Introduction to Python

COMP 6721: Applied Artificial Intelligence

Andrés Lou et al.

Concordia University

Outline

Contents of the section

What is Python?

Python is a general-purpose programming language whose emphasis is **code readability.**

What is Python?

- It is a **scripting language**, meaning there is no compiler.

What is Python?

- It is a **scripting language**, meaning there is no compiler.
- It is **dynamically typed**, meaning, amongst many other things, that variables do not require type definition when declared.

What is Python?

- It is a **scripting language**, meaning there is no compiler.
- It is **dynamically typed**, meaning, amongst many other things, that variables do not require type definition when declared.
- It supports many **programming paradigms**, including OOP, functional programming, imperative programming etc.

'Hello World'

```
#This will print the message 'Hello World'
#(these are comments btw)

print('Hello world')
```

'Hello World'

```
#This will print the message 'Hello World'
#(these are comments btw)

print('Hello world')

# This will create two lists of strings and output
# each one of their members

spanish = ['hola','mundo']
french = ['bonjour','le','monde']
for word in spanish: print(word)
for word in french: print(word)
```

'Hello World'

Words of the Wise

Enter `import this` for some wisdom.

Installation

Python is available for free via prepackaged installers for all commonly used platforms, though macOS and Linux distros already include a version of Python.

Beware the End-of-life

no 2 4 u

Python 2.0 was released nearly 20 years ago and will reach its end-of-life in 2020. Despite the large body of code written in Python 2, it is recommended that you **adopt Python 3 as soon as possible**.

Figure: *

Pictured: Guido van Rossum saying 'lolno' when asked if there was ever going to be a Python 2.8

Anaconda

One of the most useful frameworks and implementations of Python is **Anaconda**, a software package that includes the Python and R languages along with a multitude of useful libraries.

Why Python matters in AI

Python has *a ton* of useful libraries at its disposal, including the most important, widespread and commonly used libraries in modern AI and Data Science.

TensorFlow scikit learn pandas $y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$

Keras NumPy

how2python

The fact that Python is a scripting language means that you have native access to a **shell**, which is a **command-line interface (CLI)** that runs Python code. You can also run code by writing full **scripts** and then running them through the interpreter.

how2python

Running Python from the shell is useful when developing or testing.

how2python

Running Python from the shell is useful when developing or testing.

```
>> a = 2
```

how2python

Running Python from the shell is useful when developing or testing.

```
>> a = 2
>> b = [1,2,3]
```

how2python

Running Python from the shell is useful when developing or testing.

```
>> a = 2
>> b = [1,2,3]
>> a
```

how2python

Running Python from the shell is useful when developing or testing.

```
>> a = 2
>> b = [1,2,3]
>> a
2
```

how2python

Running Python from the shell is useful when developing or testing.

```
>> a = 2
>> b = [1,2,3]
>> a
2
>> b
```

how2python

Running Python from the shell is useful when developing or testing.

```
>> a = 2
>> b = [1,2,3]
>> a
2
>> b
[1,2,3]
```

how2python

Running Python from the shell is useful when developing or testing.

```
>> a = 2
>> b = [1,2,3]
>> a
2
>> b
[1,2,3]
>> c = 3
```

how2python

Running Python from the shell is useful when developing or testing.

```
>> a = 2
>> b = [1,2,3]
>> a
2
>> b
[1,2,3]
>> c = 3
>> a + c
```

how2python

Running Python from the shell is useful when developing or testing.

```
>> a = 2
>> b = [1,2,3]
>> a
2
>> b
[1,2,3]
>> c = 3
>> a + c
5
```

how2python

Scripts need to be passed through the Python interpreter for execution. The following snippet, `example.py`, is an example of a short Python script.

how2python

Scripts need to be passed through the Python interpreter for execution. The following snippet, `example.py`, is an example of a short Python script.

```python
a = 3
b = [1,2,3]
c = 4
if a in b:
    print(True)
else: print(False)
if c not in b: print(c,'is not in the list')
```

how2python

Scripts need to be passed through the Python interpreter for execution. The following snippet, `example.py`, is an example of a short Python script.

```python
a = 3
b = [1,2,3]
c = 4
if a in b:
    print(True)
else: print(False)
if c not in b: print(c,'is not in the list')
```

When run, the output is the following:

```
$ python example.py
```

how2python

Scripts need to be passed through the Python interpreter for execution. The following snippet, `example.py`, is an example of a short Python script.

```
a = 3
b = [1,2,3]
c = 4
if a in b:
    print(True)
else: print(False)
if c not in b: print(c,'is not in the list')
```

When run, the output is the following:

```
$ python example.py
True
4 is not in the list
```

how2python

Like other programming languages, Python development can be done with **IDEs**. There are *a lot* of them, and they run the gamut from text editors with Python plugins to Python-specific suites.

how2python

A very useful tool for collaborative projects and development is **Project Jupyter**, developers of the **Jupyter Notebook**, an web application designed to write documents containing live code, equations and general awesomeness.

how2python

And finally, there's Google's **Colaboratory** project, a Jupyter Notebook that runs on the cloud and on Google Drive, which gives you access to powerful computing.

Contents of the section

Data types

The Python built-in types are: **numerics**, **sequences, mappings, classes, instances** and **exceptions**.

Data types

One of the striking characteristics of Python, especially if coming from a C-like language, is that data types are only checked at run-time, meaning that variable types are implicitly declared.

Data types

One of the striking characteristics of Python, especially if coming from a C-like language, is that data types are only checked at run-time, meaning that variable types are implicitly declared.

```
a = 3
b = 3.141592
c = 'some call me tim'
d = [a,b,c]
print(d)
```

Data types

One of the striking characteristics of Python, especially if coming from a C-like language, is that data types are only checked at run-time, meaning that variable types are implicitly declared.

```
a = 3
b = 3.141592
c = 'some call me tim'
d = [a,b,c]
print(d)

[3, 3.141592, 'some call me tim']
```

Numerics

There are three built-in numeric types[1]: `int`, `float`, and `complex`.

[1] doc

Numerics

There are three built-in numeric types[1]: `int`, `float`, and `complex`.

```
n = 3
x = float(n)
a = complex(n,x)
print(n,x,a)
```

[1] doc

Numerics

There are three built-in numeric types[1]: `int`, `float`, and `complex`.

```
n = 3
x = float(n)
a = complex(n,x)
print(n,x,a)

3 3.0 (3+3j)
```

[1] doc

Sequences

There are three built-in sequence types[2]: `list`, `tuple` and `range`. The first is **mutable**, while the latter two are **immutable**.

[2] doc

Sequences

There are three built-in sequence types[2]: `list`, `tuple` and `range`. The first is **mutable**, while the latter two are **immutable**.

```
lst = ['this','is','a','list','type']
```

[2]doc

Sequences

There are three built-in sequence types[2]: `list`, `tuple` and `range`. The first is **mutable**, while the latter two are **immutable**.

```
lst = ['this','is','a','list','type']
tup = ('this','is','a','tuple','type')
```

[2] doc

Sequences

There are three built-in sequence types[2]: `list`, `tuple` and `range`. The first is **mutable**, while the latter two are **immutable**.

```
lst = ['this','is','a','list','type']
tup = ('this','is','a','tuple','type')
lst[3] = 'mutable'
# tup[3] = 'immutable' <- this would crash program
```

[2] doc

Sequences

There are three built-in sequence types[2]: `list`, `tuple` and `range`. The first is **mutable**, while the latter two are **immutable**.

```
lst = ['this','is','a','list','type']
tup = ('this','is','a','tuple','type')
lst[3] = 'mutable'
# tup[3] = 'immutable' <- this would crash program

print(lst)
```

[2] doc

Sequences

There are three built-in sequence types[2]: `list`, `tuple` and `range`. The first is **mutable**, while the latter two are **immutable**.

```
lst = ['this','is','a','list','type']
tup = ('this','is','a','tuple','type')
lst[3] = 'mutable'
# tup[3] = 'immutable' <- this would crash program

print(lst)

['this','is','a','mutable','type']
```

[2]doc

Sequences

The `range` type is used to create discrete intervals, usually used in loops.

Sequences

The range type is used to create discrete intervals, usually used in loops.

```
rng_1 = range(10)
rng_2 = range(2,10)
rng_3 = range(0,30,5)
for i in rng_1: print(i)
for i in rng_2: print(i)
for i in rng_3: print(i) # try these yourself
```

Sequences

`for` loops

In Python, `for` loops always **iterate over sequence types**, visiting each element in the sequence at a time. This stands in contrast to iterating over a given collection of types using a boolean flag and a counter (as it is the case in C-like languages).

Iterating over sequence types

```
names = ['mal','zoe','jayne','wash','kaylee']
n = len(names)
for i in range(n): print(names[i])
for name in names: print(name)
```

Iterating over sequence types

```
names = ['mal','zoe','jayne','wash','kaylee']
n = len(names)
for i in range(n): print(names[i])
for name in names: print(name)
```

Both give you:

```
mal
zoe
jayne
wash
kaylee
```

Iterating over sequences

Something extremely important to keep in mind is that, if you want to modify a given element in a mutable sequence, **you must refer to it explicitly using its index**.

Iterating over sequences

Something extremely important to keep in mind is that, if you want to modify a given element in a mutable sequence, **you must refer to it explicitly using its index**.

```
# This will have no effect on the members of the list
for name in names: name = 'blue'

# This will replace each member of the list with
# the string 'blue'
for i in range(n): names[i] = 'blue'
```

Sequences

Sequences can be accessed and manipulated using a number of operations, including **indexing**, **slicing**, **concatenation**, **repetition**, and boolean operations of **membership**.

Sequences

Sequences can be accessed and manipulated using a number of operations, including **indexing**, **slicing**, **concatenation**, **repetition**, and boolean operations of **membership**.

```
>> a = [1,2,3,4,5]
```

Sequences

Sequences can be accessed and manipulated using a number of operations, including **indexing**, **slicing**, **concatenation**, **repetition**, and boolean operations of **membership**.

```
>> a = [1,2,3,4,5]
>> a[2]
```

Sequences

Sequences can be accessed and manipulated using a number of operations, including **indexing**, **slicing**, **concatenation**, **repetition**, and boolean operations of **membership**.

```
>> a = [1,2,3,4,5]
>> a[2]
3
```

Sequences

Sequences can be accessed and manipulated using a number of operations, including **indexing**, **slicing**, **concatenation**, **repetition**, and boolean operations of **membership**.

```
>> a = [1,2,3,4,5]
>> a[2]
3
>> a[1:3]
```

Sequences

Sequences can be accessed and manipulated using a number of operations, including **indexing**, **slicing**, **concatenation**, **repetition**, and boolean operations of **membership**.

```
>> a = [1,2,3,4,5]
>> a[2]
3
>> a[1:3]
[2, 3]
```

Sequences

Sequences can be accessed and manipulated using a number of operations, including **indexing**, **slicing**, **concatenation**, **repetition**, and boolean operations of **membership**.

```
>> a = [1,2,3,4,5]
>> a[2]
3
>> a[1:3]
[2, 3]
>> a+[6,7]
```

Sequences

Sequences can be accessed and manipulated using a number of operations, including **indexing**, **slicing**, **concatenation**, **repetition**, and boolean operations of **membership**.

```
>> a = [1,2,3,4,5]
>> a[2]
3
>> a[1:3]
[2, 3]
>> a+[6,7]
[1, 2, 3, 4, 5, 6, 7]
```

Sequences

Sequences can be accessed and manipulated using a number of operations, including **indexing**, **slicing**, **concatenation**, **repetition**, and boolean operations of **membership**.

```
>> a = [1,2,3,4,5]
>> a[2]
3
>> a[1:3]
[2, 3]
>> a+[6,7]
[1, 2, 3, 4, 5, 6, 7]
>> a*2
```

Sequences

Sequences can be accessed and manipulated using a number of operations, including **indexing**, **slicing**, **concatenation**, **repetition**, and boolean operations of **membership**.

```
>> a = [1,2,3,4,5]
>> a[2]
3
>> a[1:3]
[2, 3]
>> a+[6,7]
[1, 2, 3, 4, 5, 6, 7]
>> a*2
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

Sequences

Sequences can be accessed and manipulated using a number of operations, including **indexing**, **slicing**, **concatenation**, **repetition**, and boolean operations of **membership**.

```
>> a = [1,2,3,4,5]
>> a[2]
3
>> a[1:3]
[2, 3]
>> a+[6,7]
[1, 2, 3, 4, 5, 6, 7]
>> a*2
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
>> 3 in a
```

Sequences

Sequences can be accessed and manipulated using a number of operations, including **indexing**, **slicing**, **concatenation**, **repetition**, and boolean operations of **membership**.

```
>> a = [1,2,3,4,5]
>> a[2]
3
>> a[1:3]
[2, 3]
>> a+[6,7]
[1, 2, 3, 4, 5, 6, 7]
>> a*2
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
>> 3 in a
True
```

Sequences

Sequences can be accessed and manipulated using a number of operations, including **indexing**, **slicing**, **concatenation**, **repetition**, and boolean operations of **membership**.

```
>> a = [1,2,3,4,5]
>> a[2]
3
>> a[1:3]
[2, 3]
>> a+[6,7]
[1, 2, 3, 4, 5, 6, 7]
>> a*2
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
>> 3 in a
True
>> 8 not in a
```

Sequences

Sequences can be accessed and manipulated using a number of operations, including **indexing**, **slicing**, **concatenation**, **repetition**, and boolean operations of **membership**.

```
>> a = [1,2,3,4,5]
>> a[2]
3
>> a[1:3]
[2, 3]
>> a+[6,7]
[1, 2, 3, 4, 5, 6, 7]
>> a*2
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
>> 3 in a
True
>> 8 not in a
True
```

Sequences

btw Python `string` types are also sequences. These can be written in a variety of ways:

Sequences

btw Python `string` types are also sequences. These can be written in a variety of ways:

```
'this is a string'
```

Sequences

btw Python `string` types are also sequences. These can be written in a variety of ways:

```
'this is a string'
"this is also a string"
```

Sequences

btw Python `string` types are also sequences. These can be written in a variety of ways:

```
'this is a string'
"this is also a string"
"this one 'contains' single quotation marks"
```

Sequences

btw Python `string` types are also sequences. These can be written in a variety of ways:

```
'this is a string'
"this is also a string"
"this one 'contains' single quotation marks"
'this is the "converse"'
```

Sequences

btw Python `string` types are also sequences. These can be written in a variety of ways:

```
'this is a string'
"this is also a string"
"this one 'contains' single quotation marks"
'this is the "converse"'
'''lots of ways of representing strings'''
```

Sequences

btw Python `string` types are also sequences. These can be written in a variety of ways:

```
'this is a string'
"this is also a string"
"this one 'contains' single quotation marks"
'this is the "converse"'
'''lots of ways of representing strings'''
"""lots of them"""
```

Sequences

Strings are sequences

All of the sequence operations from the previous slides also work with strings. Thus, the usual **string operations**[3] you're used to from other C-like languages are also available in Python.

[3]Full list here

Strings

```
st = ' ALL YOUR BASE ARE BELONG TO US '
```

Strings

```
st = ' ALL YOUR BASE ARE BELONG TO US '
print(st.casefold())
print(st[10:14])
print(st.endswith('TO US '))
print(st.strip())
```

Strings

```
st = ' ALL YOUR BASE ARE BELONG TO US '
print(st.casefold())
print(st[10:14])
print(st.endswith('TO US '))
print(st.strip())

 all your base are belong to us
BASE
True
ALL YOUR BASE ARE BELONG TO US
```

Strings

A particularly useful string method four our purposes is the `split` method, which splits a string by a given separator:

Strings

A particularly useful string method four our purposes is the `split` method, which splits a string by a given separator:

```
st_space = 'This is separated by spaces'
st_comma = 'This,is,separated,by,commas'
print(st_space.split(" "))
print(st_comma.split(","))
```

Strings

A particularly useful string method four our purposes is the `split` method, which splits a string by a given separator:

```
st_space = 'This is separated by spaces'
st_comma = 'This,is,separated,by,commas'
print(st_space.split(" "))
print(st_comma.split(","))

['This', 'is', 'separated', 'by', 'spaces']
['This', 'is', 'separated', 'by', 'commas']
```

Strings

A particularly useful string method four our purposes is the `split` method, which splits a string by a given separator:

```
st_space = 'This is separated by spaces'
st_comma = 'This,is,separated,by,commas'
print(st_space.split(" "))
print(st_comma.split(","))

['This', 'is', 'separated', 'by', 'spaces']
['This', 'is', 'separated', 'by', 'commas']
```

We'll come back to this when we deal with parsing contents read from a file.

Mappings

The most widely used mapping structure in Python is the `dict` type[4], which works very similarly to a traditional **hash map**.

[4] doc

Mappings

The most widely used mapping structure in Python is the `dict` type[4], which works very similarly to a traditional **hash map**.

```
a = dict(quebec='french',ontario='english')
b = {'guatemala':'spanish','brazil':'portuguese',
         'usa':'english'}
print(a['quebec'])
print(a['ontario'])
print(b)
```

[4] doc

Mappings

The most widely used mapping structure in Python is the `dict` type[4], which works very similarly to a traditional **hash map**.

```
a = dict(quebec='french',ontario='english')
b = {'guatemala':'spanish','brazil':'portuguese',
         'usa':'english'}
print(a['quebec'])
print(a['ontario'])
print(b)
```

```
french
english
{'guatemala':'spanish','brazil':'portuguese','usa':'english'}
```

[4] doc

Mappings

`dict` types are mutable: it is possible to add and remove key-value pairs.

Mappings

dict types are mutable: it is possible to add and remove key-value pairs.

```
a = dict(kaylee='mechanic',wash='pilot',zoe='second')
a['simon'] = 'medic'
del a['wash']
print(a)
```

Mappings

dict types are mutable: it is possible to add and remove key-value pairs.

```
a = dict(kaylee='mechanic',wash='pilot',zoe='second')
a['simon'] = 'medic'
del a['wash']
print(a)

{'kaylee': 'mechanic', 'zoe': 'second', 'simon': 'medic'}
```

Useful data structures: Set types

The Python Set types, `set` and `frozenset`, are data structure implementations of the mathematical concept of sets[5]. The difference between each type is that the former is **mutable**, while the latter is **immutable**. Elements belonging to set types are **unordered** and **do not allow for repetitions**.

[5] doc

Useful data structures: Set types

```
>> a = {'a','b','c','d','d','d'}
```

Useful data structures: Set types

```
>> a = {'a','b','c','d','d','d'}
>> a
```

Useful data structures: Set types

```
>> a = {'a','b','c','d','d','d'}
>> a
{'d', 'a', 'c', 'b'}
```

Useful data structures: Set types

```
>> a = {'a','b','c','d','d','d'}
>> a
{'d', 'a', 'c', 'b'}
>> len(a)
```

Useful data structures: Set types

```
>> a = {'a','b','c','d','d','d'}
>> a
{'d', 'a', 'c', 'b'}
>> len(a)
4
```

Useful data structures: Set types

```
>> a = {'a','b','c','d','d','d'}
>> a
{'d', 'a', 'c', 'b'}
>> len(a)
4
>> 'f' in a
```

Useful data structures: Set types

```
>> a = {'a','b','c','d','d','d'}
>> a
{'d', 'a', 'c', 'b'}
>> len(a)
4
>> 'f' in a
False
```

Useful data structures: Set types

```
>> a = {'a','b','c','d','d','d'}
>> a
{'d', 'a', 'c', 'b'}
>> len(a)
4
>> 'f' in a
False
>> a.issuperset({'a','b'})
```

Useful data structures: Set types

```
>> a = {'a','b','c','d','d','d'}
>> a
{'d', 'a', 'c', 'b'}
>> len(a)
4
>> 'f' in a
False
>> a.issuperset({'a','b'})
True
```

Useful data structures: Set types

```
>> a = {'a','b','c','d','d','d'}
>> a
{'d', 'a', 'c', 'b'}
>> len(a)
4
>> 'f' in a
False
>> a.issuperset({'a','b'})
True
>> a|{'e','f','g'} # union
```

Useful data structures: Set types

```
>> a = {'a','b','c','d','d','d'}
>> a
{'d', 'a', 'c', 'b'}
>> len(a)
4
>> 'f' in a
False
>> a.issuperset({'a','b'})
True
>> a|{'e','f','g'} # union
{'b', 'g', 'd', 'c', 'f', 'a', 'e'}
```

Useful data structures: Set types

```
>> a = {'a','b','c','d','d','d'}
>> a
{'d', 'a', 'c', 'b'}
>> len(a)
4
>> 'f' in a
False
>> a.issuperset({'a','b'})
True
>> a|{'e','f','g'} # union
{'b', 'g', 'd', 'c', 'f', 'a', 'e'}
>> a & {'c','d','e','f','g'} # intersection
```

Useful data structures: Set types

```
>> a = {'a','b','c','d','d','d'}
>> a
{'d', 'a', 'c', 'b'}
>> len(a)
4
>> 'f' in a
False
>> a.issuperset({'a','b'})
True
>> a|{'e','f','g'} # union
{'b', 'g', 'd', 'c', 'f', 'a', 'e'}
>> a & {'c','d','e','f','g'} # intersection
{'c','d'}
```

Don't fear the README

srsly

It is **good coding practice** to be able to **read the documentation** of any given piece of software you work with. Google and Stack Overflow are your friends but they should not be your only resource.

Contents of the section

Functions

Python supports the use of **functions**, both as free functions and members of a class.

Functions

Python supports the use of **functions**, both as free functions and members of a class.

```python
# the following is a function that takes
# a list of numbers and returns the total sum
def sum(numbers):
        n = len(numbers)          # len returns the number
        sum = 0                   # of elements in a sequence
        for i in range(n):
                sum += numbers[i]
        return sum

n = [1,2,3]
print('The sum of',n,'is',sum(n))
```

Functions

Python supports the use of **functions**, both as free functions and members of a class.

```
# the following is a function that takes
# a list of numbers and returns the total sum
def sum(numbers):
        n = len(numbers)          # len returns the number
        sum = 0                   # of elements in a sequence
        for i in range(n):
                sum += numbers[i]
        return sum

n = [1,2,3]
print('The sum of',n,'is',sum(n))

The sum of [1, 2, 3] is 6
```

Diff'rent scopes

Notice how Python denotes different expressions and different scopes via **line breaks**, **white space** and **indentation**.

Diff'rent scopes

```
numbers = [1,2,3,4,5,6,7,8,9]
for number in numbers:
        if number % 2 == 0:
                print('The number',number,'is even')
                print('Its index is',numbers.index(number))
        else:
                print('The number',number,'is odd')
```

Diff'rent scopes

```
numbers = [1,2,3,4,5,6,7,8,9]
for number in numbers:
        if number % 2 == 0:
                print('The number',number,'is even')
                print('Its index is',numbers.index(number))
        else:
                print('The number',number,'is odd')
```

Diff'rent scopes

```
numbers = [1,2,3,4,5,6,7,8,9]
for number in numbers:
        if number % 2 = 0:
                print('The number',number,'is even')
                print('Its index is',numbers.index(number))
        else:
                print('The number',number,'is odd')
```

Contents of the section

Object-Oriented Programming

The OOP paradigm is implemented in Python in a similar fashion to the C-like languages[6]. Member functions, both **constructors** and **methods**, are supported, as are the functionalities of **inheritance** and **encapsulation**, though the latter behaves in a peculiar manner.

[6]The Python tutorial page

Object Oriented Programming

Compared to C-like languages, perhaps the most salient feature of Python OOP is the `self` keyword.

An example of a custom class

```
# A rough class describing a RPG character
class PlayerCharacter:
        # This is how you implement a class constructor
        def __init__(self,job,strength,intelligence,level):
                self.__job = job
                self.__strength = strength
                self.__intelligence = intelligence
                self.__level = level

        def isSmart(self):
                return self.__intelligence > 12

        def levelUp(self,level_increase):
                self.__level += level_increase
```

An example of a custom class

```
# An instantiation of an object of the class PlayerCharacter
import PlayerCharacter

pc1 = PlayerCharacter('Wizard',10,12,1)
pc2 = PlayerCharacter('Fighter',13,9,1)

if pc1.isSmart(): print('This character can cast spells')

pc2.levelUp(1)
print('The',pc2.__job,'is levelling up!')
```

Loading modules

btw that's how you import modules and libraries, using the `import` statement. Given a module, you can either import all of its concents or select which classes/functions you wish to use.

Loading modules

btw that's how you import modules and libraries, using the `import` statement. Given a module, you can either import all of its concents or select which classes/functions you wish to use.

```
# there are two ways of importing the factorial function
# from the math library
```

Loading modules

btw that's how you import modules and libraries, using the `import` statement. Given a module, you can either import all of its concents or select which classes/functions you wish to use.

```
# there are two ways of importing the factorial function
# from the math library
>> import math
```

Loading modules

btw that's how you import modules and libraries, using the `import` statement. Given a module, you can either import all of its concents or select which classes/functions you wish to use.

```
# there are two ways of importing the factorial function
# from the math library
>> import math
>> math.factorial(3)
```

Loading modules

btw that's how you import modules and libraries, using the `import` statement. Given a module, you can either import all of its concents or select which classes/functions you wish to use.

```
# there are two ways of importing the factorial function
# from the math library
>> import math
>> math.factorial(3)
6
```

Loading modules

btw that's how you import modules and libraries, using the `import` statement. Given a module, you can either import all of its concents or select which classes/functions you wish to use.

```
# there are two ways of importing the factorial function
# from the math library
>> import math
>> math.factorial(3)
6
>> from math import factorial
```

Loading modules

btw that's how you import modules and libraries, using the `import` statement. Given a module, you can either import all of its concents or select which classes/functions you wish to use.

```
# there are two ways of importing the factorial function
# from the math library
>> import math
>> math.factorial(3)
6
>> from math import factorial
>> factorial(3)
```

Loading modules

btw that's how you import modules and libraries, using the `import` statement. Given a module, you can either import all of its concents or select which classes/functions you wish to use.

```
# there are two ways of importing the factorial function
# from the math library
>> import math
>> math.factorial(3)
6
>> from math import factorial
>> factorial(3)
6
```

No privacy

It is noticeable that Python, unlike other C-like languages, **does not have access level modifiers**, meaning the concept of private or public members does not exist. Python trusts you will behave like an adult. No, really.

No privacy

Follow the Way of the Python

Coding in Python is all about making it easy for your code to be readable at glance, that is, making your programming style "**pythonic**".
Obfuscating permissions and strict type-checking and commonly thought of as being against the philosophy of Python.

Contents of the section

Files

Python can handle the **writing** and **reading** of both text and binary files[7]. File handles are created with the `open` function, which takes two arguments: the **file name** and the **mode**.

[7]Tutorial page

Reading a file

Given a file, `data.csv` (located in the same working directory), we can read the file like this:

Reading a file

Given a file, `data.csv` (located in the same working directory), we can read the file like this:

```
file_handle = open('data.csv','r')
file_content = file_handle.read()
file_handle.close()
```

Reading a file

Given a file, `data.csv` (located in the same working directory), we can read the file like this:

```
file_handle = open('data.csv','r')
file_content = file_handle.read()
file_handle.close()
```

The mode argument is optional: if omitted, the interpreter will default to using `'r'`.

Reading a file

Different ways to read

There's a little more to it when it comes to reading the contents of a file. We can read a file using the methods `read`, which reads the file contents **as one big string**, `readline`, which reads **a single line from the file**, and `readlines`, which returns **a list of all the lines in the file**.

Writing to a file

If we wish to create a new file and write content to it, we use the `'w'` mode:

Writing to a file

If we wish to create a new file and write content to it, we use the 'w' mode:

```
names = ['mal','zoe','jayne','wash','kaylee',
         'inara','book','simon','river']
file_handle = open('names.txt','w')
for name in names: file_handle.write(name+'\n')
file_handle.close()
```

Writing to a file

If we wish to create a new file and write content to it, we use the `'w'` mode:

```
names = ['mal','zoe','jayne','wash','kaylee',
         'inara','book','simon','river']
file_handle = open('names.txt','w')
for name in names: file_handle.write(name+'\n')
file_handle.close()
```

The `'w'` mode will create a new file and will also **overwrite** any preexisting file with the same name.

Appending to a file

If we wish to add content to a file without overwriting it, we use the `'a'` mode:

Appending to a file

If we wish to add content to a file without overwriting it, we use the 'a' mode:

```
file_handle = open('names.txt','a')
file_handle.write('serenity')
file_handle.close()
```

Parsing file contents

Using these methods, we can write simple parsers to extract structured data from files. Suppose we have a csv file, `serenity.csv`, with the following contents:

Parsing file contents

Using these methods, we can write simple parsers to extract structured data from files. Suppose we have a csv file, `serenity.csv`, with the following contents:

```
Malcolm Reynolds,Captain,31
Zoe Alleyne,Second,30
Jayne Cobb,Crew,42
Hoban Washburne,Pilot,31
Kaywinnet Frye,Mechanic,21
```

Parsing file contents

Using these methods, we can write simple parsers to extract structured data from files. Suppose we have a csv file, `serenity.csv`, with the following contents:

```
Malcolm Reynolds,Captain,31
Zoe Alleyne,Second,30
Jayne Cobb,Crew,42
Hoban Washburne,Pilot,31
Kaywinnet Frye,Mechanic,21
```

Each line represents and entry, and each entry is delimited by commas, which denote respectively the fields of *name*, *role* and *age*.

Parsing file contents

We will write a parser that returns a `dict` structure that pairs a *role* to a `tuple` containing the *name* and the *age*.

Parsing file contents

We will write a parser that returns a `dict` structure that pairs a *role* to a `tuple` containing the *name* and the *age*.

```
in_file = open('serenity.csv')
```

Parsing file contents

We will write a parser that returns a `dict` structure that pairs a *role* to a `tuple` containing the *name* and the *age*.

```
in_file = open('serenity.csv')
entries = in_file.readlines()
```

Parsing file contents

We will write a parser that returns a `dict` structure that pairs a *role* to a `tuple` containing the *name* and the *age*.

```
in_file = open('serenity.csv')
entries = in_file.readlines()
serenity_crew = dict()
```

Parsing file contents

We will write a parser that returns a `dict` structure that pairs a *role* to a `tuple` containing the *name* and the *age*.

```
in_file = open('serenity.csv')
entries = in_file.readlines()
serenity_crew = dict()
for line in entries:
    entry = line.split(",")
```

Parsing file contents

We will write a parser that returns a `dict` structure that pairs a *role* to a `tuple` containing the *name* and the *age*.

```python
in_file = open('serenity.csv')
entries = in_file.readlines()
serenity_crew = dict()
for line in entries:
    entry = line.split(",")
    serenity_crew[entry[1]] = (entry[0],entry[2])
```

Parsing file contents

We will write a parser that returns a `dict` structure that pairs a *role* to a `tuple` containing the *name* and the *age*.

```
in_file = open('serenity.csv')
entries = in_file.readlines()
serenity_crew = dict()
for line in entries:
    entry = line.split(",")
    serenity_crew[entry[1]] = (entry[0],entry[2])

print(serenity_crew['Captain'])
print(serenity_crew['Second'])
print(serenity_crew['Pilot'])
print(serenity_crew['Mechanic'])
```

Binary files

by(tes) the way

If you are dealing with files that contain no text, you should use the `'b'` mode to indicate binary mode. Binary files ignore usual text formatting conventions such as end line characters, text encoding, etc.

The `with` statement

The `with` statement, amongst its more general functionalities, can be used to open, read/write files and close them within a single code block.

The `with` statement

The `with` statement, amongst its more general functionalities, can be used to open, read/write files and close them within a single code block.

```python
serenity_crew = dict()
with open('data.csv') as file_handle:
    entries = file_handle.readlines()
    for line in entries:
        entry = line.split(",")
        serenity_crew[entry[1]] = (entry[0],entry[2])
```

Contents of the section

Just what is NumPy?

NumPy is a library that adds the functionality of **large, multidimensional arrays** along with a whole bunch of useful **functions pertaining the handling of matrices**. As you might expect, it is especially useful in Machine Learning.

Just what is NumPy?

Most of the contents of the following section are taken from the official tutorial pages, which can be found here.

NumPy basics

The basic data structure of NumPy is the multidimensional array, which is implemented as a **table of elements of the same type**, indexed by an **n-dimensional tuple of positive integers**.

NumPy basics

The basic data structure of NumPy is the multidimensional array, which is implemented as a **table of elements of the same type**, indexed by an **n-dimensional tuple of positive integers**.

```
import numpy as np
a = np.array([(1,2,3),(4,5,6),(7,8,9)])
print(a)
```

NumPy basics

The basic data structure of NumPy is the multidimensional array, which is implemented as a **table of elements of the same type**, indexed by an **n-dimensional tuple of positive integers**.

```
import numpy as np
a = np.array([(1,2,3),(4,5,6),(7,8,9)])
print(a)

[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]
```

NumPy basics

Dimensions in a NumPy array are called **axes**, and the number of elements in a given axis is the **length** of the axis. For example, the array `[[1,2,3],[4,5,6]]` has two axes, the first of which is of length 2 and the second of length 3.

NumPy basics

A NumPy array has several properties in accordance to its axes and the distribution of its elements across them.

NumPy basics

A NumPy array has several properties in accordance to its axes and the distribution of its elements across them.

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

NumPy basics

A NumPy array has several properties in accordance to its axes and the distribution of its elements across them.

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

NumPy basics

A NumPy array has several properties in accordance to its axes and the distribution of its elements across them.

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

NumPy basics

A NumPy array has several properties in accordance to its axes and the distribution of its elements across them.

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

NumPy basics

A NumPy array has several properties in accordance to its axes and the distribution of its elements across them.

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

NumPy basics

A NumPy array has several properties in accordance to its axes and the distribution of its elements across them.

```
>> a = np.array([[1,2,3],[4,5,6],[7,8,9]])
>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>> a.ndim
2
>> a.shape
```

NumPy basics

A NumPy array has several properties in accordance to its axes and the distribution of its elements across them.

```
>> a = np.array([[1,2,3],[4,5,6],[7,8,9]])
>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>> a.ndim
2
>> a.shape
(3,3)
```

NumPy basics

A NumPy array has several properties in accordance to its axes and the distribution of its elements across them.

```
>> a = np.array([[1,2,3],[4,5,6],[7,8,9]])
>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>> a.ndim
2
>> a.shape
(3,3)
>> a.size
```

NumPy basics

A NumPy array has several properties in accordance to its axes and the distribution of its elements across them.

```
>> a = np.array([[1,2,3],[4,5,6],[7,8,9]])
>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>> a.ndim
2
>> a.shape
(3,3)
>> a.size
9
```

Creating an array

There is more than one way of creating arrays:

Creating an array

There is more than one way of creating arrays:

```
>> # create a list and convert it to an array
```

Creating an array

There is more than one way of creating arrays:

```
>> # create a list and convert it to an array
>> a = [1,2,3,4,5,6]
```

Creating an array

There is more than one way of creating arrays:

```
>> # create a list and convert it to an array
>> a = [1,2,3,4,5,6]
>> a = np.array(a)
```

Creating an array

There is more than one way of creating arrays:

```
>> # create a list and convert it to an array
>> a = [1,2,3,4,5,6]
>> a = np.array(a)
>> a
```

Creating an array

There is more than one way of creating arrays:

```
>> # create a list and convert it to an array
>> a = [1,2,3,4,5,6]
>> a = np.array(a)
>> a
array([1, 2, 3, 4, 5, 6])
```

Creating an array

There is more than one way of creating arrays:

```
>> # create a list and convert it to an array
>> a = [1,2,3,4,5,6]
>> a = np.array(a)
>> a
array([1, 2, 3, 4, 5, 6])
>> # reshape a preexisting array
```

Creating an array

There is more than one way of creating arrays:

```
>> # create a list and convert it to an array
>> a = [1,2,3,4,5,6]
>> a = np.array(a)
>> a
array([1, 2, 3, 4, 5, 6])
>> # reshape a preexisting array
>> a = a.reshape(2,3)
```

Creating an array

There is more than one way of creating arrays:

```
>> # create a list and convert it to an array
>> a = [1,2,3,4,5,6]
>> a = np.array(a)
>> a
array([1, 2, 3, 4, 5, 6])
>> # reshape a preexisting array
>> a = a.reshape(2,3)
>> a
```

Creating an array

There is more than one way of creating arrays:

```
>> # create a list and convert it to an array
>> a = [1,2,3,4,5,6]
>> a = np.array(a)
>> a
array([1, 2, 3, 4, 5, 6])
>> # reshape a preexisting array
>> a = a.reshape(2,3)
>> a
array([[1, 2, 3],
       [4, 5, 6]])
```

Creating an array

There is more than one way of creating arrays:

```
>> # create a list and convert it to an array
>> a = [1,2,3,4,5,6]
>> a = np.array(a)
>> a
array([1, 2, 3, 4, 5, 6])
>> # reshape a preexisting array
>> a = a.reshape(2,3)
>> a
array([[1, 2, 3],
       [4, 5, 6]])
>> # create one by specifying its shape and values
```

Creating an array

There is more than one way of creating arrays:

```
>> # create a list and convert it to an array
>> a = [1,2,3,4,5,6]
>> a = np.array(a)
>> a
array([1, 2, 3, 4, 5, 6])
>> # reshape a preexisting array
>> a = a.reshape(2,3)
>> a
array([[1, 2, 3],
       [4, 5, 6]])
>> # create one by specifying its shape and values
>> a = np.array([[1.0,2.0,3.0],[4.0,5.0,6.0]])
```

Creating an array

There is more than one way of creating arrays:

```
>> # create a list and convert it to an array
>> a = [1,2,3,4,5,6]
>> a = np.array(a)
>> a
array([1, 2, 3, 4, 5, 6])
>> # reshape a preexisting array
>> a = a.reshape(2,3)
>> a
array([[1, 2, 3],
       [4, 5, 6]])
>> # create one by specifying its shape and values
>> a = np.array([[1.0,2.0,3.0],[4.0,5.0,6.0]])
>> a
```

Creating an array

There is more than one way of creating arrays:

```
>> # create a list and convert it to an array
>> a = [1,2,3,4,5,6]
>> a = np.array(a)
>> a
array([1, 2, 3, 4, 5, 6])
>> # reshape a preexisting array
>> a = a.reshape(2,3)
>> a
array([[1, 2, 3],
       [4, 5, 6]])
>> # create one by specifying its shape and values
>> a = np.array([[1.0,2.0,3.0],[4.0,5.0,6.0]])
>> a
>> array([[1., 2., 3.],
          [4., 5., 6.]])
```

Creating an array

It is common to know beforehand just how big an array is without actually knowing each of the values. To this end, it is possible to create an array filled with placeholders to be filled afterwards.

Creating an array

It is common to know beforehand just how big an array is without actually knowing each of the values. To this end, it is possible to create an array filled with placeholders to be filled afterwards.

```
>> np.zeros((3,4))
```

Creating an array

It is common to know beforehand just how big an array is without actually knowing each of the values. To this end, it is possible to create an array filled with placeholders to be filled afterwards.

```
>> np.zeros((3,4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

Creating an array

It is common to know beforehand just how big an array is without actually knowing each of the values. To this end, it is possible to create an array filled with placeholders to be filled afterwards.

```
>> np.zeros((3,4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>> np.ones((3,4))
```

Creating an array

It is common to know beforehand just how big an array is without actually knowing each of the values. To this end, it is possible to create an array filled with placeholders to be filled afterwards.

```
>> np.zeros((3,4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>> np.ones((3,4))
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

Creating an array

It is common to know beforehand just how big an array is without actually knowing each of the values. To this end, it is possible to create an array filled with placeholders to be filled afterwards.

```
>> np.zeros((3,4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>> np.ones((3,4))
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
>> np.empty((2,3))
```

Creating an array

It is common to know beforehand just how big an array is without actually knowing each of the values. To this end, it is possible to create an array filled with placeholders to be filled afterwards.

```
>> np.zeros((3,4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>> np.ones((3,4))
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
>> np.empty((2,3))
array([[3.73603959e-262, 6.02658058e-154, 6.55490914e-260],
       [5.30498948e-313, 3.14673309e-307, 1.00000000e+000]])
```

Array operations

Arithmetic operations are applied element-wise:

Array operations

Arithmetic operations are applied element-wise:

```
>> a = np.array([20,30,40,50])
```

Array operations

Arithmetic operations are applied element-wise:

```
>> a = np.array([20,30,40,50])
>> b = np.arange(4)
```

Array operations

Arithmetic operations are applied element-wise:

```
>> a = np.array([20,30,40,50])
>> b = np.arange(4)
>> b
```

Array operations

Arithmetic operations are applied element-wise:

```
>> a = np.array([20,30,40,50])
>> b = np.arange(4)
>> b
array([0, 1, 2, 3])
```

Array operations

Arithmetic operations are applied element-wise:

```
>> a = np.array([20,30,40,50])
>> b = np.arange(4)
>> b
array([0, 1, 2, 3])
>> a-b
```

Array operations

Arithmetic operations are applied element-wise:

```
>> a = np.array([20,30,40,50])
>> b = np.arange(4)
>> b
array([0, 1, 2, 3])
>> a-b
array([20, 29, 38, 47])
```

Array operations

Arithmetic operations are applied element-wise:

```
>> a = np.array([20,30,40,50])
>> b = np.arange(4)
>> b
array([0, 1, 2, 3])
>> a-b
array([20, 29, 38, 47])
>> b**2
```

Array operations

Arithmetic operations are applied element-wise:

```
>> a = np.array([20,30,40,50])
>> b = np.arange(4)
>> b
array([0, 1, 2, 3])
>> a-b
array([20, 29, 38, 47])
>> b**2
array([0, 1, 4, 9])
```

Array operations

Arithmetic operations are applied element-wise:

```
>> a = np.array([20,30,40,50])
>> b = np.arange(4)
>> b
array([0, 1, 2, 3])
>> a-b
array([20, 29, 38, 47])
>> b**2
array([0, 1, 4, 9])
>> # the * operator is also applied element-wise
```

Array operations

Arithmetic operations are applied element-wise:

```
>> a = np.array([20,30,40,50])
>> b = np.arange(4)
>> b
array([0, 1, 2, 3])
>> a-b
array([20, 29, 38, 47])
>> b**2
array([0, 1, 4, 9])
>> # the * operator is also applied element-wise
>> a*b
```

Array operations

Arithmetic operations are applied element-wise:

```
>> a = np.array([20,30,40,50])
>> b = np.arange(4)
>> b
array([0, 1, 2, 3])
>> a-b
array([20, 29, 38, 47])
>> b**2
array([0, 1, 4, 9])
>> # the * operator is also applied element-wise
>> a*b
array([  0,  30,  80, 150])
```

Array operations

Matrix multiplication is carried out with the `@` operator, or with the `dot` method.

Array operations

Matrix multiplication is carried out with the `@` operator, or with the `dot` method.

```
>> a = np.array([1,2,3])
```

Array operations

Matrix multiplication is carried out with the `@` operator, or with the `dot` method.

```
>> a = np.array([1,2,3])
>> B = np.arange(9).reshape(3,3)
```

Array operations

Matrix multiplication is carried out with the `@` operator, or with the `dot` method.

```
>> a = np.array([1,2,3])
>> B = np.arange(9).reshape(3,3)
>> a@B
```

Array operations

Matrix multiplication is carried out with the `@` operator, or with the `dot` method.

```
>> a = np.array([1,2,3])
>> B = np.arange(9).reshape(3,3)
>> a@B
array([24, 30, 36])
```

Array operations

Matrix multiplication is carried out with the `@` operator, or with the `dot` method.

```
>> a = np.array([1,2,3])
>> B = np.arange(9).reshape(3,3)
>> a@B
array([24, 30, 36])
>> B.dot(a)
```

Array operations

Matrix multiplication is carried out with the `@` operator, or with the `dot` method.

```
>> a = np.array([1,2,3])
>> B = np.arange(9).reshape(3,3)
>> a@B
array([24, 30, 36])
>> B.dot(a)
array([ 8, 26, 44])
```

Indexing, slicing and iterating

As with other sequences, arrays can be indexed, sliced and iterated over to access their elements.

Indexing, slicing and iterating

As with other sequences, arrays can be indexed, sliced and iterated over to access their elements.

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

Indexing, slicing and iterating

As with other sequences, arrays can be indexed, sliced and iterated over to access their elements.

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

Indexing, slicing and iterating

As with other sequences, arrays can be indexed, sliced and iterated over to access their elements.

```
>> a = np.array(([.1,.2,.3],[.4,.5,.6],[.7,.8,.9]))
>> a[1,2]
0.6
```

Indexing, slicing and iterating

As with other sequences, arrays can be indexed, sliced and iterated over to access their elements.

```
>> a = np.array(([.1,.2,.3],[.4,.5,.6],[.7,.8,.9]))
>> a[1,2]
0.6
>> a[1]
```

Indexing, slicing and iterating

As with other sequences, arrays can be indexed, sliced and iterated over to access their elements.

```
>> a = np.array(([.1,.2,.3],[.4,.5,.6],[.7,.8,.9]))
>> a[1,2]
0.6
>> a[1]
array([0.4, 0.5, 0.6])
```

Indexing, slicing and iterating

As with other sequences, arrays can be indexed, sliced and iterated over to access their elements.

```
>> a = np.array(([.1,.2,.3],[.4,.5,.6],[.7,.8,.9]))
>> a[1,2]
0.6
>> a[1]
array([0.4, 0.5, 0.6])
>> a[0:2,0]
```

Indexing, slicing and iterating

As with other sequences, arrays can be indexed, sliced and iterated over to access their elements.

```
>> a = np.array(([.1,.2,.3],[.4,.5,.6],[.7,.8,.9]))
>> a[1,2]
0.6
>> a[1]
array([0.4, 0.5, 0.6])
>> a[0:2,0]
array([0.1, 0.4])
```

Indexing, slicing and iterating

When iterating, the first axis is taken as the the iterating item by default:

Indexing, slicing and iterating

When iterating, the first axis is taken as the the iterating item by default:

```
>> for row in a: row
```

Indexing, slicing and iterating

When iterating, the first axis is taken as the the iterating item by default:

```
>> for row in a: row
array([0.1, 0.2, 0.3])
array([0.4, 0.5, 0.6])
array([0.7, 0.8, 0.9])
```

Indexing, slicing and iterating

When iterating, the first axis is taken as the the iterating item by default:

```
>> for row in a: row
array([0.1, 0.2, 0.3])
array([0.4, 0.5, 0.6])
array([0.7, 0.8, 0.9])
```

If you wanted to iterate over all elements of the array, you need to use the `flat` attribute:

Indexing, slicing and iterating

When iterating, the first axis is taken as the the iterating item by default:

```
>> for row in a: row
array([0.1, 0.2, 0.3])
array([0.4, 0.5, 0.6])
array([0.7, 0.8, 0.9])
```

If you wanted to iterate over all elements of the array, you need to use the `flat` attribute:

```
>> for element in a.flat: element + 1
```

Indexing, slicing and iterating

When iterating, the first axis is taken as the the iterating item by default:

```
>> for row in a: row
array([0.1, 0.2, 0.3])
array([0.4, 0.5, 0.6])
array([0.7, 0.8, 0.9])
```

If you wanted to iterate over all elements of the array, you need to use the `flat` attribute:

```
>> for element in a.flat: element + 1
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9
```

Shape manipulation

Arrays can be reshaped without affecting their contents:

Shape manipulation

Arrays can be reshaped without affecting their contents:

```
>> a = np.floor(10*np.random.random((3,4)))
```

Shape manipulation

Arrays can be reshaped without affecting their contents:

```
>> a = np.floor(10*np.random.random((3,4)))
>> a
```

Shape manipulation

Arrays can be reshaped without affecting their contents:

```
>> a = np.floor(10*np.random.random((3,4)))
>> a
array([[2., 2., 6., 8.],
       [1., 6., 1., 6.],
       [5., 1., 5., 0.]])
```

Shape manipulation

Arrays can be reshaped without affecting their contents:

```
>> a = np.floor(10*np.random.random((3,4)))
>> a
array([[2., 2., 6., 8.],
       [1., 6., 1., 6.],
       [5., 1., 5., 0.]])
>> a.ravel()
```

Shape manipulation

Arrays can be reshaped without affecting their contents:

```
>> a = np.floor(10*np.random.random((3,4)))
>> a
array([[2., 2., 6., 8.],
       [1., 6., 1., 6.],
       [5., 1., 5., 0.]])
>> a.ravel()
array([2., 2., 6., 8., 1., 6., 1., 6., 5., 1., 5., 0.])
```

Shape manipulation

Arrays can be reshaped without affecting their contents:

```
>> a = np.floor(10*np.random.random((3,4)))
>> a
array([[2., 2., 6., 8.],
       [1., 6., 1., 6.],
       [5., 1., 5., 0.]])
>> a.ravel()
array([2., 2., 6., 8., 1., 6., 1., 6., 5., 1., 5., 0.])
>> a.T
```

Shape manipulation

Arrays can be reshaped without affecting their contents:

```
>> a = np.floor(10*np.random.random((3,4)))
>> a
array([[2., 2., 6., 8.],
       [1., 6., 1., 6.],
       [5., 1., 5., 0.]])
>> a.ravel()
array([2., 2., 6., 8., 1., 6., 1., 6., 5., 1., 5., 0.])
>> a.T
array([[2., 1., 5.],
       [2., 6., 1.],
       [6., 1., 5.],
       [8., 6., 0.]])
```

Shape manipulation

To this end, we use the methods `reshape` and `resize`, which take integer arguments representing the required shape: the former returns a reshaped version of its argument, while the latter modifies its argument itself.

Shape manipulation

To this end, we use the methods `reshape` and `resize`, which take integer arguments representing the required shape: the former returns a reshaped version of its argument, while the latter modifies its argument itself.

```
>> a = np.array((1,2,3,4,5,6,7,8,9))
```

Shape manipulation

To this end, we use the methods `reshape` and `resize`, which take integer arguments representing the required shape: the former returns a reshaped version of its argument, while the latter modifies its argument itself.

```
>> a = np.array((1,2,3,4,5,6,7,8,9))
>> a.reshape(3,3) # does not modify a
```

Shape manipulation

To this end, we use the methods `reshape` and `resize`, which take integer arguments representing the required shape: the former returns a reshaped version of its argument, while the latter modifies its argument itself.

```
>> a = np.array((1,2,3,4,5,6,7,8,9))
>> a.reshape(3,3) # does not modify a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Shape manipulation

To this end, we use the methods reshape and resize, which take integer arguments representing the required shape: the former returns a reshaped version of its argument, while the latter modifies its argument itself.

```
>> a = np.array((1,2,3,4,5,6,7,8,9))
>> a.reshape(3,3) # does not modify a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>> a.resize(3,3) # does modify a
```

Shape manipulation

To this end, we use the methods `reshape` and `resize`, which take integer arguments representing the required shape: the former returns a reshaped version of its argument, while the latter modifies its argument itself.

```
>> a = np.array((1,2,3,4,5,6,7,8,9))
>> a.reshape(3,3) # does not modify a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>> a.resize(3,3) # does modify a
>> a
```

Shape manipulation

To this end, we use the methods `reshape` and `resize`, which take integer arguments representing the required shape: the former returns a reshaped version of its argument, while the latter modifies its argument itself.

```
>> a = np.array((1,2,3,4,5,6,7,8,9))
>> a.reshape(3,3) # does not modify a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>> a.resize(3,3) # does modify a
>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

That's pretty much it

If you ever get stuck or have questions about the syntax or semantics of Python or NumPy, be sure to check the Python tutorial and the NumPy Quickstart tutorial.