Variables and Assignment

When programming, it is useful to be able to store information in variables. A **variable** is a string of characters and numbers associated with a piece of information. The **assignment operator**, denoted by the "=" symbol, is the operator that is used to assign values to variables in Python. The line `x=1` takes the known value, 1, and **assigns** that value to the variable with name "x". After executing this line, this number will be stored into this variable. Until the value is changed or the variable deleted, the character x behaves like the value 1.

```
x = 1
x
```

```
1
```

TRY IT! Assign the value 2 to the variable y. Multiply y by 3 to show that it behaves like the value 2.

```
y = 2
y
```

```
2
```

```
y*3
```

```
6
```

A variable is more like a container to store the data in the computer's memory, the name of the variable tells the computer where to find this value in the memory. For now, it is sufficient to know that the notebook has its own memory space to store all the variables in the notebook. As a result of the previous example, you will see the variable "x" and "y" in the memory. You can view a list of all the variables in the notebook using the magic command `%whos`.

TRY IT! List all the variables in this notebook

```
%whos
```

```
Variable   Type    Data/Info
---------------------------
x          int     1
y          int     2
```

Note that the equal sign in programming is *not* the same as a truth statement in mathematics. In math, the statement *x = 2* declares the universal truth within the given framework, *x is 2*. In programming, the statement `x=2` means a known value is being associated with a variable name, *store* 2 in x. Although it is perfectly valid to say *1 = x* in mathematics, assignments in Python always go *left*: meaning the value to the right of the equal sign is assigned to the variable on the left of the equal sign. Therefore, *1=x* will generate an error in Python. The assignment operator is always last in the order of operations relative to mathematical, logical, and comparison operators.

TRY IT! The mathematical statement x=x+1 has no solution for any value of *x*. In programming, if we initialize the value of *x* to be 1, then the statement makes perfect sense. It means, "Add x and 1, which is 2, then assign that value to the variable x". Note that this operation overwrites the previous value stored in *x*.

```
x = x + 1
x
```

```
2
```

There are some restrictions on the names variables can take. Variables can only contain alphanumeric characters (letters and numbers) as well as underscores. However, the first character of a variable name must be a letter or underscores. Spaces within a variable name are not permitted, and the variable names are case-sensitive (e.g., *x* and *X* will be considered different variables).

TIP! Unlike in pure mathematics, variables in programming almost always represent *something* tangible. It may be the distance between two points in space or the number of rabbits in a population. Therefore, as your code becomes increasingly complicated, it is very important that your variables carry a name that can easily be associated with what they represent. For example, the distance between two points in space is better represented by the variable *dist* than *x*, and the number of rabbits in a population is better represented by *nRabbits* than *y*.

Note that when a variable is assigned, it has no memory of *how* it was assigned. That is, if the value of a variable, *y*, is constructed from other variables, like *x*, reassigning the value of *x* will not change the value of *y*.

EXAMPLE: What value will *y* have after the following lines of code are executed?

```
x = 1
y = x + 1
x = 2
y
```

```
2
```

WARNING! You can overwrite variables or functions that have been stored in Python. For example, the command `help = 2` will store the value 2 in the variable with name *help*. After this assignment *help* will behave like the value 2 instead of the function *help*. Therefore, you should always be careful not to give your variables the same name as built-in functions or values.

TIP! Now that you know how to assign variables, it is important that you learn to *never* leave unassigned commands. An **unassigned command** is an operation that has a result, but that result is not assigned to a variable. For example, you should never use *2+2*. You should instead assign it to some variable `x=2+2`. This allows you to "hold on" to the results of previous commands and will make your interaction with Python must less confusing.

You can clear a variable from the notebook using the *del* function. Typing `del x` will clear the variable *x* from the workspace. If you want to remove all the variables in the notebook, you can use the magic command *%reset*.

In mathematics, variables are usually associated with unknown numbers; in programming, variables are associated with a value of a certain type. There are many data types that can be assigned to variables. A **data type** is a classification of the type of information that is being stored in a variable. The basic data types that you will utilize throughout this book are boolean, int, float, string, list, tuple, dictionary, set. A formal description of these data types is given in the following sections.

< 2.0 Variables and Basic Data Structures | Contents | 2.2 Data Structure - Strings >

Data Structure - Strings

We talked about different data types, such as int, float and boolean, these are all related to single value. The rest of this chapter will introduce you more data types so that we could store multiple values. The data structure related to these new types are Strings, Lists, Tuples, Sets, and Dictionaries. We will start with the strings.

A string is a sequence of characters, such as "Hello World" we saw in chapter 1. Strings are surrounded by either single or double quotation marks. We could use *print* function to output the strings to the screen.

TRY IT! Print "I love Python!" to the screen.

```
print("I love Python!")
```

TRY IT! Assign the character "S" to the variable with name s. Assign the string "Hello World" to the variable w. Verify that s and w have the type string using the *type* function.

```
s = "S"
w = "Hello World"
```

```
type(s)
```

```
str
```

```
type(w)
```

```
str
```

Note that a blank space, " ", between "Hello" and "World" is also a type *str*. Any symbol can be a char, even the ones that have been reserved for operators. Note that as a *str*, they do not perform the same function. Although they look the same, Python interprets them completely differently.

TRY IT! Create an empty string. Verify that the empty string is a *str*.

```
s = " "
type(s)
```

```
str
```

A string is an array of characters, therfore it has length to indicate the size of the string. For example, we could check the size of the string by using the built-in function *len*.

```
len(w)
```

```
11
```

Strings also have indexes to indicate the location of each character, so that we could easily find out some character. The index of the position start with 0, as shown in the following picture.

Character	H	e	l	l	o		W	o	r	l	d
Index	0	1	2	3	4	5	6	7	8	9	10

We could get access to any character by using a bracket and the index of the position. For example, if we want to get the character 'W', then we need to do:

```
w[6]
```

```
'W'
```

We could also select a sequence as well using string slicing. For example, if we want to get the "World", we could do the following command.

```
w[6:11]
```

```
'World'
```

[6:11] means the start position is from index 6 and the end position is index 10. For Python string slicing range, the upper-bound is exclusive, which means that [6:11] is actually to slice the characters from 6 -> 10. The syntax for slicing in Python is [start:end:step], the 3rd one - step is optional.

You can ignore the end position if you want to slice to the end of the string. For example, the following command is the same as the above one:

```
w[6:]
```

```
'World'
```

TRY IT! Retrieve the word "Hello" from string *w*.

```
w[:5]
```

```
'Hello'
```

You can also use negative index when slice the strings, which means counting from the end of the string. For example, -1 means the last character, -2 means the 2nd to last and so on.

TRY IT! Slice the "Wor" within the word "World".

```
w[6:-2]
```

```
'Wor'
```

TRY IT! Retrieve every other character in the variable *w*

```
w[::2]
```

```
'HloWrd'
```

Strings can not be used in the mathematical operations.

TRY IT! Use '+' to add two numbers. Verify that "+" does not behave like the addition operator, `+`.

```
1 "+" 2
```

```
  File "<ipython-input-13-46b54f731e00>", line 1
    1 "+" 2
      ^
SyntaxError: invalid syntax
```

WARNING! Numbers can also be expressed as *str*. For example, `x = '123'` means that x is the string 123 not the number 123. However, strings represent words or text and so should not have addition defined on them.

TIP! You may find yourself in a situation where you would like to use an apostrophe as a *str*. This is problematic since an apostrophe is used to denote strings. Fortunately, an apostrophe can be used in a string in the following way. The backslash (\) is a way to tell Python this is part of the string, not to denote strings. The backslash character is used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character.

```
'don\'t'
```

```
"don't"
```

One string could be concatenated to another string. For example:

```
str_a = "I love Python! "
str_b = "You too!"

print(str_a + str_b)
```

```
I love Python! You too!
```

We could convert other data types to strings as well using the built-in function *str*. This is useful, for example, we have an variable x that stored 1 as an integer type, if we want to print it out directly with a string, we will get an error saying we can not concatenate string with an integer.

```
x = 1
print("x = " + x)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-3e562ba0dd83> in <module>()
      1 x = 1
----> 2 print("x = " + x)

TypeError: can only concatenate str (not "int") to str
```

The correct way to do it is to convert the integer to string first, and then print it out.

TRY IT! Print out *x = 1* to the screen.

```
print("x = " + str(x))
```

```
x = 1
```

```
type(str(x))
```

```
str
```

In Python, string as an object that has various methods that could be used to manipulate it (we will talk more about object-oriented programming later). They way to get access to the various methods is to use this patter "string.method_name".

TRY IT! Turn the variable w to upper case.

```
w.upper()
```

```
'HELLO WORLD'
```

TRY IT! Count the number of occurrence for letter "l" in w.

```
w.count("l")
```

```
3
```

TRY IT! Replace the "World" in variable w to "Berkeley".

```
w.replace("World", "Berkeley")
```

```
'Hello Berkeley'
```

There are different ways to pre-format a string. Here we introduce two ways to do it. For example, if we have two variables *name* and *country*, and wants to print them out in a sentence, but we don't want to use the string concatenation we used before, since it will use many '+' signs in the string. We could do the following instead:

```
name = "UC Berkeley"
country = 'USA'

print("%s is a great school in %s!"%(name, country))
```

```
UC Berkeley is a great school in USA!
```

WHAT IS HAPPENING? In the previous example, the %s in the double quotation marks is telling Python that we want to insert some strings at this location (s stands for string in this case). The *%(name, country)* is where the two strings we want to insert.

NEW! There is a different way that only introduced in Python 3.6 and above, it is called f-String which means formated-String. You can easily format a string with the following line:

```
print(f"{name} is a great school in {country}.")
```

```
UC Berkeley is a great school in USA.
```

You could even print out a numerical expression without convert the data type as we did before.

TRY it! Print out the result of *34** directly using f-String.

```
print(f"{3*4}")
```

```
12
```

Ok, we learned many things from the data structure - string, this is our first sequence data structure. Let's learn more now.

< 2.1 Variables and Assignment | Contents | 2.3 Data Structure - Lists >

Data Structure - Lists

We saw strings in the previous section that could hold a sequence of characters. Now, let's see a more versatile sequential data structure in Python - Lists. The way to define it is to use a pair of brackets [ ], and the elements within it are separated by commas. A list could hold any type of data: numerical, or strings or other types. For example:

```
list_1 = [1, 2, 3]
list_1
```

```
[1, 2, 3]
```

```
list_2 = ['Hello', 'World']
list_2
```

```
['Hello', 'World']
```

We could put mixed types in the list as well

```
list_3 = [1, 2, 3, 'Apple', 'orange']
list_3
```

```
[1, 2, 3, 'Apple', 'orange']
```

We can also nest the lists, for example:

```
list_4 = [list_1, list_2]
list_4
```

```
[[1, 2, 3], ['Hello', 'World']]
```

The way to retrieve the element in the list is very similar to the strings, see the following figure for the index

List	1	2	3	Apple	Orange
Index	0	1	2	3	4

TRY IT! Get the 3rd element in list_3

```
list_3[2]
```

```
3
```

TRY IT! Get the first 3 elements in list_3

```
list_3[:3]
```

```
[1, 2, 3]
```

TRY IT! Get the last element in list_3

```
list_3[-1]
```

```
'orange'
```

TRY IT! Get the first list from list_4.

```
list_4[0]
```

```
[1, 2, 3]
```

Similarly, we could get the length of the list by using the *len* function.

```
len(list_3)
```

```
5
```

We can also concatenate two lists by simply using a + sign.

TRY IT! Add list_1 and list_2 to one list.

```
list_1 + list_2
```

```
[1, 2, 3, 'Hello', 'World']
```

New items could be added to an existing list by using the *append* method from the list.

```
list_1.append(4)
list_1
```

```
[1, 2, 3, 4]
```

Note! The *append* function operate on the list itself as shown in the above example, 4 is added to the list. But in the list_1 + list_2 example, list_1 and list_2 won't change. You can check list_2 to verify this.

We could also insert or remove element from the list by using the methods *insert* and *remove*, but they are also operating on the list directly.

```
list_1.insert(2,'center')
list_1
```

```
[1, 2, 'center', 3, 4]
```

Note! Using the *remove* method will only remove the first occurrence of the item (read the documentation of the method). There is another way to delete an item by using its index - function *del*.

```
del list_1[2]
list_1
```

```
[1, 2, 3, 4]
```

We could also define an empty list and add in new element later using *append* method. It is used a lot in Python when you have to loop through a sequence of items, we will learn more later in the iteration chapter.

TRY IT! Define an empty list and add values 5 and 6 to the list.

```
list_5 = []
list_5.append(5)
list_5
```

```
[5]
```

```
list_5.append(6)
list_5
```

```
[5, 6]
```

We could also quickly check if an element is in the list using the operator *in*.

TRY IT! Check if number 5 is in the *list_5*.

```
5 in list_5
```

```
True
```

Using the *list* function, we could turn other sequence items into a list.

TRY IT! Turn the string 'Hello World' into a list of characters.

```
list('Hello World')
```

```
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
```

The lists will be used a lot in Python when we are working with data, they have many different use cases that you will see in the later sections.

< 2.2 Data Structure - Strings | Contents | 2.4 Data Structure - Tuples >

This notebook contains an excerpt from the Python Programming and Numerical Methods - A Guide for Engineers and Scientists, the content is also available at Berkeley Python Numerical Methods.

The copyright of the book belongs to Elsevier. We also have this interactive book online for a better learning experience. The code is released under the MIT license. If you find this content useful, please consider supporting the work on Elsevier or Amazon!

Data Structure - Tuples

Let's learn one more different sequence data structure in Python - Tuples. It is usually defined by using a pair of parentheses ( ) , and its elements are separated by commas. For example:

```
tuple_1 = (1, 2, 3, 2)
tuple_1
```

```
(1, 2, 3, 2)
```

As strings and lists, they way for indexing the tuples, slicing the elements, and even some methods are very similar.

TRY IT! Get the length of tuple_1.

```
len(tuple_1)
```

```
4
```

TRY IT! Get the elements from index 1 to 3 for tuple_1.

```
tuple_1[1:4]
```

```
(2, 3, 2)
```

TRY IT! Count the occurrence for number 2 in tuple_1.

```
tuple_1.count(2)
```

```
2
```

You may ask, what's the difference between lists and tuples? If they are similar to each other, why do we need another sequence data structure?

Well, tuples are created for a reason. From the Python documentation:

> Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are **immutable**, and usually contain a **heterogeneous** sequence of elements that are accessed via unpacking (see later in this section) or indexing (or even by attribute in the case of named tuples). Lists are **mutable**, and their elements are usually **homogeneous** and are accessed by iterating over the list.

What does it mean by immutable? It means the elements in the tuple, once defined, they can not be changed. But elements in a list can be changed without any problem. For example:

```
list_1 = [1, 2, 3]
list_1[2] = 1
list_1
```

```
[1, 2, 1]
```

```
tuple_1[2] = 1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-76fb6b169c14> in <module>()
----> 1 tuple_1[2] = 1

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

What does heterogeneous mean? Tuples usually contain a heterogeneous sequence of elements, while lists usually contain a homogeneous sequence. Let's see an example, that we have a list that contains different fruits. Usually the name of the fruits could be stored in a list, since they are homogeneous. Now we want to have a data structure to store how many fruit do we have for each type, this is usually where the tuples comes in, since the name of the fruit and the number are heterogeneous. Such as ('apple', 3) which means we have 3 apples.

```
# a fruit list
['apple', 'banana', 'orange', 'pear']
```

```
['apple', 'banana', 'orange', 'pear']
```

```
# a list of (fruit, number) pairs
[('apple', 3), ('banana', 4) , ('orange', 1), ('pear', 4)]
```

```
[('apple', 3), ('banana', 4), ('orange', 1), ('pear', 4)]
```

Tuples could be accessed by unpacking, it requires that the number of variables on the left side of the equals sign equals to the number of elements in the sequence.

```
a, b, c = list_1
print(a, b, c)
```

```
1 2 1
```

NOTE! The opposite operation to unpacking is packing as shown in the following example. We could see that we don't need the parentheses to define a tuple, but it is always good to have that.

```
list_2 = 2, 4, 5
list_2
```

```
(2, 4, 5)
```

< 2.3 Data Structure - Lists | Contents | 2.5 Data Structure - Sets >

< 2.4 Data Structure - Tuples | Contents | 2.6 Data Structure - Dictionaries >

Data Structure - Sets

Another data type in Python is sets. It is a type that could store an unordered collection with no duplicate elements. It is also support the mathematical operations like union, intersection, difference, and symmetric difference. It is defined by using a pair of braces { }, and its elements are separated by commas.

```
{3, 3, 2, 3, 1, 4, 5, 6, 4, 2}
```

```
{1, 2, 3, 4, 5, 6}
```

One quick usage of this is to find out the unique elements in a string, list, or tuple.

TRY IT! Find the unique elements in list [1, 2, 2, 3, 2, 1, 2].

```
set_1 = set([1, 2, 2, 3, 2, 1, 2])
set_1
```

```
{1, 2, 3}
```

TRY IT! Find the unique elements in tuple (2, 4, 6, 5,2).

```
set_2 = set((2, 4, 6, 5, 2))
set_2
```

```
{2, 4, 5, 6}
```

TRY IT! Find the unique character in string "Banana".

```
set('Banana')
```

```
{'B', 'a', 'n'}
```

We mentioned that sets support the mathematical operations like union, intersection, difference, and symmetric difference.

TRY IT! Get the union of set_1 and set_2.

```
print(set_1)
print(set_2)
```

```
{1, 2, 3}
{2, 4, 5, 6}
```

```
set_1.union(set_2)
```

```
{1, 2, 3, 4, 5, 6}
```

TRY IT! Get the intersection of set_1 and set_2.

```
set_1.intersection(set_2)
```

```
{2}
```

TRY IT! Is set_1 a subset of {1, 2, 3, 3, 4, 5}?

```
set_1.issubset({1, 2, 3, 3, 4, 5})
```

```
True
```

< 2.4 Data Structure - Tuples | Contents | 2.6 Data Structure - Dictionaries >

Data Structure - Dictionaries

We introduced several sequential data type in the previous sections. Now we will introduce you a new and useful type - Dictionaries. It is a mapping type, which makes it a totally different type than the ones we talked before. Instead of using a sequence of numbers to index the elements (such as lists or tuples), dictionaries are indexed by keys, which could be a string, number or even tuple (but not list). A dictionary is a key-value pairs, and each key maps to a corresponding value. It is defined by using a pair of braces { }, while the elements are a list of comma separated key:value pairs (note the key:value pair is separated by the colon, with key at front and value at the end).

```
dict_1 = {'apple':3, 'oragne':4, 'pear':2}
dict_1
```

```
{'apple': 3, 'oragne': 4, 'pear': 2}
```

Within a dictionary, elements are stored without order, therefore, you can not access a dictionary based on a sequence of index numbers. To get access to a dictionary, we need to use the key of the element - dictionary[key].

TRY IT! Get the element 'apple' from dict_1.

```
dict_1['apple']
```

```
3
```

We could get all the keys in a dictionary by using the *keys* method, or all the values by using the method *values*.

TRY IT Get all the keys and values from dict_1.

```
dict_1.keys()
```

```
dict_keys(['apple', 'oragne', 'pear'])
```

```
dict_1.values()
```

```
dict_values([3, 4, 2])
```

We could also get the size of a dictionary by using the *len* function.

```
len(dict_1)
```

```
3
```

We could define an empty dictionary and then fill in the element later. Or we could turn a list of tuples with (key, value) pairs to a dictionary.

TRY IT! Define an empty dictionary named *school_dict* and add value "UC Berkeley":"USA".

```
school_dict = {}
school_dict['UC Berkeley'] = 'USA'
school_dict
```

```
{'UC Berkeley': 'USA'}
```

TRY IT! Add another element "Oxford":"UK" to *school_dict*.

```
school_dict['Oxford'] = 'UK'
school_dict
```

```
{'UC Berkeley': 'USA', 'Oxford': 'UK'}
```

TRY IT! Turn the list of tuples *[("UC Berkeley", "USA"), ('Oxford', 'UK')]* into a dictionary.

```
dict([("UC Berkeley", "USA"), ('Oxford', 'UK')])
```

```
{'UC Berkeley': 'USA', 'Oxford': 'UK'}
```

We could also check if an element belong to a dictionary using the operator *in*.

TRY IT! Determine if "UC Berkeley" is in *school_dict*.

```
"UC Berkeley" in school_dict
```

```
True
```

TRY IT! Determine whether "Harvard" is not in *school_dict*.

```
"Harvard" not in school_dict
```

```
True
```

We could also use the *list* function to turn a dictionary with a list of keys. For example:

```
list(school_dict)
```

```
['UC Berkeley', 'Oxford']
```

< 2.5 Data Structure - Sets | Contents | 2.7 Introducing Numpy Arrays >

Introducing Numpy Arrays

In the 2nd part of this book, we will study the numerical methods by using Python. We will use array/matrix a lot later in the book. Therefore, here we are going to introduce the most common way to handle arrays in Python using the Numpy module. Numpy is probably the most fundamental numerical computing module in Python.

NumPy is important in scientific computing, it is coded both in Python and C (for speed). On its website, a few important features for Numpy is listed:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Here we will only introduce you the Numpy array which is related to the data structure, but we will gradually touch on other aspects of Numpy in the following chapters.

In order to use Numpy module, we need to import it first. A conventional way to import it is to use "np" as a shortened name.

```
import numpy as np
```

WARNING! Of course, you could call it any name, but conventionally, "np" is accepted by the whole community and it is a good practice to use it for obvious purposes.

To define an array in Python, you could use the *np.array* function to convert a list.

TRY IT! Create the following arrays:

$x = \begin{pmatrix} 1 & 4 & 3 \end{pmatrix}$

$y = \begin{pmatrix} 1 & 4 & 3 \\ 9 & 2 & 7 \end{pmatrix}$

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

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

```
y = np.array([[1, 4, 3], [9, 2, 7]])
y
```

```
array([[1, 4, 3],
       [9, 2, 7]])
```

NOTE! A 2-D array could use a nested lists to represent, with the inner list represent each row.

Many times we would like to know the size or length of an array. The array *shape* attribute is called on an array M and returns a 2 × 3 array where the first element is the number of rows in the matrix M and the second element is the number of columns in M. Note that the output of the *shape* attribute is a tuple. The *size* attribute is called on an array M and returns the total number of elements in matrix M.

TRY IT! Find the rows, columns and the total size for array y.

```
y.shape
```

```
(2, 3)
```

```
y.size
```

```
6
```

NOTE! You may notice the difference that we only use *y.shape* instead of *y.shape()*, this is because *shape* is an attribute rather than a method in this array object. We will introduce more of the object-oriented programming in a later chapter. For now, you need to remember that when we call a method in an object, we need to use the parentheses, while the attribute don't.

Very often we would like to generate arrays that have a structure or pattern. For instance, we may wish to create the array z = [1 2 3 … 2000]. It would be very cumbersome to type the entire description of z into Python. For generating arrays that are in order and evenly spaced, it is useful to use the *arange* function in Numpy.

TRY IT! Create an array *z* from 1 to 2000 with an increment 1.

```
z = np.arange(1, 2000, 1)
z
```

```
array([   1,    2,    3, ..., 1997, 1998, 1999])
```

Using the *np.arange*, we could create *z* easily. The first two numbers are the start and end of the sequence, and the last one is the increment. Since it is very common to have an increment of 1, if an increment is not specified, Python will use a default value of 1. Therefore *np.arange(1, 2000)* will have the same result as *np.arange(1, 2000, 1)*. Negative or noninteger increments can also be used. If the increment "misses" the last value, it will only extend until the value just before the ending value. For example, *x = np.arange(1,8,2)* would be [1, 3, 5, 7].

TRY IT! Generate an array with [0.5, 1, 1.5, 2, 2.5].

```
np.arange(0.5, 3, 0.5)
```

```
array([ 0.5,  1. ,  1.5,  2. ,  2.5])
```

Sometimes we want to guarantee a start and end point for an array but still have evenly spaced elements. For instance, we may want an array that starts at 1, ends at 8, and has exactly 10 elements. For this purpose you can use the function *np.linspace. linspace* takes three input values separated by commas. So *A = linspace(a,b,n)* generates an array of n equally spaced elements starting from a and ending at b.

TRY IT! Use *linspace* to generate an array starting at 3, ending at 9, and containing 10 elements.

```
np.linspace(3, 9, 10)
```

```
array([ 3.        ,  3.66666667,  4.33333333,  5.        ,  5.66666667,
        6.33333333,  7.        ,  7.66666667,  8.33333333,  9.        ])
```

Getting access to the 1D numpy array is similar to what we described for lists or tuples, it has an index to indicate the location. For example:

```
# get the 2nd element of x
x[1]
```

```
4
```

```
# get all the element after the 2nd element of x
x[1:]
```

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

```
# get the last element of x
x[-1]
```

```
3
```

For 2D arrays, it is slightly different, since we have rows and columns. To get access to the data in a 2D array M, we need to use M[r, c], that the row r and column c are separated by comma. This is referred to as array indexing. The r and c could be single number, a list and so on. If you only think about the row index or the column index, than it is similar to the 1D array. Let's use the $y = \begin{pmatrix} 1 & 4 & 3 \\ 9 & 2 & 7 \end{pmatrix}$ as an example.

TRY IT! Get the element at first row and 2nd column of array *y*.

```
y[0,1]
```

```
4
```

TRY IT! Get the first row of array *y*.

```
y[0, :]
```

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

TRY IT! Get the last column of array *y*.

```
y[:, -1]
```

```
array([3, 7])
```

TRY IT! Get the first and third column of array *y*.

```
y[:, [0, 2]]
```

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

There are some predefined arrays that are really useful. For example, the *np.zeros*, *np.ones*, and *np.empty* are 3 useful functions. Let's see the examples.

TRY IT! Generate a 3 by 5 array with all the as 0.

```
np.zeros((3, 5))
```

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

TRY IT! Generate a 5 by 3 array with all the element as 1.

```
np.ones((5, 3))
```

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

NOTE! The shape of the array is defined in a tuple with row as the first item, and column as the second. If you only need a 1D array, then it could be only one number as the input: *np.ones(5)*.

TRY IT! Generate a 1D empty array with 3 elements.

```
np.empty(3)
```

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

NOTE! The empty array is not really empty, it is filled with random very small numbers.

You can reassign a value of an array by using array indexing and the assignment operator. You can reassign multiple elements to a single number using array indexing on the left side. You can also reassign multiple elements of an array as long as both the number of elements being assigned and the number of elements assigned is the same. You can create an array using array indexing.

TRY IT! Let a = [1, 2, 3, 4, 5, 6]. Reassign the fourth element of A to 7. Reassign the first, second, and thrid elements to 1. Reassign the second, third, and fourth elements to 9, 8, and 7.

```
a = np.arange(1, 7)
a
```

```
array([1, 2, 3, 4, 5, 6])
```

```
a[3] = 7
a
```

```
array([1, 2, 3, 7, 5, 6])
```

```
a[:3] = 1
a
```

```
array([1, 1, 1, 7, 5, 6])
```

```
a[1:4] = [9, 8, 7]
a
```

```
array([1, 9, 8, 7, 5, 6])
```

TRY IT! Create a zero array b with shape 2 by 2, and set $b = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ using array indexing.

```
b = np.zeros((2, 2))
b[0, 0] = 1
b[0, 1] = 2
b[1, 0] = 3
b[1, 1] = 4
b
```

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

WARNING! Although you can create an array from scratch using indexing, we do not advise it. It can confuse you and errors will be harder to find in your code later. For example, b[1, 1] = 1 will give the result $b = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$, which is strange because b[0, 0], b[0, 1], and b[1, 0] were never specified.

Basic arithmetic is defined for arrays. However, there are operations between a scalar (a single number) and an array and operations between two arrays. We will start with operations between a scalar and an array. To illustrate, let c be a scalar, and b be a matrix.

b + c, *b − c*, *b * c* and *b / c* adds a to every element of b, subtracts c from every element of b, multiplies every element of b by c, and divides every element of b by c, respectively.

TRY IT! Let $b = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$. Add and substract 2 from b. Multiply and divide b by 2. Square every element of b. Let c be a scalar. On your own, verify the reflexivity of scalar addition and multiplication: b + c = c + b and cb = bc.

```
b + 2
```

```
array([[ 3.,  4.],
       [ 5.,  6.]])
```

```
b - 2
```

```
array([[-1.,  0.],
       [ 1.,  2.]])
```

```
2 * b
```

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

```
b / 2
```

```
array([[ 0.5,  1. ],
       [ 1.5,  2. ]])
```

```
b**2
```

```
array([[  1.,   4.],
       [  9.,  16.]])
```

Describing operations between two matrices is more complicated. Let b and d be two matrices of the same size. b − d takes every element of b and subtracts the corresponding element of d. Similarly, b + d adds every element of d to the corresponding element of b.

TRY IT! Let $b = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ and $d = \begin{pmatrix} 3 & 4 \\ 5 & 6 \end{pmatrix}$. Compute b + d and b - d.

```
b = np.array([[1, 2], [3, 4]])
d = np.array([[3, 4], [5, 6]])
```

```
b + d
```

```
array([[ 4,  6],
       [ 8, 10]])
```

```
b - d
```

```
array([[-2, -2],
       [-2, -2]])
```

There are two different kinds of matrix multiplication (and division). There is element-by-element matrix multiplication and standard matrix multiplication. For this section, we will only show how element-by-element matrix multiplication and division work. Standard matrix multiplication will be described in later chapter on Linear Algebra. Python takes the * symbol to mean element-by-element multiplication. For matrices b and d of the same size, b * d takes every element of b and multiplies it by the corresponding element of d. The same is true for / and **.

TRY IT! Compute b * d, b / d, and b**d.

```
b * d
```

```
array([[ 3,  8],
       [15, 24]])
```

```
b / d
```

```
array([[ 0.33333333,  0.5       ],
       [ 0.6       ,  0.66666667]])
```

```
b**d
```

```
array([[   1,   16],
       [ 243, 4096]])
```

The transpose of an array, b, is an array, d, where b[i, j] = d[j, i]. In other words, the transpose switches the rows and the columns of b. You can transpose an array in Python using the array method *T*.

TRY IT! Compute the transpose of array b.

```
b.T
```

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

Numpy has many arithmetic functions, such as sin, cos, etc., can take arrays as input arguments. The output is the function evaluated for every element of the input array. A function that takes an array as input and performs the function on it is said to be **vectorized**.

TRY IT! Compute *np.sqrt* for x = [1, 4, 9, 16].

```
x = [1, 4, 9, 16]
np.sqrt(x)
```

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

Logical operations are only defined between a scalar and an array and between two arrays of the same size. Between a scalar and an array, the logical operation is conducted between the scalar and each element of the array. Between two arrays, the logical operation is conducted element-by-element.

TRY IT! Check which elements of the array x = [1, 2, 4, 5, 9, 3] are larger than 3. Check which elements in x are larger than the corresponding element in y = [0, 2, 3, 1, 2, 3].

```
x = np.array([1, 2, 4, 5, 9, 3])
y = np.array([0, 2, 3, 1, 2, 3])
```

```
x > 3
```

```
array([False, False,  True,  True,  True, False], dtype=bool)
```

```
x > y
```

```
array([ True, False,  True,  True,  True, False], dtype=bool)
```

Python can index elements of an array that satisfy a logical expression.

TRY IT! Let x be the same array as in the previous example. Create a variable y that contains all the elements of x that are strictly bigger than 3. Assign all the values of x that are bigger than 3, the value 0.

```
y = x[x > 3]
y
```

```
array([4, 5, 9])
```

```
x[x > 3] = 0
x
```

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

< 2.6 Data Structure - Dictionaries | Contents | 2.8 Summary and Problems >