type(str_var))
      3 
----> 4 float_var = float(str_var)
      5 print(float_var, type(float_var))

ValueError: could not convert string to float: 'not a number'
```

Even strings that contain a number with a decimal part cannot be converted to an integer:

```
str_var = '4.563'
print(str_var, type(str_var))

int_var = int(str_var)
print(int_var, type(int_var))
```

```
4.563 <class 'str'>
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-a3094efdeb44> in <module>
      2 print(str_var, type(str_var))
      3 
----> 4 int_var = int(str_var)
      5 print(int_var, type(int_var))

ValueError: invalid literal for int() with base 10: '4.563'
```

Data Structures

In this chapter we will present a brief overview of Python's standard data structures, namely tuples, lists and dictionaries. For a more broad overview you can refer to the documentation.

Tuple

Just as strings are a sequence of characters, tuples are a sequence of objects. This makes their use far more general.

You can set a tuple by separating the objects using commas. For example:

```python
t = 1, 2, 3, 'a', 'b', 'c'

print(t)
```

```
(1, 2, 3, 'a', 'b', 'c')
```

This is called tuple packing.

You can also put brackets around the objects, which is useful if you need to instance a tuple and use it in the same line (for example as a function argument):

```python
print(('a', 1, 'b', 2, 'c', 3))
```

```
('a', 1, 'b', 2, 'c', 3)
```

Like strings, tuples can be indexed and sliced:

```python
print('Index 3:', t[3])
print('Slice from index 3:', t[3:])
```

```
Index 3: a
Slice from index 3: ('a', 'b', 'c')
```

Tuples are also immutable (like strings):

```python
t[2] = 5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-5255d5d095a8> in <module>
----> 1 t[2] = 5

TypeError: 'tuple' object does not support item assignment
```

You can unpack a tuple into multiple variables, just like you can pack multiple values into a tuple:

```python
t = (1, 2, 3)
print('t is ', t)

x, y, z = t
print('x is', x)
print('y is', y)
print('z is', z)
```

```
t is  (1, 2, 3)
x is 1
y is 2
z is 3
```

Lists

Lists are used to store a collection of objects but are more flexible than tuples. You can create lists using the `list` function with another iterable object or square brackets `[]`:

```
list1 = list((1, 2, 3))
print('list1', list1)

list2 = [4, 8, 9]
print('list2', list2)
```

```
list1 [1, 2, 3]
list2 [4, 8, 9]
```

You can access elements of the list by indexing and slicing it:

```
letters = ['a', 'b', 'c', 'd', 'e']
print('Letters:', letters)
print('First character:', letters[0])
print('Second character:', letters[1])
print('Last character:', letters[-1])
print('Every second character:', letters[::2])
```

```
Letters: ['a', 'b', 'c', 'd', 'e']
First character: a
Second character: b
Last character: e
Every second character: ['a', 'c', 'e']
```

Unlike tuples you can alter the elements of a list after instancing it:

```
letters = ['a', 'b', 'c', 'd', 'e']
print(letters)

print('Changing the third character')

letters[2] = 'z'
print(letters)
```

```
['a', 'b', 'c', 'd', 'e']
Changing the third character
['a', 'b', 'z', 'd', 'e']
```

You can also assign new values to slices:

```
letters = ['a', 'b', 'c', 'd', 'e']
print(letters)

print('Changing the first three characters')
letters[:3] = ['x', 'y', 'z']
print(letters)
```

```
['a', 'b', 'c', 'd', 'e']
Changing the first three characters
['x', 'y', 'z', 'd', 'e']
```

Concatenating Lists

The `+` operator acts on lists in a similar way to strings, concatenating the two lists:

```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']

print(list1 + list2)
```

```
[1, 2, 3, 'a', 'b', 'c']
```

`list.append()`

You can add elements to the end of the list using the `.append()` method:

```
letters = ['a', 'b', 'c', 'd', 'e']
print(letters)

print('Appending an additional letter')

letters.append('f')
print(letters)
```

```
['a', 'b', 'c', 'd', 'e']
Appending an additional letter
['a', 'b', 'c', 'd', 'e', 'f']
```

`list.insert()`

If you want to insert an element into a specific place in the list you can use the `.insert()` method. This takes the index and the object you want to add as the arguments:

```
numbers = [1, 2, 4, 5, 6]
print(numbers)

print('Inserting number 3 at index 2')

numbers.insert(2, 3)
print(numbers)
```

```
[1, 2, 4, 5, 6]
Inserting number 3 at index 2
[1, 2, 3, 4, 5, 6]
```

`lists.remove()`

If you want to remove the first instance of an element of a list with a specific value you can use the `.remove()` method:

```
numbers = [1, 2, 1, 3, 4]
print(numbers)

print('Removing first 1 from numbers')

numbers.remove(1)
print(numbers)
```

```
[1, 2, 1, 3, 4]
Removing first 1 from numbers
[2, 1, 3, 4]
```

`list.pop()`

If you want to retrieve and remove an element at a particular index you can use the `.remove()` method, which takes the index of the element you want to retrieve as the argument:

```
numbers = [1, 2, 3, 4, 5]
print(numbers)

print('Retrieving number at index 2:', numbers.pop(2))

print(numbers)
```

```
[1, 2, 3, 4, 5]
Retrieving number at index 2: 3
[1, 2, 4, 5]
```

Dictionaries

So far we have only looked at sequence data structures, where elements are referred to by their position in the sequence. In dictionaries, however, the objects stored are referred to by a key. Keys must be an immutable type, for example a string, number or tuple containing only immutable types.

You can create a dictionary using the `dict` function; and assign values using the subscript notation:

```
dictionary[key] = value
```

```
d = dict()

d[1] = 'a'
d['lst'] = [1, 2, 3]

print(d)
```

```
{1: 'a', 'lst': [1, 2, 3]}
```

You can also access dictionary values using the subscript notation:

```
print(d[1])
```

```
a
```

An alternative way to initialize a dictionary with key-value pairs is:

```
{key1 : value1, key2 : value2}
```

much like it appears in the print output:

```
d = {1 : 'a',  'lst' : [1, 2, 3]}

print(d)
```

```
{1: 'a', 'lst': [1, 2, 3]}
```

Note that using a key that doesn't exist in the dictionary will give you a `KeyError`:

```
print(d[2])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-5-c8f93a31d4a2> in <module>
----> 1 print(d[2])

KeyError: 2
```

Listing the Keys Which Exist

Often you will want a list of the keys which a dictionary has. For this you can use the `dict.keys()` method:

```
print(d.keys())
```

```
dict_keys([1, 'lst'])
```

One use for this is to check if a dictionary has the key you're looking for if you want to avoid an error:

```python
if 2 in d.keys():
    print(d[2])
else:
    print(2, 'not a key in d')
```

```
2 not a key in d
```

Part

Scientific Packages

Numpy

The NumPy package provides us with arrays and matrices (efficient data structures), special functions, random number generators, and more.

The documentation for the SciPy, NumPy and many other scientific packages can be found here: https://www.scipy.org.

Importing NumPy

The standard way to import NumPy is using the alternative name `np`:

```
import numpy as np
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-1-0aa0b027fcb6> in <module>
----> 1 import numpy as np

ModuleNotFoundError: No module named 'numpy'
```

Arrays

Arrays are one of NumPy's most important objects.

An array is a sequence of homogeneous data (each element must be the same data type). NumPy arrays use NumPy specific data types which are listed here.

Though we shall see that arrays can be indexed and sliced similarly to strings, tuples and lists, they behave differently under operations.

Arrays can have any number of dimensions. In this section we will only consider the 1 dimensional case.

Contents

Creating Arrays

Arrays can be created using the `np.array()` function with a list, tuple or another array as the argument:

```
#Array of integers
np.array([1, 2, 3, 4])
```

```
array([1, 2, 3, 4])
```

```
#Array pf strings
np.array(('a', 'b', 'c'))
```

```
array(['a', 'b', 'c'], dtype='<U1')
```

Remember that arrays are homogeneous:

```
#Trying to create an array with different types
np.array([1, 2.3, 'x'])
```

```
array(['1', '2.3', 'x'], dtype='<U32')
```

Indexing and Slicing

As said before, arrays can be indexed and sliced similarly to lists and strings

```
letters = np.array(['a', 'b', 'c', 'd', 'e'])
print('Letters:', letters)
print('First character:', letters[0])
print('Second character:', letters[1])
print('Last character:', letters[-1])
print('Every second character:', letters[::2])
```

```
Letters: ['a' 'b' 'c' 'd' 'e']
First character: a
Second character: b
Last character: e
Every second character: ['a' 'c' 'e']
```

Mutable But Tricky To Resize

Similarly to lists, arrays are mutable (you can change the array after initializing it). For example, you can change an element of an array:

```
arr = np.array((1, 2, 3 ,4))
print('Array:', arr)

print('')
print('Changing element 2')
print('')

arr[2] = 7
print('Array:', arr)
```

```
Array: [1 2 3 4]

Changing element 2

Array: [1 2 7 4]
```

However, unlike lists, it's not easy or efficient to alter the size of an array. It is still possible to resize (with `np.resize()`) and to concatenate (with `np.concatenate()`) arrays, but they don't have certain handy functions for lists like `.append()` and `.insert()`.

In general you should only create an array once you know how big it needs to be. If you need to add elements to an array, consider starting with a list and converting that to an array when you need the array properties.

Iterating Through Arrays

Like strings, tuples and lists, arrays are iterable:

```
arr = np.array([1, 2, 3, 4])

for a in arr:
    print(a)
```

```
1
2
3
4
```

Vectorized Operations

One of the most useful properties of NumPy arrays is their vectorized operations. That is arithmetic operations between an array and array, and an array and scalar are performed element by element.

For example consider the scalar operations:

```
2*np.array([1, 2, 3, 4])
```

```
array([2, 4, 6, 8])
```

```
np.array([1, 4, 5]) + 1
```

```
array([2, 5, 6])
```

Array on array operations are also performed element by element:

```
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([2, 4, 6, 8])

print(arr2, '-', arr1, 'is', arr2 - arr1)
print(arr2, '/', arr1, 'is', arr2/arr1)
```

```
[2 4 6 8] - [1 2 3 4] is [1 2 3 4]
[2 4 6 8] / [1 2 3 4] is [2. 2. 2. 2.]
```

These vectorized operations are far more efficient than iterating through the arrays and operating on each element individually, i.e.

```
#More efficient:
print(arr1, '+', arr2, 'is', arr1 + arr2)
```

```
[1 2 3 4] + [2 4 6 8] is [ 3  6  9 12]
```

```
#Less efficient
arr3 = np.array(4*[0])

for  i in range(4):
    arr3[i] = arr1[i] + arr2[i]

print(arr1, '+', arr2, 'is', arr3)
```

```
[1 2 3 4] + [2 4 6 8] is [ 3  6  9 12]
```

Creating Structured Arrays

Often we would like to create a large array with a particular structure. We could create these arrays from lists using list comprehension, but NumPy provides some useful built in functions to use instead.

`np.arange()`

This function is analogous to the `range()` function. It produces a series of values where you can specify the starting value, stopping value and the step size.

The syntax is:

```
np.arange(start, stop, step)
```

Similar to the `range()` function, you can use 1, 2 or 3 arguments:

```
#1 argument: stop
np.arange(5)
```

```
array([0, 1, 2, 3, 4])
```

```
#2 arguments: start, stop
np.arange(1, 5)
```

```
array([1, 2, 3, 4])
```

```
#3 arguments: start, stop, step
np.arange(1, 10 ,2)
```

```
array([1, 3, 5, 7, 9])
```

Unlike the `range()` function. `np.arange()` also allows for floating point values:

```
np.arange(2.3, 3, 0.1)
```

```
array([2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3. ])
```

`np.linspace()`

This function creates a series of evenly spaced values between a stopping and starting value. The number of items in the array can also be specified.

The syntax:

```
np.linspace(start, stop, number)
```

If `number` is not specified an array of length 50 is created.

```
np.linspace(0, 1, 10)
```

```
array([0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
       0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])
```

`np.zeroes()`

This function creates a uniform array of zeros. It takes the shape of the array you want to generate as an argument.

```
np.zeros(shape)
```

For a one dimensional array `shape` is just the size of the array:

```
np.zeros(5)
```

```
array([0., 0., 0., 0., 0.])
```

`np.zeros()` can be useful if you wish to create an array with a particular size, but will only be filling in the values later.

`np.ones()`

`np.ones()` is similar to `np.zeros()`, except it generates a uniform array of ones.

```
np.ones(7)
```

```
array([1., 1., 1., 1., 1., 1., 1.])
```

Note that, if you want a uniform array of a different value, you can either add that value to an array of zeros or multiply that value with an array of ones.

Array Methods and Attributes

In this section we will look at some methods and attributes that arrays have. This is not a complete list, but rather highlighting things you may find useful.

Let's start off by creating a fairly large array, for example a collection of human height measurements:

```
heights = np.array(
      [ 2.13159377,  1.8864508 ,  1.63504183,  1.71173878,  1.78826872,
        1.60621813,  1.74630706,  2.11123384,  1.54212979,  1.39184441,
        1.7919224 ,  1.80299245,  1.73770464,  1.95233673,  1.47179093,
        1.70506609,  1.41194434,  2.05643464,  1.8262583 ,  1.47764985,
        1.61362183,  1.65600316,  1.42078883,  1.78059602,  1.80600655,
        1.91634004,  1.82746488,  1.82688072,  1.82053352,  1.84882458,
        1.80672297,  1.4646136 ,  1.71033286,  1.83272236,  1.97074545,
        1.96265325,  1.39817665,  1.55933323,  1.59111903,  1.53108805,
        1.33635392,  1.74971951,  1.56885338,  1.6614742 ,  1.70868504,
        1.58476337,  1.69233894,  1.73520641,  1.71248418,  1.75484377])
```

To get the number of elements in an array, we can use the `size` attribute:

```
print('The size of the heights array:', heights.size)
```

```
The size of the heights array: 50
```

For 1 dimensional arrays this is gives us the same value as using `len()`, but for multidimensional arrays, `len()` will not return the total number of elements.

Minimum and Maximum Values

You can use the `min()` and `max()` methods to get the minimum and maximum values of an array respectively.

```
print('Minimum height:', heights.min())
print('Maximum height', heights.max())
```

```
Minimum height: 1.33635392
Maximum height 2.13159377
```

Again, this gives you similar results to the functions in the Standard Library, but is the only option for arrays of higher dimensions.

Statistical Functions

NumPy provides us with some basic statistical functions out of the box. For example the `mean()` (arithmetic mean or average) and `std()` (standard deviation).

```
print('Average height: ', heights.mean())
print('Standard deviation of heights: ', heights.std())
```

```
Average height:  1.712684356
Standard deviation of heights:  0.18476698650385862
```

```
print('Average height:', np.mean(heights))
print('Standard deviation of heights:', np.std(heights))
print('Maximum height:', np.max(heights))
print('Mimimum height:', np.min(heights))
```

```
Average height: 1.712684356
Standard deviation of heights: 0.18476698650385862
Maximum height: 2.13159377
Mimimum height: 1.33635392
```

Array Conditional Statements and `numpy.where()`

Comparison and Bitwise Operations on Arrays

We can apply comparison operators to arrays:

```
a1 = np.array([1,  2, 3, 4, 5])

a2 = np.array([2, 1, 5, 6, 4])

a1 < a2
```

```
array([ True, False,  True,  True, False])
```

As you can see this gives us an array of booleans, each element representing the outcome of comparing the corresponding element of `a1` to `a2`.

What if we wanted to combine the boolean arrays with a logical operator? For example, if we want an array of booleans for the condition `a1` is less-than `a2` and greater than `2`. Unfortunately the boolean comparison operators we used in Comparison Operators won't work, for example using `and`:

```
a1 < a2 and a1 > 2
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-3-74cb81b02f02> in <module>
----> 1 a1 < a2 and a1 > 2

ValueError: The truth value of an array with more than one element is ambiguous. Use
a.any() or a.all()
```

In order to combine boolean arrays (without a loop) we need to use **bitwise** operators.

Bitwise operators treat numbers as a string of bits and act on them bit by bit. In the case of a boolean array, the operator acts on it element by element. The bitwise operators we are interested are:

Operator	Name	Analogous boolean operator
`&`	Bitwise and	`and`
`|`	Bitwise or	`or`
`~`	Bitwise complement	`not`

(See https://wiki.python.org/moin/BitwiseOperators for a more comprehensive list and explanation of bitwise operations.)

Returning to our original example:

```
(a1 < a2) & (a1 > 2)
```

```
array([False, False,  True,  True, False])
```

Note that the comparisons must be grouped in brackets for this to work:

```
a1 < a2 & a1 > 2
```

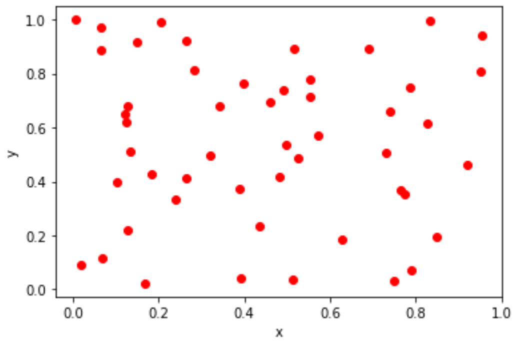
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-5-c3606bc24b97> in <module>
----> 1 a1 < a2 & a1 > 2

ValueError: The truth value of an array with more than one element is ambiguous. Use
a.any() or a.all()
```

Example - Random Points in a Region

We can use `np.where()` to check which points in an array lie inside or outside of region.

First let's generate an array of 50 random points in 2D space:

```
points = np.random.random((2, 50))

plt.plot(points[0, :], points[1, :], 'ro')

plt.xlabel('x')
plt.ylabel('y')

plt.show()
```



Note that axis 0 of `points` is used to represent the x and y values, and axis 1 represents points. i.e. for the points $(x_0, y_0), (x_1, y_1), \ldots, (x_{49}, y_{49})$, `points` is:

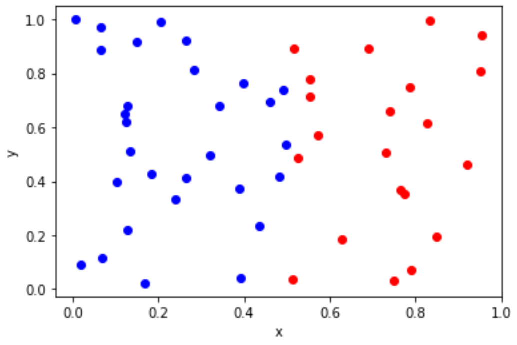
x0 x1 x2 x3 x4 ... x48 x49

y0 y1 y2 y3 y4 ... y48 y49

Now, let's plot the points which lie to the left of 0.5 as blue and the others as red:

```
is_left = points[0, :] < 0.5 #True where left of 0.5

plt.plot(points[0, is_left], points[1, is_left], 'bo')
plt.plot(points[0, ~ (is_left)], points[1, ~ is_left], 'ro')

plt.xlabel('x')
plt.ylabel('y')
```

```
Text(0, 0.5, 'y')
```

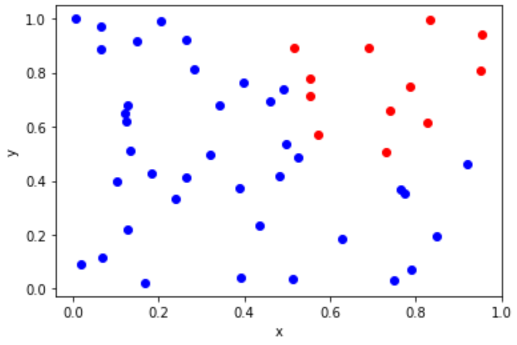


Note that, in the example above, we have used an array of booleans to **slice the elements of the array which are true**. We have also use the **bitwise compliment** to get the complement of our comparison result, there is no need to recalculate it.

Now, lets plot the points right of 0.5 and above 0.5 (the top left square) as red and the rest as blue (remember the **bitwise and**):

```
#True if in top left square
is_top_left = (points[0, :] > 0.5) & (points[1, :] > 0.5)

plt.plot(points[0, is_top_left], points[1, is_top_left], 'ro')
plt.plot(points[0, ~ (is_top_left)], points[1, ~ is_top_left], 'bo')

plt.xlabel('x')
plt.ylabel('y')
```

```
Text(0, 0.5, 'y')
```



`numpy.where()`

Documentation

```
numpy.where(condition[,x, y])
```

Returns elements chosen from `x` or `y` depending on the condition. If no `x` or `y` arguments are provided it returns and array of indices.

```
arr = np.arange(10, 20)

arr_where = np.where(arr > 15)

print('arr1:', arr)
print('Indices where arr1 is greater than 15:', arr_where)
print('The sub-array of arr1 that is greater than 15:', arr[arr_where])
```

```
arr1: [10 11 12 13 14 15 16 17 18 19]
Indices where arr1 is greater than 15: (array([6, 7, 8, 9]),)
The sub-array of arr1 that is greater than 15: [16 17 18 19]
```

If both `x` and `y` is specified, the elements of the returned array come from `x` if `condition` is true, or from `y` if `condition` is false.

```
x = np.linspace(1, 5, 5)
#y = np.linspace(-5, -1, 5)
y = -x

condition = [True, False, True, True, False]

print('x:', x)
print('y:', y)
print('Condition:', condition)
print('x where True, y where False:', np.where(condition, x, y))
```

```
x: [1. 2. 3. 4. 5.]
y: [-1. -2. -3. -4. -5.]
Condition: [True, False, True, True, False]
x where True, y where False: [ 1. -2.  3.  4. -5.]
```

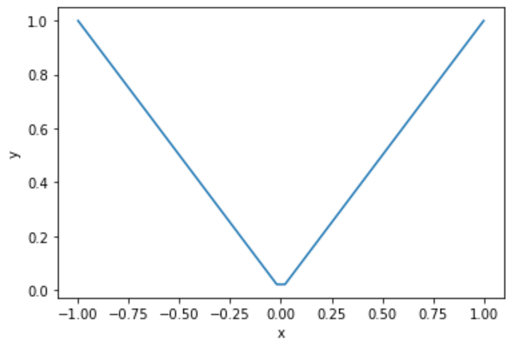
Example - Piecewise defined functions

One use for `np.where()` is to define a piecewise defined function that works on arrays.

As a first example, let's use `np.where()` to plot the absolute value function (you should really use `np.abs()` for this):

$$y = \begin{cases} -x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

```
x = np.linspace(-1, 1)

y = np.where(x >= 0, x, -x)

plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



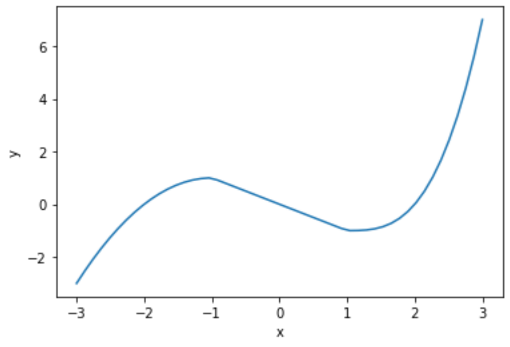
Note that, in the plot above, the line does not reach zero, but flattens out to a value above it. This is because the array `x` does not contain the value 0, but values around it.

Now, consider the piecewise function:

$$f(x) = \begin{cases} -(x+1)^2 + 1 & \text{if } x < -1 \\ -x & \text{if } -1 \leq x \geq 1 \\ (x-1)^3 - 1 & \text{if } x > 1 \end{cases}$$

where there are three regions. To handle this we can use 2 `np.where()` calls:

```
x = np.linspace(-3, 3)

#Left condition
y = np.where(x < -1, -(x+1)**2 + 1, -x)

#Right condition
y = np.where(x > 1, (x - 1)**3 - 1, y)

plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('y')

plt.show()
```



Matplotlib

In this chapter we shall take a quick look at plotting with Matplotlib's Pyplot module. Matplotlib offers many plotting functions and plots have many features that can be tweaked. For these reasons we will only be scratching the surface of using Matplotlib. A good resource for finding out what is possible is the Matplotlib Thumbnail Gallery which features many example plots along with their source code. The matplotlib documentation can be found here.

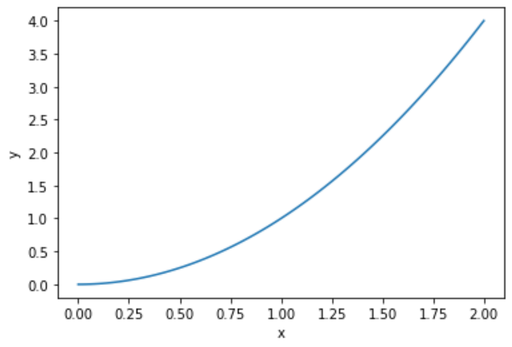
Simple Plots with Pyplot

The Pyplot module of Matplotlib acts as an interface to the Matplotlib package. This gives us access to a library of 2-dimensional plotting functions. The standard way of importing Pyplot is:

```
import matplotlib.pyplot as plt
```

As a first example, let's plot the line $y = x^2$:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2, 100)
y = x*x

plt.plot(x, y) #Plots a line

plt.xlabel('x') #x-axis label
plt.ylabel('y')

plt.show() #Visualizes the plot
```



where:

- `plt.plot()` plots a straight line, which is one of the many types of plots available in the module (see the Thumbnail Gallery for more).
- The `plt.xlabel()` and `plt.ylabel()` functions set the labels for the x and y-axis of the plot to the given arguments respectively.
- `plt.show()` shows the current figure (discussed in the following section). In the regular Python environment this function will pause the code and bring up a window containing the plot. Elements of the plot can be edited in this window and this plot can be saved. Closing the window resumes the script.
  In Jupyter Notebook, running `plt.show()` will display the plot in the cell output and will not pause the script.

Figures

A Matplotlib figure contains plot elements, for example a set of (or multiple sets of) axis, a title etc. Figures can be created using

```
fig = plt.figure()
```

When using `plt.plot()` Matplotlib will automatically add the plot to the last figure that was defined. Refer to the Subplots page for accessing the figure axis directly.

If you want to specify the dimensions of the plot, you can create a figure with the first positional or keyword argument:

```
fig = plt.figure(figsize = (width, height) )
```

where `figsize` (a 2-tuple of width and height) is in inches.

For more information on the figure class see the documentation.

Saving Figures

You can save figures using the

```
plt.savefig(filename)
```

function, where `filename` is the filename of the image to be saved. If a file extension is specified, the image will be saved using that type, the default type is a PNG.

This will save the current figure, if you want to save a particular figure then you can use `fig.savefig()`.

If you're not specifying the figure, make sure to save **before** you call `plt.show()` as this will clear the figure.

Line Color

You can specify the line color for the plot using either a positional (single letter) argument:

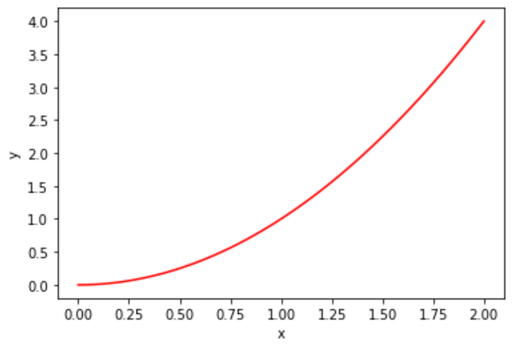
```
plt.plot(x, y, 'r')
```

or using a keyword argument:

```
plt.plot(x, y, color = 'red')
```

where the examples above both produce red lines, for example:

```
x = np.linspace(0, 2, 100)
y = x*x

plt.plot(x, y, 'r')

plt.xlabel('x')
plt.ylabel('y')

plt.show()
```



The list of colors, as found in the Matplotlib documentation, is:

Single Letter	Full Name
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

Shades of gray can be given as a string representation of a float between 0 and 1, for example:

```
color = '0.75'
```

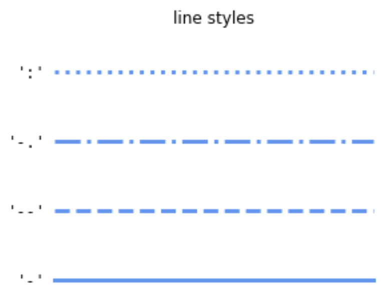
Line Style

Similar to the color of the plot, you can also set the line style, either as a positional argument:

or as a keyword argument:

Note that both the color and line style can be combined when set using the positional argument.

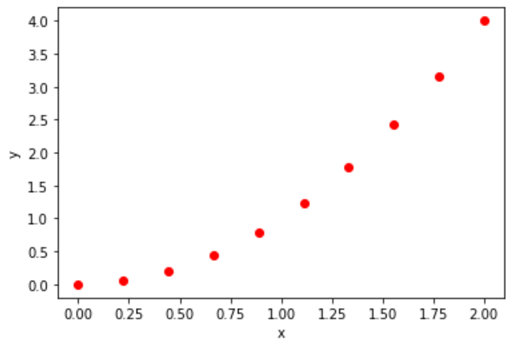
The reference for the lines given below is taken from the documentation:



Marker

In addition to line style and color, you can specify a marker. The markers are placed at each data point. The possible markers are listed in the documentation, as an example let's plot the data points as circles (`'o'` in the positional argument, or `marker = 'o'` as a keyword argument):

```python
x = np.linspace(0, 2, 10)
y = x*x

plt.plot(x, y, 'ro')

plt.xlabel('x')
plt.ylabel('y')

plt.show()
```

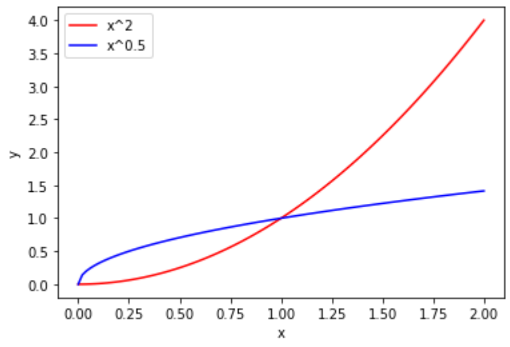


As you can see the line style is set to `'None'` by default if a marker is specified without a line style.

Legends

You can add a legend to your figure by labeling the plots with the keyword argument `label` and calling the `plt.legend()` function:

```python
x = np.linspace(0, 2, 100)

plt.plot(x, x*x, 'r', label = 'x^2')

plt.plot(x, np.sqrt(x), 'b', label = 'x^0.5')

plt.xlabel('x')
plt.ylabel('y')

plt.legend()

plt.show()
```



Subplots

You can create subplots in two different ways:

`fig.add_subplot()`

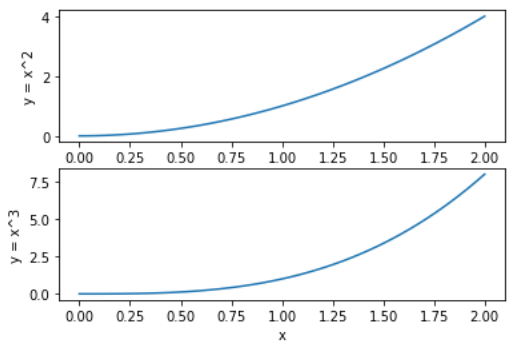
One way to add subplots is by creating a figure and calling the `fig.add_subplot()` method to add an axis to it with (one of) the call signature:

```
fig.add_subplot(nrows, ncols, index)
```

where `nrows` and `ncols` are the total number of rows and columns of axis and `index` is the position on the grid of axis.

Consider the plot with two rows and a single column:

```
x = np.linspace(0, 2)


fig = plt.figure()

#Top axis
ax0 = fig.add_subplot(2, 1, 1)
ax0.plot(x , x**2)
ax0.set_xlabel('x') #Note `set_xlabel` instead of `xlabel`
ax0.set_ylabel('y = x^2')

#Bottom axis
ax1 = fig.add_subplot(2, 1, 2)
ax1.plot(x, x*x*x)
ax1.set_xlabel('x')
ax1.set_ylabel('y = x^3')

plt.show()
```



Refer to the documentation for additional options.

`plt.subplots()`

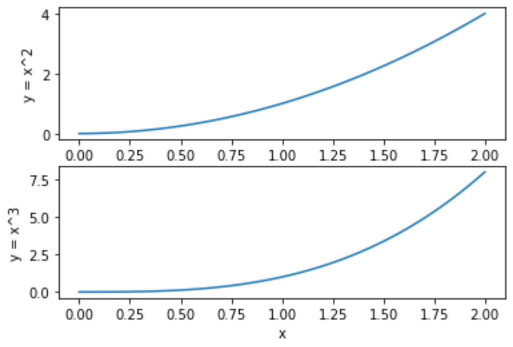
An alternative way to create subplots is to use the `plt.subplots()` function which returns the figure object and a tuple of axis. The call signature is:

```
plt.subplots(nrows = 1, ncols = 1)
```

where `nrows` and `ncols` are the number of rows an columns as before.

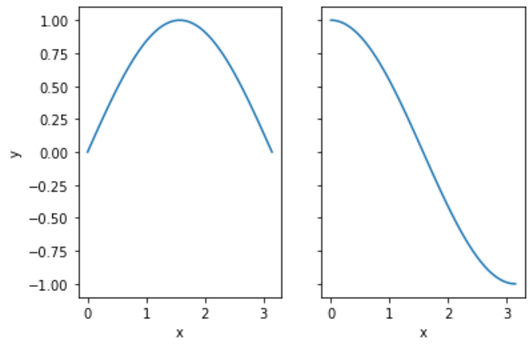
Let's recreate the previous plot using this function:

```
x = np.linspace(0, 2)

fig, ax = plt.subplots(2, 1)

#Top axis
ax[0].plot(x , x**2)
ax[0].set_xlabel('x') #Note `set_xlabel` instead of `xlabel`
ax[0].set_ylabel('y = x^2')

#Bottom axis
ax[1].plot(x, x*x*x)
ax[1].set_xlabel('x')
ax[1].set_ylabel('y = x^3')

plt.show()
```



A couple of additional keyword arguments are `sharex` and `sharey`. These take boolean values. If true the subplots will share the relevant axis's ticks. For example:

```
x = np.linspace(0, np.pi)

fig, ax = plt.subplots(1, 2, sharey = True)

ax[0].plot(x, np.sin(x))
ax[0].set_xlabel('x')

ax[1].plot(x, np.cos(x))
ax[1].set_xlabel('x')

ax[0].set_ylabel('y') #You can set this for the other axis

plt.show()
```



Refer to the documentation for additional options.

Using Subplots For General Plots

The subplot functions above are also used in general practice to create single axis plots, due to the ability to create a reference to the axis, which grants further customization. Simply:

```
fig = plt.figure()
ax = fig.add_subplot()

ax.plot(np.linspace(0, 10))
ax.set_xlabel('x')
ax.set_ylabel('y')

plt.show()
```