

Python3

1. Getting started with Python3
2. Data types (*input*, *print*)
3. Operations, Mutability and Methods
4. Flow control
5. Comprehension
6. Writing your own Functions
7. Built-in Modules: *math*, *random*, *sys*
8. Writing your own Modules
9. Reading and writing of files

Chapter 1. Getting started with Python3

Python is a general-purpose, versatile, and powerful high-level programming language. It was created by Guido van Rossum and released in 1991. Today it is extensively used in academia as well as in industry. Python is popular for many reasons: it is beginner-friendly, has a toolset to deal with mathematics and statistics, it is great for visualizing data, and has a strong community. The Python community works continuously on developing and improving Python libraries while enriching this open-source ecosystem. Python is considered one of the best programming languages for data analysis and machine learning.

1.1 Installing Anaconda and Spyder

A1. Download and Install Anaconda and Spyder

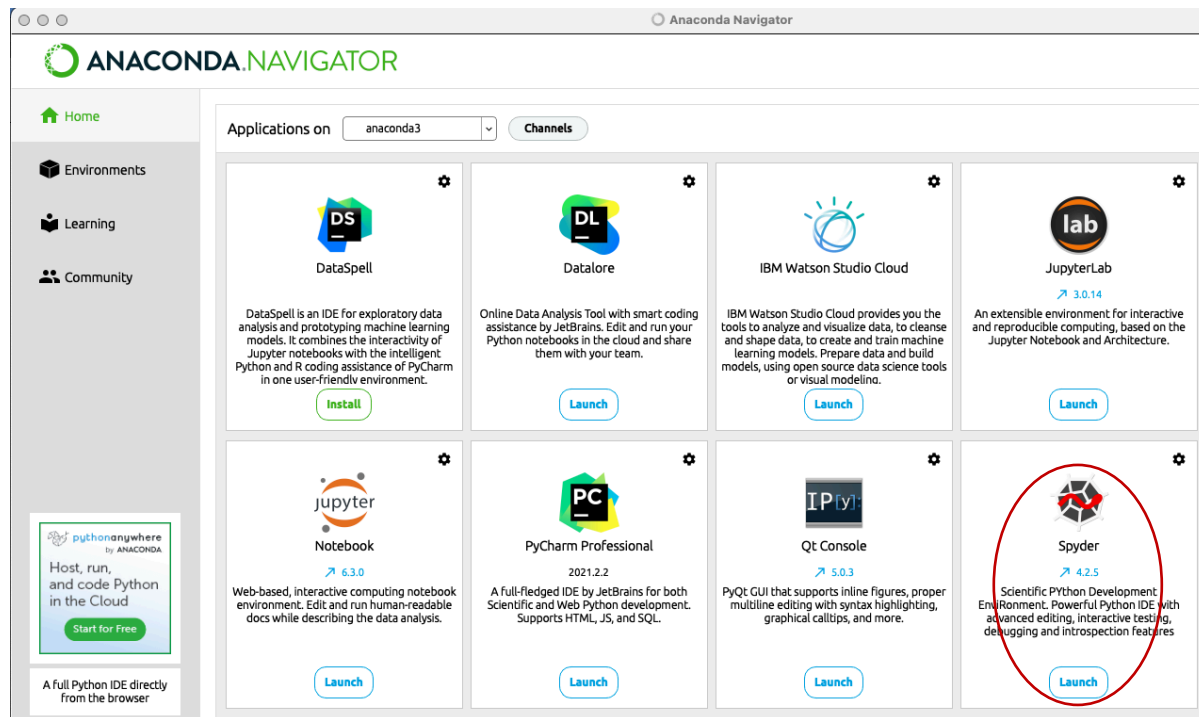
One easy way to write, test, and run python code is an "integrated development environment" (or IDE). We will be using, Spyder, the Scientific Python Development Environment that is included with Anaconda. Anaconda will also allow access to many python libraries such as Matplotlib, and NumPy, which we will be using in this course.

There is a version of Anaconda for Mac, Windows, and Linux

Click this link: <https://www.anaconda.com/products/individual>

- Locate your operating system (Windows or OS or Linux)
- Click the Graphical Installer
- Follow Directions to install Anaconda to laptop
- Open from *Application* Folder → click *Anaconda* → *Navigators*
- Press *Install* under Spyder

- Launch Spyder's Application




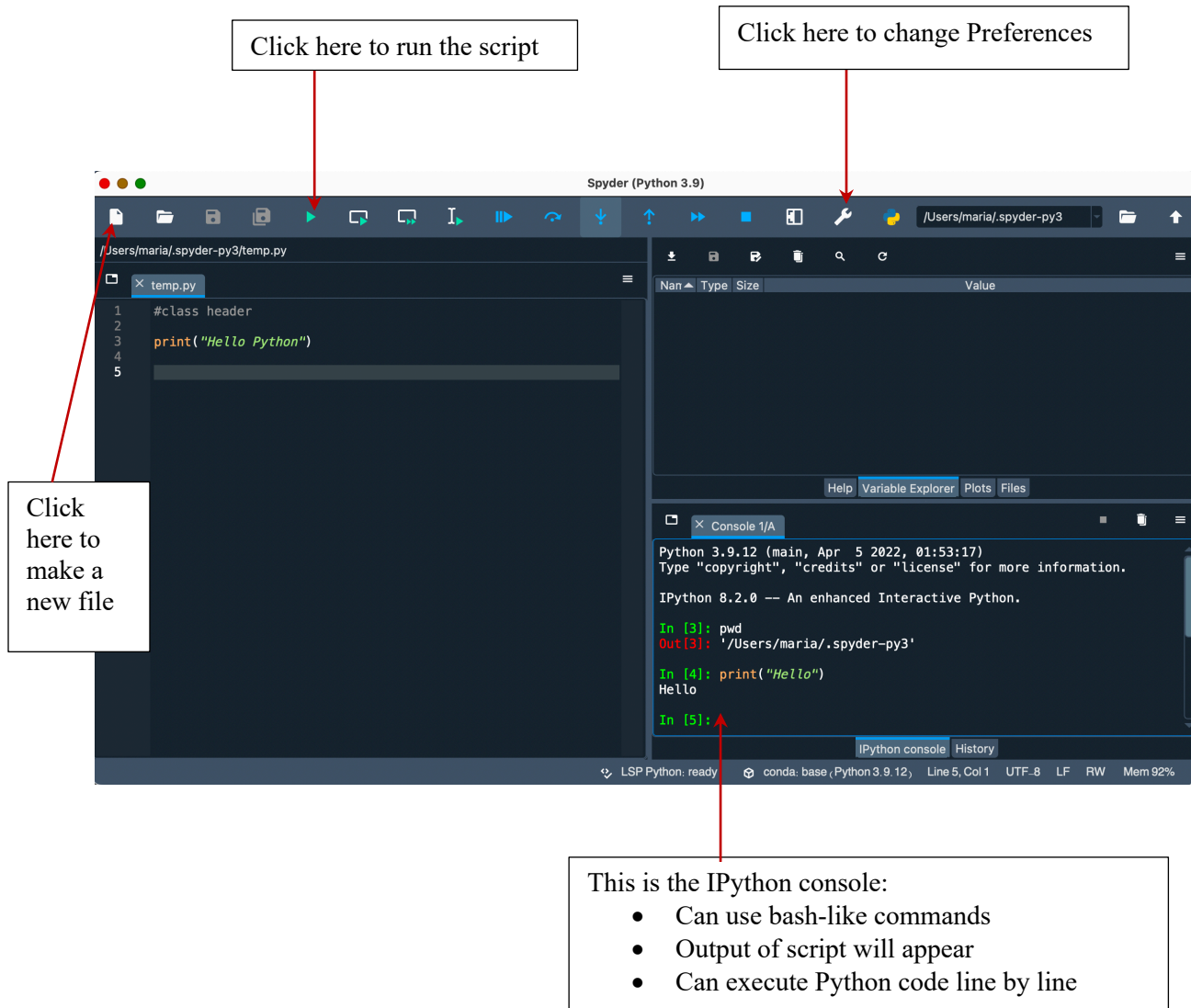
A2. Using the Spyder integrated development environment

In the default configuration, there are three windows displayed:

- An editor window (left) in which you write your scripts and run them by using the green “play” button
- A display window (upper right), which could be a variable browser window, a help window, a plotting window, and a files view window, depending on the button you click on it.
- IPython window console (lower right), in which individual python commands can be executed, and outputs from scripts will appear. You can also use bash-like commands.

You can change the default appearance (e.g., change color, font size, etc.) by:

- clicking on Python → Preferences → Appearance or
- clicking on the preference  symbol



A3. Bash-like commands on the IPython console

Try these on the IPython console:

```
pwd
/Users/maria/.spyder-py3
```

The default current directory is a hidden directory called `.spyder-py3`. You can change the directory by using the `cd` command, and write pathnames as you would in bash

```
cd ~
pwd
/Users/maria
```

```
ls *.py
```

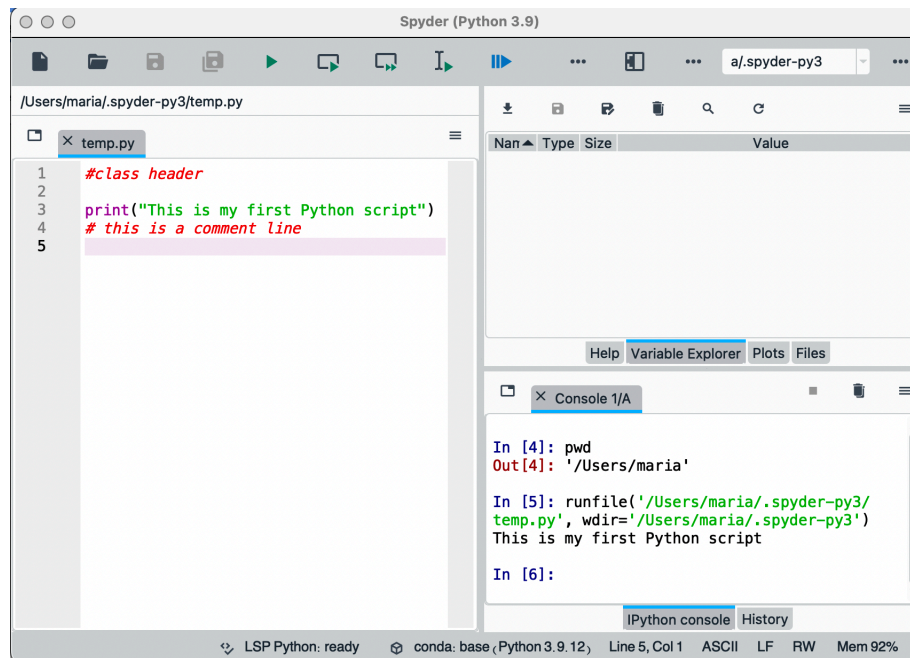
Play with other bash commands on the IPython console to see which ones work and which do not work.

A4. Your First Python program (*.py file) – make, run, and save scripts

Make a python script called temp.py – write your class header, a print statement, and a comment line. Save and run the script.

Scripts will be saved in your current working directory by clicking on *save*.

If you want to save scripts in another directory, you can either change directory at the IPython console, or click on *save as*



1.2 Alternative to Anaconda

1. You could use python3.arg IDLE which comes with every Python distribution. You may find IDLE under Applications-> Python 3.? (here “?” is whatever version you installed. But make sure it is a recent version).

You can find the IDLE by clicking on Launchpad, and searching for Idle, 

<https://docs.python.org/3/library/idle.html>

<https://realpython.com/python-idle/>

However, you still need the bash terminal to run the scripts, like

```
$ python3 temp.py
```

2. Another option is to install Visual Studio Code

<https://code.visualstudio.com/>

1.3 General syntax

Python syntax is clear and intuitive. This section discusses some of the major features.

1.3.1 Python is case sensitive.

Care must be exercised to ensure proper use of capitalization in Python syntax, in both Python statements and in names of variables, modules, and functions. A variable named `Pressure` is not the same as a variable named `pressure`, likewise, the `print()` function is lowercase. Use of `Print()` (with capital P) will result in an error, unless a user-defined function of this name is created.

1.3.2 Naming variables

When naming variables there are some rules to follow to make valid variable names:

- The first character of the variable must be an alphabet or underscore.
- All the characters except the first character may be an alphabet of lower-case (a-z), upper-case (A-Z), underscore, or digit (0-9).
- must not contain any white-space, or special character (!, @, #, %, ^, &, *).
- must not be any keyword defined in the language. There are reserved words in Python that should not be used as a variable or function names, such as, `and`, `from`, `import`, `in`, `is`, `not`, `or`, `pass`, etc.
- names are case sensitive; for example, `myname`, and `MyName` are not the same.

Examples of valid variable names:

```
a123 = 10
n_9 = 100
```

Examples of invalid variable names:

```
1a = 100
n%4 = 10
in = 100
```

Notice that in variable assignment spaces do not matter in Python.

```
x = 1 #note the spaces or x=1
```

1.3.3 Continuation of lines

There is no limit to how long a line of code in Python can be. However, it is best to break up long lines for easier reading. The symbol `\` is used at the end of a line of code to signify that the next line is a continuation. `print('If you want to break a line of code use \`
`and this is the continuation of the line of code')`

1.3.4 Comments

Comments in Python are indicated by the symbol `#`. Any code between this symbol and a new line is ignored. Comments can begin at the beginning or in the middle of a line of code.

Chapter 2. Data type

In Python, data takes the form of *objects* (*abstract data types*), and python programs manipulate these data objects.

A **data type** or simply **type** is a classification identifying one of the various types of objects.

In Python, there are two kinds of objects:

- **Scalar objects**, which cannot be subdivided – example: the number 42 is a scalar object. A numeric data type is only one number.
- **Non-scalar objects**, which can be subdivided (examples: a list of numbers [1,2,3,4,5], the string *pop* are made up respectively of numbers and characters). Non scalar objects could be sequence types or mapping types.

In this course we will study these objects:

- Numeric: Integers, Float
- Sequence: Strings, Lists, Tuples
- Mapping: Dictionary

Here a table reporting different data types.

| Example | Data type |
|------------------------------------------------|--------------------|
| <code>i = 20</code> | <code>int</code> |
| <code>f = 20.5</code> | <code>float</code> |
| <code>s = "Hello"</code> | <code>str</code> |
| <code>L = ["apple", "banana", "cherry"]</code> | <code>list</code> |
| <code>T= ("apple", "banana", "cherry")</code> | <code>tuple</code> |
| <code>d= {"name" : "John", "age" : 36}</code> | <code>dict</code> |

2.1 Identity, type, and value

In Python, every object (or data type) has these three attributes:

- **Values** – The values stored by the object.
- **Type** – The kind of object that is created, e.g., integer, list, string etc. An object's type defines the possible values and operations that type supports. **The `type()` function returns the type of an object**
- **Identity** – The address that the object refers to in the computer's memory. **The `id()` function returns an integer representing its identity.**

Since everything in Python is an object, every variable holds an object instance.

Make a script called `data-types.py`, and in it, define different data types, and print the value and its type.

```
i = 20          #integer type
print(i)
print(type(i))

f = 20.567      #float type
print(f)
print(type(f))

s = 'Hello World' #string type
print(s)
print(type(s))

L = ['apple', 'banana', 'cherry'] #list type
print(L)
print(type(L))

T = ('apple', 'banana', 'cherry') #tuple type
print(T)
print(type(T))

D = {'name' : 'John', 'age' : 36} #dictionary type
print(D)
print(type(D))
```

Notice, you should use the `print` function to display output to screen.

Now run the script, and this is the output:

```
20
<class 'int'>
20.567
<class 'float'>
Hello World
<class 'str'>
['apple', 'banana', 'cherry']
<class 'list'>
```

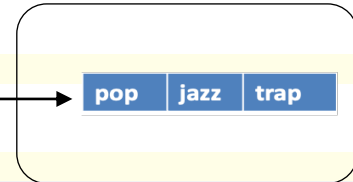
```
('apple', 'banana', 'cherry')
<class 'tuple'>
{'name': 'John', 'age': 36}
<class 'dict'>
```

Here is an example of getting information on the type, identity, and value of an object.

Define this variable:

```
L=['pop','jazz','trap']
```

L



Print its type:

```
print(type(L))
<class 'list'>
```

Print its id:

```
print(id(L))
140179176978496
```

Print the list to display the values:

```
print(L)
['pop', 'jazz', 'trap'] #values are the elements: 'pop' 'jazz' 'trap'
```

2.2 Variables and Object References

What is happening when you make a variable assignment? This is an important question in Python because the answer differs somewhat from what you'd find in many other programming languages. Python is a highly object-oriented language. In fact, virtually every item of data in a Python program is an object of a specific type or class.

Python variables are references to objects, but the actual data is contained in the objects. Since variables are pointing to objects and objects can be of arbitrary data types, in Python the data type of a variable may change during program execution. You can assign an integer value to a variable, use it as an integer for a while and then assign a string to the same variable. This is different than C, C++ or Java where a variable is associated with a fixed data type, e.g., if a variable is of type integer, solely integers can be saved in the variable for the duration of the program.

In the following line of code, we assign the value 42 to a variable:

```
var = 42
```

Equal sign = is an **assignment** of a value to a variable name.

The equal = sign in the assignment shouldn't be seen as *is equal to*. It should be interpreted as *is set to*, meaning in our example *the variable var is set to 42*.

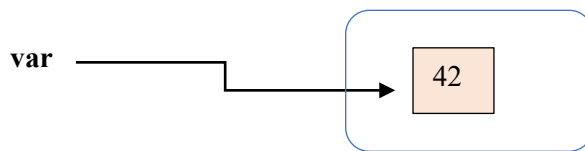
When Python executes an assignment like `var = 42`, it evaluates the right side of the assignment and recognizes that it corresponds to the integer number 42, and it creates an object of the integer class to save this data.

In Python, the type of a variable can change during the execution of a script, or to be precise, a new object, which can be of any type, will be assigned to it.

We illustrate this in the following example.

```
var = 42 #integer object is created and is bound to the name var
print(type(var))
<class 'int'>
```

An assignment binds the name of a variable to the value stored in the computer memory.



You can check the memory address of a variable by using the function `id()`.

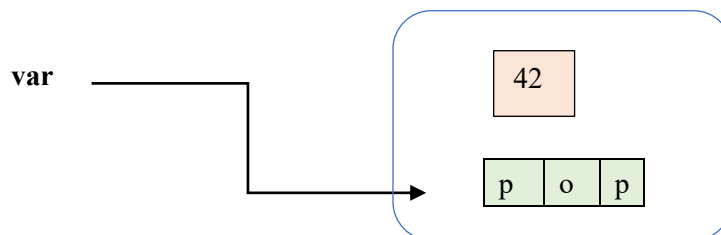
```
id(var)
140384630140496
```

Now we change the value stored in variable `var`

```
var = "pop" #string object is created and is bound to the name var
print(type(var))
<class 'str'>
```

Let's check the memory address of `var`

```
id(var)
140384630144752
```



The value assigned to the variable `var` changed from 42 to “pop” and it shows a different memory address after each assignment. This proves that Python uses dynamic binding for its variables, which only act as labels pointing to some memory address holding some value that we assign to a variable. **The conclusion is that Python uses dynamic binding for variables.**

We can indeed re-bind a variable name using a new variable assignment. The previous value, in this case 42, might still be stored in memory, but it has lost the handle. The Python garbage collector will take care of lost values.

We want to take a closer look at variables now.

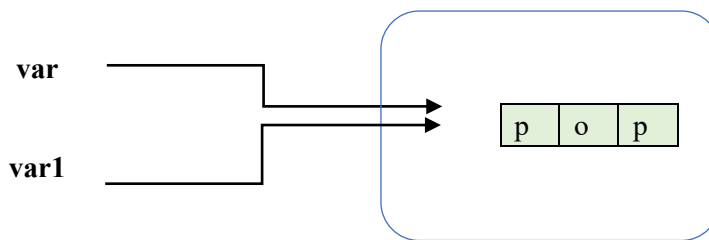
Python variables are references to objects, but the actual data is contained in the objects.

We created a string object *pop* and assigned it to the variable *var*.

After this we assign *var* to the variable *var1*:

```
var1 = var
print(var1)
```

This means that both variables reference the same object. The following picture illustrates this. In this case Python will not make a copy of the object (value) but will make a new reference to the same value. This is important to know when you deal with a data type that is mutable, like a list type. We will talk about mutable and immutable types in the next chapter.



To confirm this, you can check the memory address of both variables by using the function *id()*

```
var="pop"
id(var)
140330528820848
```

```
var1=var
id(var)
140330528820848
```

You see that *var* and *var1* point to the same address.

Now, let's go over another example, and let's calculate the area of a circle:

```
pi=3.14
radius=2
area=pi*(radius**2)
print(area)
12.56
```

Now let's change the value of the radius

```
radius= radius+1
print(radius)
3
```

What happens to the area variable? Did it change value too?

```
print(area)
12.56
```

The value for area does not change until you do the calculation again

```
area=pi*(radius**2)
print(area)
28.26
```

Multiple Assignment

Python allows you to assign a single value to several variables simultaneously. For example

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example

```
a,b,c = 1, "john", [1,2,3,4]
```

Here, one integer object with the value 1 is assigned to variable a, one string object with the value "john" is assigned to variable b, and a list object with the value [1,2,3,4] is assigned to variable c.

2.3 Numeric data type

A **scalar object** cannot be subdivided. Scalar objects are the following:

Numeric:

int – represents integer, ex 5

float – represents real number, ex 3.45

Bool – represents Boolean values True and False – we will use these in conditional statements and while loops.

NoneType – special - has only one value *None* – it represents the absence of a type.

Numeric data types store numeric values. Python has two main built-in **numeric** types that implement the integer and floating-point data types. They are called *int* and *float*. Additionally, there is the complex data type which stores complex numbers.

```
ival=3
print(ival)
3
```

```
print(type(ival))
<class 'int'>
```

```
fval=3.14
print(fval)
3.14
```

```
print(type(fval))
<class 'float'>
```

2.3.1 Numbers, number syntax and scientific notation

Make a variable, m1, to be an integer of 1 million (1,000,000). Try the following:

```
m1=1,000,000
m2=1000000
```

Try printing the variables m1 and m2. Which one worked? Print the data type of each variable.

Scientific notation is **a way of writing very large or very small numbers**. A number is written in scientific notation when a number between 1 and 10 is multiplied by a power of 10. For example, 650,000,000 can be written in scientific notation as 6.5×10^8 .

In python you can write, for example 6.5×10^8 , by using `**` or the `pow` function:

```
6.5*10**8
650000000.0
```

```
6.5*pow(10,8)
650000000.0
```

2.3.2 Expression

You can combine objects and operators to form expressions. An expression has a value, which has a type. Syntax for simple expression:

`<object> <operator> <object> → <object> result`

| Object Operator Object | result |
|------------------------------------------------------------------------------------|--------|
| int + int int - int int * int int**int | int |
| int/int | float |
| int + float int - float int * float int/float int**float float**int | float |
| float + float float - float float * float float/float float**float | float |

Expressions with addition +, subtraction -, and multiplication *:

If both operands are int, the result is int

```
i=1
j=9
type(i+j)
type(i-j)
print(type(i*j))
<class 'int'>
```

If either or both operands are float, the result is float

```
i=1.0
j=9
type(i+j)
type(i-j)
print((type(i*j)))
<class 'float'>
```

Expressions with / division result in float

```
i=1
j=3
z=i/j      #result is stored in variable z
print(type(z))
<class 'float'>
```

Let's see other operators:

```
i=4
j=2
i % j      #the remainder when i is divided by j
0
```

```
print(i**j)  #i to the power of j
16
```

Operators and order of increasing precedence:

Left → Right flow

| | |
|-----|----------------|
| + | addition |
| - | subtraction |
| * | multiplication |
| / | division |
| ** | power |
| () | parentheses |

Typical precedence you might expect in math.

```
print(3*2**3)      #order of precedence results in 24
24
```

The standard arithmetic operations, +, -, *, /, and ** (exponentiation), can be used with parentheses forcing the order of operations away from normal operator precedence.

```
(3*2)**3          #order of precedence results in 216
216
```

Additional operators

/= addition of an = sign to the operator resets the variable to the result. We can do this for any operator

```
x=23
x/=2          #variable reset. x is reassigned to x/2
x
11.5
```

```
25/7          # division
3.5714285714285716
```

```
25//7         #floor divide, rounds down
3
```

2.3.3 math functions

Here some built-in math functions. Example

```
abs(-4)        #absolute value
4
```

```
pow(2,3)       # power
8
```

2.3.4 The math module

Mathematical calculations are an essential part of most Python development. Whether you're working on a scientific project, a financial application, or any other type of programming endeavor, you just can't escape the need for math. For straightforward mathematical calculations in Python, you can use the built-in mathematical **operators**, such as addition (+), subtraction (-), division (/), and multiplication (*). But more advanced operations, such as exponential, logarithmic, trigonometric, or power functions, are not built in. Does that mean you need to implement all these functions from scratch? Fortunately, no. Python provides a [module](#) specifically designed for higher-level mathematical operations: the **math** module. The math module is reported in Chapter 7.1.

2.5 Type conversion functions: int() and float()

Sometimes it is necessary to convert values from one type to another. For example, you cannot sum a string with an integer:

```
s='1210'
i=1340
print(i+s)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In the example, we added an integer value to a string value. Notice the `TypeError` message reported for *unsupported operand type(s)* because we wanted to add integer data type to a string data type. We will use the `int()` function to convert a string type to a numeric type, and then add the values.

```
s1=int(s)
print(i+s1)
2550
```

```
print(i+int(s))
2550
```

If data types are not compatible, Python will throw a `TypeError` message.

Here some examples on `int()` and `float()` functions

```
int('2014') #create integer object 2014 from string '2014'
2014
```

```
int(3.141592) #create integer object 3 from float 3.141592
3
```

```
float('1.99') #create float object 1.99 from string '1.99'
1.99
```

```
float(5) #create float object 5.0 from integer 5
5.0
```

2.6 Basic input and output: input and print functions

In Python, input and output operations (IO Operations) are performed by using two built-in functions. The following are the two built-in functions to perform output operations and input operations.

- `print()` - used for output operation.
- `input()` - used for input operations.

2.6.1 print function

The print function prints arguments arg1, arg2, .. to standard output.

```
print(arg1,arg2,...,argN)
```

Each argument is a data type, including the result of operations.

```
D={1:10,2:20}
L=[1,3,6]
s="Hello"
print(D,L,s)
{1: 10, 2: 20} [1, 3, 6] Hello #the print function makes one space in output
between arguments
```

Arithmetic operations can be performed within the print function

```
x=2
y=5
print(x, y, x/y, x**y)
2 5 0.4 32
```

```
print("Hello", x+y, "times")
Hello 7 times
```

Tab and new line characters

You can include tab and newline string characters in print function

```
"\n"    #newline character is a string type
"\t"    #tab character is a string type
```

```
print("Maria","\t", "Bob")
Maria      Bob
```

```
print("Maria","\n", "Bob")
Maria
Bob
```

2.6.2 print function for formatting output with % – OLD style of Python printing

The modulus is an operator for strings. It is taken from the % format style as used in Bash. This is often referred to as the *printf* method of printing and is used in many languages. Since we used this in bash, we will show how to use it in Python. There is a new way of formatting string in Python, using the string method “format”, which we will include as an optional topic.

Formatting operator % To specify formats follow same syntax as printf() in awk and bash – the only difference is how you list arguments


```
print("format" %arg)
print('format1 format2' %(arg1,arg2))

 %[width].[precision]type

%type
%s  string
%d  integer
%e  scientific notation
%f  floating point real number
```

Examples:

```
pi = 3.141592653589793
print("pi is %.2f " % pi)
pi is 3.14
```

The distance of the Earth from the Sun is 149,597,870 kilometers. We can store this number in a variable.

```
d=149597870
print("distance Earth-Sun is %.4e km" %d)
distance Earth-Sun is 1.4960e+08 km
```

If you have a large number or a small number (i.e., 0.00000001), you might want to format it in scientific notation (i.e., by using the %e)

```
print("pi is %1.2f and d is %.1e " %(pi,d))
pi is 3.14 and d is 1.5e+08
```

1.5e+08 means 1.5×10^8

2.6.3 Input function

The input function can be used for making an interactive script with a user. Input function reads a line from input entered by a user, converts it to a string, and returns that string, which you can store in a variable.

In a script called my-input.py write the following:

```
x1=input("input data here: ")
print(x1)
print(type(x1))
```

Run the script

```
input data here:
```

Now you should enter data, like music, and press return

```
input data here: music
music
<class 'str'>
```

input data here, is the string we wrote within the print function, which appears in the screen when you run the script.

music is the user's input, which will be stored in x1. The type of x1 is string.

Now edit the script to perform arithmetic calculations, run the script and enter a number like 3.14

```
x1=input("input data here: ")
print(x1/10)
print(type(x1))
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

If you want the user to input a number because you want to perform, e.g. arithmetic calculations, you have to perform type conversion from a string to a numeric type by using functions int() or float():

```
x1=float(input("input one number: ")) #you can do this, if the user inputs
only one number
print(x1/10)
print(type(x1))
input one number: 3.14
0.0314
<class 'float'>
```

Now edit the script, and ask the user to input two numbers separated by a comma, because you want to add them and print result to screen – start from this

```
num=input("input two numbers: ")
print(num)
print(type(num))
```

Run the script and input two numbers separated by a comma:

```
input data here: 12,455
12,455
<class 'str'>
```

The two numbers are stored as a string type and the string is *12,455*.

You cannot use float() and int() directly on *12,455* because the string does not contain only one number.

Now we modify the script and use a string type conversion method and type conversion function to sum the two numbers:

```
num=input("input two numbers: ")
print(num)
print(type(num))

L=num.split(',') #convert a string to a list
print(L)
print(float(L[0])+float(L[1]))
```

```
input two numbers: 12,455
12,455
<class 'str'>
['12', '455']
467.0
```

2.7 Sequence data types: String, List, Tuple

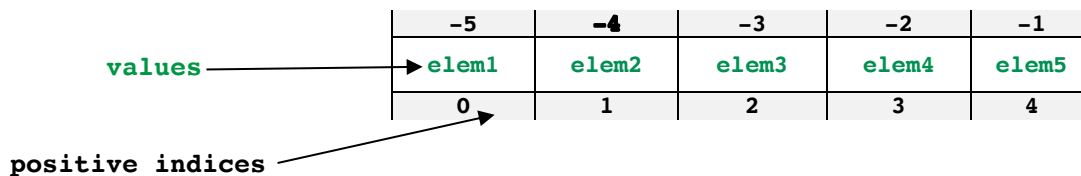
Sequence data types are non-scalar objects, which means they can be subdivided (examples: a list of numbers [1,2,3,4,5] and the string *pop* are made up respectively of numbers and characters).

Python Sequence Data Types come in several forms including Strings, Lists and Tuples. If we think of regular variables like boxes or containers that hold some information, then imagine if we placed a line or queue of people inside one of these boxes. Each person in the line takes up a unique position in the line and we can refer to the line of people by the name we've given to the box, or we can refer to an individual person by the name of the box and the position that they are standing (1st, 2nd, 3rd etc.). Sequence data types are very much like this.

Each element within the sequence data type is referenced by a number, known as the Index, which starts at 0 for the 1st element, 1 for the 2nd element, and so on.

Elements are also referenced with negative integers, and the last element is referenced by -1.

A sequence data type is an ordered collection of items (elements), and each element is referenced by an index



```
s = 'Hello World'      # string elements are Unicode characters
L = ['pop',46,[1,4,5]] # list elements can be of any data type
T = ('pop',46,[1,4,5]) # tuple elements can be of any data type
```

2.7.1 Access one element of a sequence – single elements Indexing

We can use an index to access each element of a sequence type, such as a string, list, and tuple. We can use positive indexes, which start from 0, or negative indexes, which start from -1

To access one element, we need to use an index within square brackets:

```
seq[index] # access one element referenced by index
```

```
s='Hello World' #string type
<class 'str'>
```

```
s[0]
H
```

```
s[4]
o
```

| | | | | | | | | | | |
|-----|-----|----|----|----|----|----|----|----|----|----|
| -11 | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |
| H | e | l | l | o | | W | o | r | l | d |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

```
s[-1]
d
```

```
s[-3]
r
```

```
s[14] #this is an error, since the index does not reference any character
IndexError: string index out of range
```

```
L = ['pop',46,78.5, [1,4,5]]
type(L)
<class 'list'>
```

| -4 | -3 | -2 | -1 |
|-----|----|------|---------|
| pop | 46 | 78.5 | [1,4,5] |
| 0 | 1 | 2 | 3 |

```
L[1] #access second element
46
```

```
type(L[1]) #the type of the second element
<class 'int'>
```

```
L[1]*10 #since the type is numeric I can multiply
4600
```

```
L[-1] #access last element
[1,4,5]
```

```
L[0] #access first element
'pop'
```

```
type(L[0])
<class 'str'>
```

```
T = ('pop',46,78.5, [1,4,5])
<class 'tuple'>
```

```
T[0] #access first element
'pop'
```

```
type(T[0])
<class 'str'>
```

2.7.5 Access nested sequence objects

Since the first element of L is a string type, you can use another [] to access an element of that string

```
L[0][0]
'p'
```

```
L[-1][0]
1
```

Since the last element of T is a list type, you can use another [] to access an element of that list

```
T[-1]
[1,4,5])
```

```
T[-1][1]
4
```

2.7.2 Subsequences can be created with the slice or slicing notation

```
seq[start:end]      # from index start through index end-1
seq[start:]         # from index start through last index
seq[:end]           # from index 0 through index end-1
seq[:]              # a copy of the whole sequence
seq[::-1]           # reverse the sequence
seq[start:end:step] # from index start through not past end, by step
seq[-2:]            # last two elements in the sequence
seq[:-2]            # everything except the last two elements
```

```
s='Hello World'
<class 'str'>
```

```
s[1:4]    #from index 1 to index 3
'ell'
```

| | | | | | | | | | | |
|-----|-----|----|----|----|----|----|----|----|----|----|
| -11 | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |
| H | e | l | l | o | | W | o | r | l | d |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

```
s[:4]     #from index 0 to index 3
'Hello'
```

```
s[6:]     #from index 6 to the end
World
```

```
s1=s[:]   #copy of the string
print(s1)
Hello World
```

```
s2=s[::-1] # reverse the sequence
print(s2)
'dlroW olleH'
```

```
L = ['pop', 46, 78.5, [1, 4, 5]]
type(L)
<class 'list'>
```

| -4 | -3 | -2 | -1 |
|-----|----|------|-----------|
| pop | 46 | 78.5 | [1, 4, 5] |
| 0 | 1 | 2 | 3 |

```
L[1:] #element at index 1 through last index
[46, 78.5, [1, 4, 5]]
```

```
L[:3] #first element through element at index-1
['pop', 46, 78.5]
```

```
L[::-1] #reverse the list
[[1, 4, 5], 78.5, 46, 'pop']
```

```
L1=L[:] #copy of the sequence. A new object is created and referenced by L1
id(L)
140330551829568
```

```
id(L1)
140330530225536
```

```
print(L1)
['pop', 46, 78.5, [1, 4, 5]]
```

2.7.3 Creating a string object.

The task of the first-generation computers in the forties and fifties had been - due to technical restraints - focused on number processing. Text processing had been just a dream at that time. Nowadays, one of the main tasks of computers is text processing in all its varieties; the most prominent applications are search engines like Google. To enable text processing, programming languages need suitable data types. Strings are used in all modern programming languages to store and process textual information. Logically, a string - like any text - is a sequence of characters. The question remains what a character consists of. In computer science or computer technology, a character is a unit of information, based on Unicode standard. Unicode is a universal character encoding standard that assigns a code to every character and symbol in every language in the world. Since no other encoding standard supports all languages, Unicode is the only encoding standard that ensures that you can retrieve or combine data using any combination of languages. A string type is an ordered sequence of Unicode characters (letters, symbols, numbers, punctuation). The elements of a string types are characters.

To create a string, you enclose characters between single quotes or double quotes.

```
s1='Hello World' #string type
print(type(s1))
<class 'str'>
```

```
s1="Hello World"
print(type(s1))
<class 'str'>
```

```
s1=''          #the null character
print(type(s1))
<class 'str'>
```

```
s1='678656' #numbers enclosed in quotes are string objects
print(type(s1))
<class 'str'>
```

```
s1='a'         #single character is simply a string with one element
print(type(s1))
<class 'str'>
```

Escape is used to remove the special meaning of single and double quotes

If you wish to use a double quote in your string and you are defining the string with double quotes, then you must “escape” the double quote. By using the backslash, Python does not get confused over which quote is being used to define a string and which quote is part of the string. Similarly, for single quotes.

Try these

```
print("Hello Python")
Hello Python
```

```
print("\"Hello Python\"")
"Hello Python"
```

```
#within double quotes, single quote losses meaning
print(" ' Hello Python ' ")
' Hello Python '
```

Triple quotes You want to use triple quotes if you want to define a string in multiple lines.

In addition, text between triple quotes can be used to write multiple lines of comments in your script.

In a python script called test-triples.py

```
#You can place COMMENTS after a pound (#) sign

"""
Or you can place COMMENTS inside triple quotes if you have more than one line
of text.
"""

greetings="""Hello world
I wanted to ask you if you are happy"""

print(greeting)
print(type(greeting))
Hello world
I wanted to ask you if you are happy
<class 'str'>
```

Type conversion to string

You can create a string by using the type conversion function `str()` and the `join` method.

the `str()` function

The `str()` function is used to convert an int number or float number to string type

```
str(4) #return a string object  
'4'
```

```
str(3.141592) #return a string object  
'3.141592'
```

the `join` method

The **`join()` method is a string method** that returns a new string by joining all the elements of an iterable whose elements are strings separated by a comma.

An iterable is an object (data type) capable of returning its members one at a time, such as list, tuple, string, and dictionary.

The `join()` method provides a flexible way to create strings by joining each element of a list of strings, and a tuple of strings by a string separator. The `join` method returns a new string object, which is the concatenation of the string elements.

The syntax of the `join()` method is:

```
sep.join(list of strings)  
sep.join(tuple of strings)
```

If the iterable contains any non-string values, it raises a `TypeError`. You can use the `join` methods with a homogeneous list or tuple of strings, i.e., all the items of a list or tuple must be string type.

List of strings to string

```
L=['G', 'G', 'C', 'C', 'T', 'T', 'C'] # list of strings  
sep=''   
s=sep.join(L)  
print(s)  
'GGCCTTC'
```

A different joining string can be used:

```
sep='+'  
s1=sep.join(L)  
print(s1)  
'G+G+C+C+T+T+C'
```



```
L=['1','2','3'] # list of strings
sep=' '
s2=sep.join(L)
print(s2)
'1 2 3'
```

```
L=[1,2,3] # list of numeric type elements
sep=' '
s3=sep.join(L)
TypeError: sequence item 0: expected str instance, int found
```

Tuple of strings to string

```
T=('1','2','3')
sep='*'
s1=sep.join(T)
print(s1)
'1*2*3'
```

```
T=(1,2,3, 'T','C') #non-homogeneous tuple: items are of different types
sep=' '
s2=sep.join(T)
TypeError: sequence item 0: expected str instance, int found
```

Dictionary keys of strings to string

join() will join each key of a dictionary by a string separator. It only works if all keys are string data type

```
d={'a':'alpha','b':'beta'}
sep='-'
s1=sep.join(d)
print(s1)
'a-b'
```

```
s1='ab'
s2='123'
s3=s1.join(s2)
print(s3)
'1ab2ab3'
```

2.7.4 Creating a list object.

A list type is an ordered sequence of any data type. The elements of a list type can be of any type. To create a list data type, enclose the items within brackets and separate the elements with commas.

```
list_name = [elem1, elem2, elem3, elem4]
```

Lists can contain different sorts of elements: numbers, strings, dictionaries, and even other lists.

```
list1 = ['School', 'year', 2015, 2016]
type(list1)
<class 'list'>
```

```
list2=[1,3,5,7,9]      #homogeneous list, items are all numeric
list3=['a', 'b', 'c']  #homogeneous list, items are all string type
list4=['pop',46,[1,4,5]] #non-homogeneous list, items are of different types
list5=[]               #empty list
```

Type conversion to list

You can use the list() function and the split method to create a list from other objects.

The list() function

The list function is used to convert different types (like a string, or tuple) to a list type.

```
list((1,2,3,4))  #tuple to list
[1, 2, 3, 4]
```

```
list('G+T+C')    #string to list
['G', '+', 'T', '+', 'C']
```

```
D={1: 'a', 2: 'b', 3: 'c'}
list(D.keys()) #list of keys
[1,2,3]
```

```
D={1: 'a', 2: 'b', 3: 'c'}
list(D.values()) #list of values
['a','b','c']
```

```
D={1: 'a', 2: 'b', 3: 'c'}
list(D.items()) #list of key,value tuple
[(1, 'a'), (2, 'b'), (3, 'c')]
```

The split method

The **split() string method** breaks up a string at the specified separator and returns a list of strings. The simplified syntax of split() is:

```
str.split(sep)  sep is a string delimiter. The string splits at the specified separator.
```

```
str.split() - if the separator is not specified, any whitespace (space, newline etc.) string is a separator.
```

```
mystr='G G C C T T C T C G A A T G A A T C'
L=mystr.split()
print(L)
['G', 'G', 'C', 'C', 'T', 'T', 'C', 'T', 'C', 'G', 'A', 'A', 'T', 'G', 'A', 'A', 'T', 'C']
```

```
mystr2='G+G+C+C+T+T+C+T+C+G+A+A+T+G+A+A+T+C'
L=mystr2.split('+')
print(L)
['G', 'G', 'C', 'C', 'T', 'T', 'C', 'T', 'C', 'G', 'A', 'A', 'T', 'G', 'A', 'A', 'T', 'C']
```

If the separator is the null character and you want to create a list where each element is one character of a string, you should use the list function instead.

```
mystr='GGCCTTC'
L=mystr.split()
print(L)
['GGCCTTC']
```

```
L=list(mystr)
['G', 'G', 'C', 'C', 'T', 'T', 'C']
```

2.7.5 Creating a tuple object

A **tuple type** is like a list type, but it is defined with parentheses, (), not brackets. The elements can be of any data type.

You can define a tuple with an expression of the forms:

```
tuple_name= (elem1, elem2, elem3, elem4)
tuple_name= elem1, elem2, elem3, elem4
```

with the elements (or items) between parentheses, separated by a comma.

Or you can define a tuple by separating the items by commas, without using parentheses.

Tuples can contain different sorts of items: numbers, strings, and even other tuples.

```
tuple1 = ('physics', 'chemistry', 1997, 2000)
print(type(tuple1))
<type 'tuple'>
```

a tuple can be also created without using parentheses

```
tuple1 = 'physics', 'chemistry', 1997, 2000
print(type(tuple1))
<type 'tuple'>
```

```
print(tuple1)
('physics', 'chemistry', 1997, 2000)
```

```
tuple2= (1,3,5,7,9)      #homogeneous tuple, items are all numeric
tuple3=('a', 'b', 'c')   #homogeneous tuple, items are all string type
tuple4=('po',46,[1,4])   #non-homogeneous tuple, items are of different types
tuple5=()                #empty tuple
```

Tuple packing and unpacking.

When we create a tuple, we normally assign values to it. This is called "packing" a tuple:

```
T=10,30,20,40
print(T)
(10, 30, 20, 40)
```

But, in Python, we are also allowed to extract the individual values into variables. This is called "unpacking":

```
a,c,b,d=T
print(a)
10
```

This feature allows multiple variable assignments to be performed in this way

```
a,b="Hello",[1,3,5]
print(a)
print(b)
Hello
[1, 3, 5]
```

type conversion to tuple

You can use the tuple() function to create a tuple from certain objects

```
tuple('Mary') #string to tuple
('M', 'a', 'r', 'y')
```

```
tuple([1,2,3,4]) #list to tuple
(1, 2, 3, 4)
```

2.8 Mapping data type: Dictionary

Another useful data type built into Python is the *dictionary* (see [Mapping Types — dict](#)). Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. It is best to think of a dictionary as an ordered set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type. Strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys since lists are mutable data types. Values can be all sort of data types. Dictionaries do not support indexing, slicing, or other sequence-like behavior.

2.8.1 Create a dictionary

A dictionary is a lookup table. It consists of keys and values or key-value pairs. They can be quite handy and have many purposes.

Dictionaries can be created by placing a comma-separated list of key:value pairs within curly braces:

```
d = {key1 : value1, key2 : value2, key3 : value3}
```

Keys can be **only** immutable data types, such as strings or numbers

Values can be of any data type

An empty pair of curly braces {} is an empty dictionary, just like an empty pair of [] is an empty list.

```
empty = {}
```

There are multiple ways to create a dictionary. Below are several ways of creating the exact same dictionary:

```
a = dict(one=1, two=2, three=3)
b = {'one': 1, 'two': 2, 'three': 3}
```

You can use the dict() function for type conversion to dictionary

```
d = dict([('two', 2), ('one', 1), ('three', 3)]) #from list of 2-elements
tuple
```

dict() in combination with the zip function

```
list1=['one', 'two', 'three']
list2=[1,2,3]

d=dict(zip(list1,list2)) #converts two lists to a dictionary type, where items
of list1 are the keys, and items of list2 are the corresponding values
```

2.8.2 Accessing values in a dictionary.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is an error to extract a value using a non-existent key.

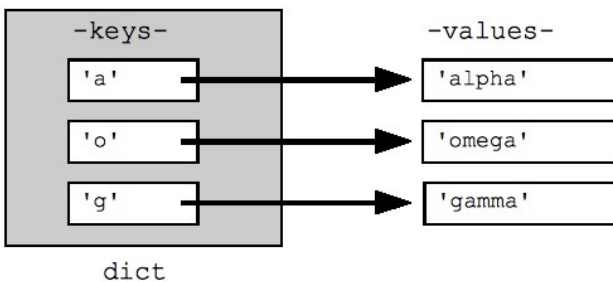
Individual values in a dictionary can be accessed by looking up a key:

```
d['key1']
```

For example, for dictionary d:

```
d={'a':'alpha','b':'beta','g':'gamma'}
d['a'] #access value 'alpha'
'alpha'
```

You can access one value at the time



Access nested data types

```
d['a'][2]
'p'
```

Summary - Type Conversion Functions and methods

NOTICE: all type conversions return a new object, which you can store in a variable.

| Function(s)/Methods | Converting what to what | Example |
|---------------------|---------------------------|---------------------|
| int() | string to integer | int('2014') 2014 |
| int() | floating point to integer | int(3.141592) 3 |

| | | |
|---------------------------|-----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| float() | string to float | <pre>float('1.99') 1.99</pre> |
| float() | integer to float | <pre>float(5) 5.0</pre> |
| | | |
| str() | integer to string | <pre>str(4) '4'</pre> |
| str() | float to string | <pre>str(3.141592) '3.141592'</pre> |
| sep.join(iterable) | list of strings to string | <pre>L=['G', 'A', 'T', 'A'] sep='' s=sep.join(L) print(s) 'GATA'</pre> <pre>L=['1', '2', '3'] sep='*' s2=sep.join(L) print(s2) '1*2*3'</pre> |
| | | |
| tuple() | string to tuple | <pre>tuple('Mary') ('M', 'a', 'r', 'y')</pre> |
| tuple() | list to tuple | <pre>tuple([1,2,3,4]) (1, 2, 3, 4)</pre> |
| | | |
| list() | tuple to list | <pre>list((1,2,3,4)) [1, 2, 3, 4]</pre> |
| list() | string to list | <pre>list('G+T+C') ['G', '+', 'T', '+', 'C']</pre> |
| string.split(sep) | string to list | <pre>s='G+T+C' L=s.split('+') print(L) ['G', 'T', 'C']</pre> |
| list(dict.keys()) | dictionary to list of keys | <pre>D={1: 'a', 2: 'b', 3} list(D.keys()) [1,2,3]</pre> |

| | | |
|-------------------------------|----------------------------------------------|--------------------------------------------------------------------------------------|
| list(dict.values()) | dictionary to list of values | <pre>D={1: 'a', 2: 'b', 3: 'c'} list(D.values()) ['a', 'b', 'c']</pre> |
| list(dict.items()) | dictionary to list of key-value tuple | <pre>D={1: 'a', 2: 'b', 3: 'c'} list(D.items()) [(1, 'a'), (2, 'b'), (3, 'c')]</pre> |
| dict(zip(list1,list2)) | Two lists to dictionary | <pre>l1=[1,2,3] l2=['a','b','c'] d1=dict(zip(l1,l2)) {1: 'a', 2: 'b', 3: 'c'}</pre> |

Chapter 3. Operations, Mutability, Methods

3.1 Concatenation and Repetition of sequences

Sequence types support operations of concatenation and repetition.

`<object> <operator> <object>` → expression evaluates to a value, which is a data type

Operators: + and *

`seq1 + seq2` concatenation of two sequence types

`seq1 * n` repetition of a sequence n times, where n is an integer

```
s1="music"
s2="jazz"
s1+s2    #string concatenation results in a string type
'musicjazz'
```

```
s1+" "+s2
'music jazz'
```

```
s3=s1+" "+s2  #store result of concatenation in variable
type(s3)
<class 'str'>
```

```
s1*3        #string repetition results in a string type
'musicmusicmusic'
```

```
L1=[1,2,3,4]
L2=[5,6,7,8]
L1+L2       #list concatenation results in a list type
[1, 2, 3, 4, 5, 6, 7, 8]
```

```
L3=['a','pop','b']
L1+L2+L3
[1, 2, 3, 4, 5, 6, 7, 8, 'a', 'pop', 'b']
L4=L1+L2+L3
type(L4)
<class 'list'>
```

```
L1*3        #list repetition results in a list type
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
```

```
s1+L        #cannot concatenate two objects of different types
TypeError: can only concatenate str (not "list") to str
```

3.2 Common Functions and Methods of sequence types

There are some functions and methods (methods are functions applied by using a dot) that work on a sequence type: string, list, and tuple.

```
seq.count(sub)  returns a number, the count of sub in the sequence
seq.index(sub)  returns the index of the first occurrence of sub
len(seq)        returns the length (number of elements) of a sequence
min(seq)        returns smallest item of seq. works for homogeneous items
max(seq)        returns largest item of seq. works for homogeneous items
```

```
s1="this is a string"
s1.count("i")
3
```

```
s1.count("is")
2
```

```
s1.index("g")
15
```

```
len(s1) #returns the number of characters in the string
16
```

```
L=["cat", 'cat', 'dog', 'elephant', 1, 10, 10, 10]
L.count("cat")
2
```

```
L.count(10)
3
```

```
L.index("cat")
0
```

```
len(L) #returns the number of elements in a list
8
```

```
L=[1, 100, 9, 8.9, 0.001]
max(L)
100
```

```
min(L)
0.001
```

```
s='abghtyu'
max(s)      #alphabetically
'y'
```

Summary – Sequence type operations

| Operation | Result |
|---------------------------|---------------------------------------------------------------------------------------------|
| <code>seq + seq1</code> | the concatenation of <code>seq</code> and <code>seq1</code> |
| <code>seq*n</code> | Repetition of <code>seq</code> <code>n</code> times |
| <code>seq[i]</code> | <code>i</code> th item of <code>seq</code> , origin 0, index <code>i</code> th-1 |
| <code>seq[i:j]</code> | slice of <code>seq</code> from <code>i</code> to <code>j</code> -1 |
| <code>seq[i:j:k]</code> | slice of <code>seq</code> from <code>i</code> to <code>j</code> -1 with step <code>k</code> |
| <code>len(seq)</code> | length of <code>seq</code> |
| <code>seq.count(x)</code> | total number of occurrences of <code>x</code> in <code>seq</code> |
| <code>seq.index(x)</code> | return index of <code>x</code> in <code>seq</code> |
| <code>min(seq)</code> | smallest item of <code>seq</code> – works for homogeneous items |
| <code>max(seq)</code> | largest item of <code>seq</code> – works for homogeneous items |

Common sequence operation

<https://docs.python.org/3/library/stdtypes.html#common-sequence-operations>

3.3 Mutability and Methods

All the data in a Python code is represented by objects or by relations between objects. In Python, every object has these three attributes:

- **Identity** – The address that the object refers to in the computer's memory. The `id()` function returns an integer representing its identity.
- **Type** – The kind of object that is created, e.g., integer, list, string etc. An object's type defines the possible values and operations that type supports. The `type()` function returns the type of an object
- **Values** – The values stored by the object.

Objects whose values can change are said to be **mutable**.

Objects whose values are unchangeable once they are created are called **immutable**.

The ID and Type cannot be changed once an object is created, but the values can be changed for Mutable objects.

Mutable Objects of built-in type that are:

- Lists
- Dictionaries

Immutable Objects of built-in type are:

- Numbers
- Strings
- Tuples

Object mutability is one of the characteristics that makes Python a dynamically typed language.

Each data type supports a set of operations and methods that can be done on values of that specific type.

Methods are functions that are associated with a specific data type. There are string methods, list methods, and dictionary methods. Furthermore, there are functions and methods for type conversion.

3.3.1 List: Mutability and Methods

Lists are mutable types. We can change their values after we create them.

We can change values by using item assignment or by applying list methods.

Item assignment

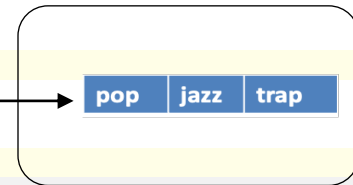
To perform item assignment, use the index that references the item (value) you want to change

```
object[index]=new_value #item assignment
```

Define this list:

```
L=['pop','jazz','trap']
```

L



Print its id

```
print(id(L))
140179176978496
```

Print the list to display the values

```
print(L)
['pop', 'jazz', 'trap']
```

We want to change the value 'pop' with 'blues', and we perform item assignment:

```
L[0]='blues'
print(id(L))
140179176978496
```

L



```
print(L)
['blues', 'jazz', 'trap']
```

Notice the variable L points to the same memory address but the value at index 0 changed.

List methods

There is a set of *methods* specific for lists.

| | |
|----------------------------------|---------------------------------------------------|
| <code>list.append(item)</code> | Add item at the end of the list |
| <code>list.clear()</code> | Remove all the items from the list |
| <code>list.copy()</code> | Returns a copy of the list |
| <code>list.extend(list1)</code> | Extend the list by concatenating list1 |
| <code>list.insert(i,item)</code> | Insert item at the index i |
| <code>list.pop(i)</code> | Remove the item at index i, return that item. |
| <code>list.remove(item)</code> | Remove the first occurrence of item from the list |
| <code>list.reverse()</code> | Reverse the order of the list |
| <code>list.sort()</code> | Sort the list |

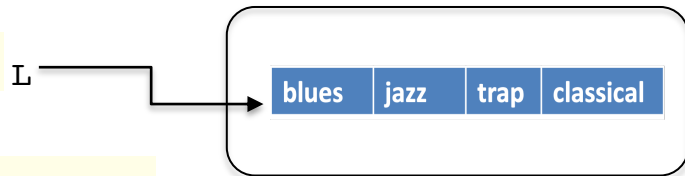
To learn more about available list methods: <https://docs.python.org/3/tutorial/datastructures.html>

Adding items

Let's try the append method, and let's append a string object

```
L.append('classical')
print(id(L))
140179176978496
```

```
print(L)
['blues', 'jazz', 'trap', 'classical']
```



Notice L points to the same object, with modified values. Values changed, but the id is the same.

Let's explore other list methods. All these methods will modify the values of list L

```
L.insert(2,"soul")    #L got modified again by the insert method
print(L)
['blues', 'jazz', 'soul', 'trap', 'classic']
```

```
L.extend(['rock', 'punk']) #same as concatenating lists
print(L)
['blues', 'jazz', 'soul', 'trap', 'classic', 'rock', 'punk']
```

```
L.append([1,2])
print(L)
['blues', 'jazz', 'soul', 'trap', 'classic', 'rock', 'punk', [1,2]]
```

Notice the difference between append and extend.

Deleting items

Let's use the remove method:

```
L.remove('jazz')
print(L)
['blues', 'soul', 'trap', 'classic', 'rock', 'punk', [1,2]]
```

and now use the pop method:

```
x=L.pop(0) #the removed item at index 0 is returned
print(x)
print(L)
'blues'
['soul', 'trap', 'classic', 'rock', 'punk', [1,2]]
```

Notice the difference between remove and pop.

There is a way to remove an item from a list given its index instead of its value: the **del** statement. This differs from the pop() method which returns a value. The del statement can also be used to remove slices from a list or clear the entire list.

```
Del list[index]
del list[start:end]
del list[start:end:step]
```

```
del L[0] #delete one element by using indexing
print(L)
['trap', 'classic', 'blues', 'd', 'e', 'f', [1, 2, 3]]
```

```
del L[3:] #delete a range of elements by using slicing
print(L)
['trap', 'classic', 'blues']
```

Methods like append, insert, remove, or sort are list methods that modify the values of a list object and have no returned value– they return *None* type (absence of a type).

If you make a new variable like this `L1=L.append(item)`, L1 will store the None type. Be careful, and do not to create variables that store the None type. However, if a method returns a value, like the pop method, then you can store that returned value into a variable.

Make a script called **list1.py** and in it

```
L=['a','b','c']
L.append(100)
print(L)

L.remove('a')
print(L)

i=L.pop(0)
print(i)
print(L)
['a', 'b', 'c', 100]
['b', 'c', 100]
b
['c', 100]
```

3.3.2 Dictionary: Mutability and Methods

Dictionaries are mapping and mutable types.

Define this dictionary:

```
D={'one': 1, 'two': 2, 'three': 3, 'four': 4}
print(id(D))
140296047627328
```

```
print(D)
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

The values are 1,2,3 and 4.

Let's now change a value in a dictionary type associated to a certain key.

Change one value

To change a value, we need to use its key:

```
d[key]=new_value
```

```
D['one']=11
print(D)
{'one': 11, 'two': 2, 'three': 3, 'four': 4}
```

The value 1, which was associated to key 'one' changed to 11.

```
print(id(D))
140296047627328
```

The id is the same, and the same variable points to the modified object.

Delete a key-value pair

A key-value pair can be removed from a dictionary with the **del** command:

```
del d[key]
```

```
del D['one']
print(D)
{'two': 2, 'three': 3, 'four': 4}
```

This has removed the key 'one' and its associated value 11 from the dictionary

Let's apply a dictionary method

```
d1={'a':100,'b':200}
D.update(d1)
print(D)
{'two': 2, 'three': 3, 'four': 4, 'a': 100, 'b': 200}
```

Dictionary Methods & Description

| | |
|-----------------------------|----------------------------------------------------------------------------------------------|
| <code>dict.clear()</code> | Removes all the elements from the dictionary |
| <code>dict.copy()</code> | Returns a copy of the dictionary |
| <code>dict.get(key)</code> | Returns the value of the specified key |
| <code>dict.pop(key)</code> | Removes the element with the specified key, and returns that value |
| <code>dict.popitem()</code> | Removes the last inserted key-value pair |
| <code>dict.keys()</code> | Returns a view object. The view object contains the keys of the dictionary, as a list. |
| <code>dict.values()</code> | Returns a view object. The view object contains the values of the dictionary, as a list. |
| <code>dict.items()</code> | Returns a view object. The view object contains tuples (key,value) pairs as items in a list. |

All dictionary methods <https://realpython.com/python-dicts/>

3.3.3 Side effects of mutability

Define this list

```
L = ['a', 'b', 'c']
```

Now assign to L1 the values of L. We know Python does not make a copy of the object, but both L and L1 reference the same object.

```
L1 = L
print(id(L))
print(id(L1))
140330551960448
140330551960448
```

What happen if we modify L? Since L and L1 reference the same object, L1 gets modified too

```
L.append(100)
print(L)
print(L1)
['a', 'b', 'c', 100]
['a', 'b', 'c', 100]
```

To avoid this, you could make an actual copy of the list (a new list object with a different id and same values) by using [:]

```
L2=L[:]      #copy of the list - make a new object (new id) with same values
print(id(L))
print(id(L2))
140330551960448
140330551969088
```

Now if we modify L, L2 does not get modified:

```
L.append('d')
print(L)
print(L2)
['a', 'b', 'c', 100, 'd']
['a', 'b', 'c', 100]
```

3.3.4 String: Immutability and string methods

Let's apply item assignment to a **string** type.

```
s="Hello"
print(id())
140179176931760
```

```
print(type(s))
<class 'str'>
```

s



```
print(s)
Hello
```

The values are the characters Hello

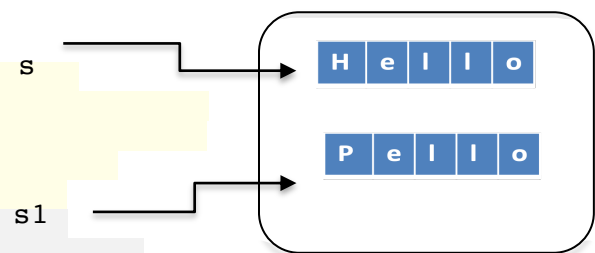
```
s[0]='P'
TypeError: 'str' object does not support item assignment
```

The error means that string type is immutable because values cannot be changed after the string is created.

String methods

Let's try to apply the string method replace:

```
s1=s.replace('H', 'P')
print(s)
print(s1)
print(id(s1))
print(id(s))
Hello
140578620091248
Pello
140578620133424
```



When you modify values of a string object a new object is created. You can store the new object with the modified values into a variable.

There is a set of string *methods* that help with sophisticated text processing tasks, and these methods only work on string objects. String are immutable types, and every time you use methods to modify the values of a string, a new string object is created instead.

Below are some more useful string methods: upper(), strip()

```
str1='Spring break is almost here'
str2=str1.upper()
print(str2)
SPRING BREAK IS ALMOST HERE
```

The methods upper() and lower() will return a new string with all the letters converted to upper- or lower-case letters.

If you apply string methods, a new string object is generated which you could store in a variable. Make a script called string1.py

```
str1='Spring break is almost here'
str2=str1.upper()
print(str2)
print(len(str2))
print(str1[:2])
SPRING BREAK IS ALMOST HERE
27
Srn ra sams ee
```

| | |
|-----------------------------------|-----------------------------------------------------------------|
| <code>seq.upper()</code> | returns the string seq uppercase |
| <code>seq.lower()</code> | returns the string seq lowercase |
| <code>seq.replace(old,new)</code> | replace old with new and returns the string seq with new value. |
| <code>seq.split()</code> | type converts a list into a string |
| <code>seq.join(iterable)</code> | type convert an iterable into a string |

All string methods <https://docs.python.org/3/library/stdtypes.html#string-methods>

3.3.5 Tuple: Immutability

List and tuple types seem very similar. They are both sequence types, and so support indexing and slicing, and are both iterable (iterable is an object which can be looped over). The main difference is that a tuple cannot be changed once it's defined. The tuple type is immutable, but the list type is a mutable type.

Let's try to change a value of a tuple type by using item assignment,

```
T=('pop', 'jazz', 'trap')
T[0]='blues'
TypeError: 'tuple' object does not support item assignment
```

This error means that tuple type is immutable because values cannot be changed after the tuple is created.

The tuple is considered immutable because the collection of objects it contains cannot be changed. However, if an element of a tuple is mutable, that element can change.

```
T=(42,[1,2,3], 'hello')
print(T)
print(id(T))
T[1].append('pop')
print(id(T))
print(T)
(42, [1, 2, 3], 'hello')
140578619813056
(42, [1, 2, 3, 'pop'], 'hello')
140578619813056
```

So, immutability is not strictly the same as having an unchangeable value, it is subtler. A tuple type supports the index and count methods!

3.3.6 Number: Immutability

Numbers are a scalar type, so we cannot use the `[]` to access value. If we modify the original value, the modified value is a new object, with a different id, which means that we did not change the value, but we created a new object containing the new value.

```
Number=10
id(number)
4441438848
```

```
number=number+1    # change value
id(number)          #id is different
4441438880
```

When you change the value of variable number, a new object is created.
Every time you change the value of an immutable object, a new object is created.

3.3.7 Get information on methods.

If you want to get information on methods, you can use the `dir` function at the Ipython console.
Make an object of a type

```
o=Object
dir(o)          #will display all the methods of that object
help(o.method) #will enter the doc page of the method
```

Example:

```
s='pop'
dir(s)
help(s.find)
```

You can use the online documentation

string methods

<https://docs.python.org/3/library/stdtypes.html#string-methods>

list methods

<https://docs.python.org/3.1/tutorial/datastructures.html>

dictionary methods

<https://realpython.com/python-dicts/>

Chapter 4. Flow control

A bit about for loops

Repetitive execution of the same block of code over and over is referred to as **iteration**.

There are two types of iteration:

- **Definite** iteration, in which the number of repetitions is specified explicitly in advance.
- **Indefinite** iteration, in which the code block executes until some condition is met.

In Python, indefinite iteration is performed with a while loop.

Definite iteration loops are frequently referred to as **for** loops because **for** is the [keyword](#) that is used to introduce them in nearly all programming languages, including Python.

Historically, programming languages have offered a few assorted flavors of for loops, which we briefly summarize here:

- **Numeric Range Loop** - the most basic for loop is a simple numeric range statement with start and end values.
- **Three-Expression Loop** - another form of for loop popularized by the C programming language contains three parts: an initialization, an expression specifying an ending condition, and an action to be performed at the end of each iteration.
Three-expression for loops are popular because the expressions specified for the three parts can be nearly anything, so this has quite a bit more flexibility than the simpler numeric range loop.
- **Collection-Based or Iterator-Based Loop** - this type of loop iterates over a collection of objects, rather than specifying numeric values or conditions:

```
for i in collection
    loop body
```

Each time through the loop, the variable *i* takes on the value of the next object in *collection*. This type of for loop is arguably the most generalized and abstract.

4.1 Iterables and Python for loop

Python implements the collection-based iteration.

Python's for loop looks like this:

```
for var in iterable:
    statement(s)    #must indent
```

iterable is a collection of objects—for example, a list or tuple.

The statement(s) in the loop body are denoted by indentation, as with all Python control structures, and are executed once for each item in `iterable`. The loop variable `var` takes on the value of the next element in `iterable` each time through the loop.

More specifically, an **iterable** is an object capable of returning its members one by one. Said in other words, an iterable is anything that you can loop over with a for loop in Python. Sequences, like lists, strings, and tuples, are a very common type of iterable. Many things in Python are iterables, but not all of them are sequences. Dictionaries and file objects are iterables as well.

Example – looping over items of a list.

```
L = ['foo', 'bar', 'baz'] #loop over items of a list

for i in L:
    print(i)
foo
bar
baz
```

In this example, `iterable` is the list `L`, and `var` is the variable `i`. Each time through the loop, `i` takes on a successive item in `L`, so `print()` displays the values 'foo', 'bar', and 'baz', respectively.

Other examples:

```
for char in 'abc':
    print(char) # loop over characters of a string
a
b
c
```

```
for item in ('abc', 12, [1,2]):
    print(item) # loop over items of a tuple
abc
12
[1, 2]
```

Just like with strings, lists and tuples, you can loop through the keys of a dictionary

```
d={'a':'alpha','b':'beta'}
for i in d:
    print(i)
a
b
```

```
d={'a':'alpha','b':'beta'}

for i in d:
    print(i,d[i]) # you can access the associated value by using the key
a alpha
b beta
```

4.1.1 Looping through a dictionary by using items() and values() methods

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the **items()** method, which returns an iterable object containing the key-value pairs of the dictionary, as tuples in a list.

```
d={'a':'alpha','b':'beta'}
for key,value in d.items():
    print(key,value)
a alpha
b beta
```

The method **values()** is a convenient way for iterating directly over the values (but you cannot access the key). The values() method returns an iterable object of values.

```
d={'a':'alpha','b':'beta'}
for value in d.values():
    print(value)
alpha
beta
```

4.2 The range function

The range function generates an iterable object of integer numbers of class range:

```
type(range(3))
<class 'range'>
```

The [range](#) function is useful to create a list or tuple of integer numbers, and generally used in iteration context such as for loops.

```
range(n)           #from 0 to n-1 and increment by 1
range(start,n)     #from start to n-1
range(start,n,step) #from start to n-1 with increment specified by step
```

4.2.1 Range functions to create a list or tuple of integer numbers

Use the range function in combination with the list function to generate a list or tuple of integer numbers.

```
list(range(5))
[0, 1, 2, 3, 4]
```

```
list(range(1,4))  
[1, 2, 3]
```

```
tuple(range(1,4,2))  
(1, 3)
```

Try these:

```
list(range(10))
```

```
list(range(3,13))
```

```
tuple(range(0,20,2))
```

```
tuple(range(-12,-1))
```

```
list(range(-2,-20,-2))
```

4.2.2 range function to loop over a sequence of integer numbers

The built-in function **range()** is useful to iterate over a sequence of integer numbers

```
for i in range(5):      #0 to 4 in step of 1  
    print(i)
```

```
for j in range(2,6):  
    print(j)
```

```
for i in range(2,30,3):  
    print(i)
```

```
for i in range(-10,-20,-2): #you can generate negative integers  
    print(i)
```

The range function is useful to repeat a set of code a specified number of times, for example if you want to prompt a user a specific number of times:

```
for i in range(4):  
    some=input("Enter something > ")  
    print("Ah .. you entered", some)
```


4.3 Looping over multiple sequence types

Sometimes we need to loop over multiple sequence types at the same time, i.e., in one for loop. For example, we have these two lists:

```
fruits = ["loquat", "jujube", "pear"]
colors = ["brown", "orange", "green"]
```

and we want to **loop over them at the same time** to get color for each fruit. We somehow need to loop over fruits at the same time as colors.

There are different ways of doing that.

4.3.1 Generate indices with range(len(seq))

If you need to access multiple sequences (like lists, strings, tuples) in one for loop, you can generate indexes by using range(len(sequence)).

```
for index in range(len(sequence)):
    print(index, sequence(index))
fruits = ["loquat", "jujube", "pear"]
colors = ["brown", "orange", "green"]

for i in range(len(colors)):
    print(colors[i], fruits[i])
brown loquat
orange jujube
green pear
```

4.3.2 Generate indices with the enumerate() function

The built-in function enumerate() adds a counter to an iterable and returns it in the form of an enumerating object. This enumerated object can then be used directly in for loops. By default, it starts the count from 0, so you can use the counter as an index.

```
for index, item in enumerate(iterable):
    print(index, item)
```

We can use the index to access values in other sequence types:

```
fruits = ["loquat", "jujube", "pear"]
colors = ["brown", "orange", "green"]

for n, fruit in enumerate(fruits):
    print(colors[n], fruit)
brown loquat
orange jujube
green pear
```

This works only for sequences because they can be indexed starting from 0. For non-sequences, like a dictionary, this is not going to work. **We can't index non-sequences.**

4.3.3 Using the zip() function

The functionality 'zip' in Python is based on the English word 'zip', "closing something with a zipper". The application of zip returns an iterator, which is capable of producing tuples. It is combining the first items of each iterable (in our example lists) into a tuple, after this it combines the second items and so on. It stops when one of them is exhausted, i.e., there are no more items available.

```
a_couple_of_letters = ["a", "b", "c", "d", "e", "f"]
some_numbers = [5, 3, 7, 9, 11, 2]
print(zip(a_couple_of_letters, some_numbers))
<zip object at 0x7efc8724f9b0>
```

The best way to see what it creates is to use it in a for loop.

```
for t in zip(a_couple_of_letters, some_numbers):
    print(t)
('a', 5)
('b', 3)
('c', 7)
('d', 9)
('e', 11)
('f', 2)
```

If you look closely at the output above and the following picture, you will hopefully understand why zip has something to do with a zipper and why they chose the name.

The zip() function takes iterables (can be zero or more), aggregates them in a tuple, and returns it.

```
colors = ["brown", "orange", "green"]
fruits = ["loquat", "jujube", "pear"]

for item in zip(fruits, colors):
    print(item)
('loquat', 'brown')
('jujube', 'orange')
('pear', 'green')
```

The return value of zip is a tuple of each of the items in `colors` and `fruits`, that are in corresponding positions. The first one from each (loquat, brown), the second one from each (jujube, orange), and so on, and it will stop at the shortest one.

We can assign a variable to each element of the tuple and do whatever we'd like with these variables, `fru` and `col` here:

```
fruits = ["loquat", "jujube", "pear"]
colors = ["brown", "orange", "green"]

for fru, col in zip(fruits, colors):
    print(col, fru)
```

```
brown loquat
orange jujube
green pear
```

zip can have an arbitrary number of iterable arguments as we can see in the following example:

```
location = ["Hogoland", "Kiel", "Berlin-Tegel", "Konstanz", "Hohenpeißenberg"]
air_pressure = [1021.2, 1019.9, 1023.7, 1023.1, 1027.7]
temperatures = [6.0, 4.3, 2.7, -1.4, -4.4]
altitude = [4, 27, 37, 443, 977]

for t in zip(location, air_pressure, temperatures, altitude):
    print(t)
('Hogoland', 1021.2, 6.0, 4)
('Kiel', 1019.9, 4.3, 27)
('Berlin-Tegel', 1023.7, 2.7, 37)
('Konstanz', 1023.1, -1.4, 443)
('Hohenpeißenberg', 1027.7, -4.4, 977)
```

The use of zip is not restricted to lists and tuples. It can be applied to all iterable objects like lists, tuples, strings, dictionaries, ranges, and many more.

```
food = ["ham", "spam", "cheese"]
for item in zip(range(1000, 1003), food):
    print(item)
(1000, 'ham')
(1001, 'spam')
(1002, 'cheese')
```

Parameters with Different Lengths

As we have seen, zip can be called with an arbitrary number of iterable objects as arguments. So far the number of elements or the length of these iterables had been the same. This is not necessary. If the lengths are different, zip will stop producing an output as soon as one of the argument sequences is exhausted.

The following example is a zip call with two list with different lengths:

```
colors = ["green", "red", "blue"]
cars = ["BMW", "Alfa Romeo"]
for car, color in zip(cars, colors):
    print(car, color)
BMW green
Alfa Romeo red
```

4.3.4 Converting two Iterables into a Dictionary

zip also offers us an excellent opportunity to convert two iterables into a dictionary. Of course, only if these iterables meet certain requirements. The iterable which should be used as keys must be unique and can only consist of immutables. We demonstrate this with the following morse code example.

```
abc = "abcdef"
morse_chars = [".-.", "-...", "-.-.", "-..", ".", "..-."]
text2morse = dict(zip(abc, morse_chars))
print(text2morse)
{'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.', 'f': '..-.'}
```

4.3.3 Advanced usages of zip – unpacking

We have a list with the six largest cities in Switzerland. It consists of tuples with the pairs city and population number:

```
cities_and_population = [("Zurich", 415367), ("Geneva", 201818), ("Basel", 177654), ("Lausanne", 139111), ("Bern", 133883), ("Winterthur", 111851)]
```

The task consists of creating two lists: one with the city names and one with the population numbers. zip is the solution to this problem, but we also have to use the star operator to unpack the list:

```
cities, populations = list(zip(*cities_and_population))
print(cities)
print(populations)
('Zurich', 'Geneva', 'Basel', 'Lausanne', 'Bern', 'Winterthur')
(415367, 201818, 177654, 139111, 133883, 111851)
```

This is needed e.g., if we want to plot these data. The following example is just used for illustrating purposes. It is not necessary to understand it completely. You have to be familiar with pandas, which we will do the last week of this course, but you can try this example:

```
import pandas as pd
cities_and_population = [("Zurich", 415367), ("Geneva", 201818), ("Basel", 177654), ("Lausanne", 139111), ("Bern", 133883), ("Winterthur", 111851)]

cities, populations = list(zip(*cities_and_population))
s = pd.Series(populations, index=cities)
s.plot(kind="bar")
```

4.3.4 Using zip() and enumerate() functions

You can combine zip and enumerate to both loop over multiple iterables and generate indices.

```
names = ['Mukesh', 'Roni', 'Chari']
ages = [24, 50, 18]

for i, (name, age) in enumerate(zip(names, ages)):
    print(i, name, age)
0 Mukesh 24
1 Roni 50
2 Chari 18
```

4.5 Nested loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

```
for item1 in iterable1:
    for item2 in iterable2:
        statements(s)
        statements(s)

number=[1,2,3]
color=['blue','yellow','red']

#For every number print each color:
for x in number:
    for y in color:
        print(x,y)
```

Summary of loops over iterable objects

```
for var in iterable:
    statement(s)    #must indent
```

| for var in iterable: | When to use |
|-----------------------------------------|---------------------------------------------------------------------------|
| for item in list: | Loop over items in a list |
| for character in string: | Loop over characters in a string |
| for i in range(n): | Loop over integer numbers |
| for index, item in enumerate(sequence): | Loop over both indexes and values |
| for i in range(len(sequence)): | Generate indices, and can use them to access values in multiple sequences |
| for k in dictionary: | Looping over the keys |

| | |
|-----------------------------------------------------------|-----------------------------------------------------------|
| <code>for v in dictionary.values():</code> | Looping over the values |
| <code>for k,v in dictionary.items():</code> | Looping over both keys and values |
| <code>for item1,item2 in zip(iterable1,iterable2):</code> | Looping over items of multiple sequences at the same time |

4.6 Conditional statements

If-statements are used in similar manners found in other languages.

In Python, the syntax for if-statements is the following – the logic is the same as in bash.

simple if statement

```
if test1:
    statement(s) #must indent
```

if-else statement

```
if test1:
    statement(s) #must indent
else:
    statement(s) #must indent
```

if-elif-else statement

```
if test1:
    statement(s) #must indent
elif test2:
    statement(s) #must indent
else:
    statement(s) #must indent
```

4.5.1 Test condition syntax

Python Comparison Operators

Comparison operators are used to compare two values:

| Operator | Name |
|--------------------|--------------------------|
| <code>==</code> | Equal |
| <code>!=</code> | Not equal |
| <code>></code> | Greater than |
| <code><</code> | Less than |
| <code>>=</code> | Greater than or equal to |
| <code><=</code> | Less than or equal to |

For example:

```
#simple if
age = 20

if age > 18:
    print("I can vote")
```

```
#if-else
age = 20

if age > 18:
    print("I can vote")
else:
    print("I cannot vote")
```

```
#if-elif-else
age = 20

if age > 18:
    print("I can vote")
elif age == 18:
    print("I just turned 18. I can vote.")
else:
    print("I cannot vote")
```

Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

| Operator | Description |
|---------------------|----------------------------------------------------------------------------------|
| <code>in</code> | Returns True if a sequence with the specified value is present in the object |
| <code>not in</code> | Returns True if a sequence with the specified value is not present in the object |

Here some examples:

You can test if a character is or is not in a string

```
'p' in 'python'      #test if character is in string
'py' not in 'python' #test if characters are not in string
```

You can test if an item is or is not in a list

```
1 in [1,2,3]        #test if item is in list
1 not in [1,2,3]     #test if item is not in list
```

You can test if a key is or is not in a dictionary:

```
D1={1:'a',2:'b',3:'c'}
1 in D1
1 not in D1
```

You can test if a value is or is not in a dictionary

```
'a' in D1.values()
'a' not in D1.values()
```

Example:

```
D1={1:'a',2:'b',3:'c'}
if 'a' in D1.values():
    print("Yes it is")
```

Python Logical Operators

Logical operators are used to combine conditional statements:

| Operator | Description |
|------------|---------------------------------------------------------|
| and | Returns True if both statements are true |
| or | Returns True if one of the statements is true |
| not | Reverse the result, returns False if the result is true |

| A | B | A AND B | A OR B | NOT A |
|-------|-------|---------|--------|-------|
| False | False | False | False | True |
| False | True | False | True | True |
| True | False | False | True | False |
| True | True | True | True | False |

Example

```
D1={1:'a',2:'b',3:'c'}
if 'p' in 'python' and 'a' in D1.values():
    print("Yes they are")
```

5.6 if-break and if-continue statement in loops

The `continue` statement allows skipping of code within a single loop if criteria have been met. In this case if a match is found go to next entry:

```
for name in ['Newton', 'Galileo', 'Euler']:
    if 'G' in name:
        continue
    print(name)
```

```
Newton
Euler
```

The `break` statement will stop the current loop and continue with statements following the loop:

```
for name in ['Newton', 'Galileo', 'Euler']:
    if 'G' in name:
        break
    print(name)
```

```
Newton
```

The `break` and `continue` statements are also used in while loops. See section 5.8

4.7 While Conditional

A while loop is an **indefinite** iteration, in which the code block executes until some condition is met. Test condition syntax is the same as for if statements.

```
while test:
    statement(s)
```

Here is an example, we also did it in bash

```
num = 0
while num < 5:
    num = num + 1
    print(num)
1
2
3
4
5
```

We can convert this while loop into a for loop

```
for i in range(1,6):
    print(i)
1
2
3
4
5
```

There are cases in which a while loop is necessary, and it is when we do not know when the test condition is met, for example, if we use the input functions or random numbers.

For example, we want to ask the user to guess a number, and we prompt the user over and over until the user guesses the number 10

```
mynumber=10
number=0 #we start this variable for the while loop to start True.

while number!=mynumber:
    number=int(input("Enter an integer number between 1-10: "))
print("You got the number: " ,number)
```

The while loop will end when the user enters the number 10.

There is another way to implement this, and it is by using while True in combination with an if-break statement:

```
mynumber=10
while True:
    number=int(input("Enter an integer number between 1-10: "))
    if number==mynumber:
        break
print("You got the number: " ,number)
```

while True, means the while loop condition is always True, and we use the if-break conditional statement to break the loop. Notice that in this case the condition is set to break the loop, i.e., to end the loop.

Another example is with random numbers. We make a while loop that will print random numbers in range 1-10, and will end when the random number is 8

```
import random
rand_num=0

while rand_num!=8:
    print(rand_num)
    rand_num = random.randint(1,11)
```

We can implement it by also using while True, and if-break

```
import random
while True:
    rand_num = random.randint(1,11)
    if rand_num == 8:
        break
    print(rand_num)
```

4.8 Summing loop

The basic steps of a summing loop are the following:

- Initialize a variable to 0
- Loop over an iterable or range of elements.
- Use + to sum the elements

Let's sum the squares of the following list

```
L=[100,300,500]

s=0 # initialize s=0
for i in L: # loop over iterable
    s=s+(i**2) # sum the squares
print(s) # print total sum
```

4.8.1 Create strings using loops

You can use a summing loop structure to create a string, where now the + sign performs concatenation.

- Initialize an empty string.

- Loop over an iterable or range of elements.
- Use concatenation to build up the string

```
veggie=['spinach', 'broccoli', 'edamame', 'bell pepper', 'kale', 'cabbage',  
'celery', 'asparagus', 'lettuce']  
  
s='' # initialize empty string  
for v in veggie: # loop over iterable  
    if len(v)==7:  
        s=s+v # construct the string by using concatenation  
print(s)  
spinachedamamecabbagellettuce
```

In this example, we loop over integer types and convert each integer into a string before performing string concatenation.

```
num=[30,15,25,11,3,4,1,5,6]  
  
s1=''  
for i in num:  
    s1=s1+str(i) # convert i to string type to then concatenate  
print(s1)  
3015251134156
```

Chapter 5. Construct data types - Comprehensions and loops

Python is famous for allowing you to write code that's elegant, easy to write, and almost as easy to read as plain English. One of the language's most distinctive features is comprehension, which you can use to create powerful functionality within a single line of code.

Comprehensions provide us with a short and concise way to construct new sequences (such as lists, sets, dictionaries, etc.) using sequences which have been already defined. Python supports the following 4 types of comprehensions:

- List Comprehensions
- Dictionary Comprehensions
- Set Comprehensions
- Generator Comprehensions

In this chapter we will go over constructing lists and dictionaries by using comprehensions and loops, and compare them.

5.1.1 List Comprehension

List Comprehensions provide an elegant way to create new lists. The following is the basic structure of a list comprehension:

```
new_list = [expression for member in iterable]
```

If you want to create a list containing the first ten perfect squares

```
squares = [i * i for i in range(10)]
print(squares)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Every list comprehension in Python includes three elements:

- **expression** is the member itself, a call to a method, or any other valid expression that returns a value.
In the example above, the expression `i * i` is the square of the member value.
- **member** is the object or value in the list or iterable. In the example above, the member value is `i`.
- **iterable** is a sequence or any other object that can return its elements one at a time. In the example above, the iterable is `range(10)`.

Expression

The *expression* is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list:

Example: we can convert the values in the new list to upper case:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x.upper() for x in fruits]
print(newlist)
['APPLE', 'BANANA', 'CHERRY', 'KIWI', 'MANGO']
```

You can set the outcome to whatever you like.

Example: we can set all values in the new list to 'hello':

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = ['hello' for x in fruits]
print(newlist)
['hello', 'hello', 'hello', 'hello', 'hello']
```

Using Conditional Logic in list comprehension

A more complete description of the comprehension formula adds support for optional **conditionals**. The most common way to add conditional logic to a list comprehension is to add a conditional to the end of the expression:

```
new_list = [expression for member in iterable (if condition)]
```

Conditions are important because they allow list comprehensions to select values. The *condition* selects items that evaluate to `True`. Use the same syntax you would use for test conditions in if statements and while loops.

In this example the conditional statement selects any string in fruits containing the character a.

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]
print(newlist)
['apple', 'banana', 'mango']
```

Here the conditional statement selects only items that are not "apple":

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if x != "apple"]
print(newlist)
['banana', 'cherry', 'kiwi', 'mango']
```

The condition `if x != "apple"` will return `True` for all elements other than "apple", making the new list contain all fruits except "apple".

Replacing values based on conditions in list comprehension.

You can place the condition at the end of the statement for selecting based on a test condition, but if you want to *change* a member value based on a test, it's useful to place the conditional if/else near the *beginning* of the expression:

```
new_list = [expression (if conditional) for member in iterable]
```

For example, if you have a list of prices, then you may want to replace negative prices with 0 and leave the positive values unchanged:

```
original_prices = [1.25, -9.45, 10.22, 3.78, -5.92, 1.16]
prices = [i if i > 0 else 0 for i in original_prices]
print(prices)
[1.25, 0, 10.22, 3.78, 0, 1.16]
```

Here, your expression `i` contains a conditional statement, `if i > 0 else 0`. This tells Python to output the value of `i` if the number is positive, but to change `i` to 0 if the number is negative.

6.1.2 Construct Lists in Python

There are a few different ways you can create lists in Python. To better understand the trade-offs of using a list comprehension in Python, let's first see how to create lists with these approaches.

Using for loop

The most common type of loop is the `for` loop. You can use a `for` loop to create a list of elements in three steps:

- Initialize an empty list.
- Loop over an iterable or range of elements.
- Append each element to the end of the list.

If you want to create a list containing the first ten perfect squares, then you can complete these steps in three lines of code:

```
squares = []
for i in range(10):
    squares.append(i * i)
print(squares)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Using while loop

You can use a while loop to create a list of elements in three steps:

1. Initialize an empty list.
2. Make a while loop
3. Within the while loop append (or extend) each element to the end of the list.

If you want to create a list of 4 unique integer random numbers in range 1-10

```
import random
L=[]
while len(L)!=4:
    num=random.randint(1,10)
    if num not in L:
        L.append(num)
print(L)
```

Using List Comprehension

List comprehensions are a third way of making lists. With this elegant approach, you could rewrite the `for` loop from the first example in just a single line of code:

```
squares = [i * i for i in range(10)]
print(squares)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Example: we want to make a new list by selecting fruit names containing both the character `a` and length of 5.

```
#with a for loop
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []
for x in fruits:
    if "a" in x and len(x)==5:
```

```
newlist.append(x)
print(newlist)
['apple', 'mango']
```

```
#with list comprehension
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x and len(x)==5]
print(newlist)
['apple', 'mango']
```

Converting Celsius values into Fahrenheit with list comprehension:

```
Celsius = [39.2, 36.5, 37.3, 37.8]
Fahrenheit = [ ((float(9)/5)*x + 32) for x in Celsius ]
print(Fahrenheit)
[102.56, 97.7, 99.14, 100.03999999999999]
```

Key Points to Remember

- List comprehension is an elegant way to define and create lists based on existing lists.
- List comprehension is generally more compact and faster than normal functions and loops for creating lists.
- However, we should avoid writing very long list comprehensions in one line to ensure that code is user-friendly.
- Remember, every list comprehension can be rewritten in a for loop, but not every for loop can be rewritten in the form of list comprehension.

Benefits of Using List Comprehensions

Many developers struggle to fully leverage the more advanced features of a list comprehension in Python. Some programmers even use them too much, which can lead to code that's less efficient and harder to read. For this, it's worth it to understand the benefits of using a list comprehension in Python when compared to the alternatives, and also learn about a few scenarios where the alternatives are a better choice.

One main benefit of using a list comprehension in Python is that it's a single tool that you can use in many different situations. This is the main reason why list comprehensions are considered powerful tools that you can use in a wide variety of situations.

List comprehensions are also more **declarative** than loops, which means they are easier to read and understand. Loops require you to focus on how the list is created. You have to manually create an empty list, loop over the elements, and add each of them to the end of the list. With a list comprehension in Python, you can instead focus on *what* you want to go in the list and trust that Python will take care of *how* the list construction takes place.

When Not to Use a List Comprehension in Python

List comprehensions are useful and can help you write elegant code that's easy to read and debug, but they're not the right choice for all circumstances. They might make your code run more slowly or use more

memory. If your code is less performant or harder to understand, then it's probably better to choose an alternative.

A list comprehension in Python works by loading the entire output list into memory. For small or even medium-sized lists, this is generally fine. If you want to sum the squares of the first one-thousand integers, then a list comprehension will solve this problem admirably. But if you wanted to sum the squares of the first *billion* integers then you may notice that your computer becomes non-responsive. That's because Python is trying to create a list with one billion integers, which consumes more memory than your computer would like. Your computer may not have the resources it needs to generate an enormous list and store it in memory, so it would be more efficient to use other methods.

6.1.3 Dictionary Comprehensions

Extending the idea of list comprehensions, we can also create a dictionary using dictionary comprehensions. The structure of a dictionary comprehension is the following:

```
output_dict = {key:value for item in iterable (if conditional)}
```

The if conditional is optional.

```
words = ['data', 'science', 'machine', 'learning']
D={i:len(i) for i in words}
print(D)
{'data': 4, 'science': 7, 'machine': 7, 'learning': 8}
```

In the dictionary comprehension, we need to specify both keys and values based on the iteration. The returned dictionary contains the words as keys and their length as values.

For this example, we will repeat the task in the first example with an additional condition. The dictionary comprehensions accept if/else conditional statements.

```
words = ['data', 'science', 'machine', 'learning']
D={i:len(i) for i in words if len(i) > 5}
print(D)
{'science': 7, 'machine': 7, 'learning': 8}
```

The returned variables only contain the words longer than 5 characters.

Replacing values based on conditions in dictionary comprehension

As with list comprehensions, to replace values based on conditions you can place the if conditional after the key:value

```
output_dict = {key:value (if conditional) for item in iterable}
```

```
words = ['data', 'science', 'machine', 'learning']
words_dict = {i:len(i) if len(i) > 5 else 'short' for i in words}
```

```
print(words_dict)
{'data': 'short', 'science': 7, 'machine': 7, 'learning': 8}
```

We implement an if/else conditional in the dictionary comprehension. If the length is greater than 5, the value becomes the length. Otherwise, we assign the word 'short' as the value.

What makes comprehensions appealing is their one liner syntax. It looks quite simple and is easier to understand than the equivalent for loops.

For instance, the equivalent for loop of the comprehension above is:

```
words_dict={}
for i in words:
    if len(i) > 5:
        words_dict[i]=len(i)
    else:
        words_dict[i]='short'

print(words_dict)
{'data': 'short', 'science': 7, 'machine': 7, 'learning': 8}
```

Using zip in dictionary comprehension

We can iterate over two iterables in a dictionary comprehension by using the zip function. The zip function returns an iterable of tuples by combining the items from each list. Key-value pairs are created by iterating over separate lists for keys and values.

```
Words = ['data', 'science', 'machine', 'learning']
values = [5, 3, 1, 8]
dict_a = {i:j for i, j in zip(words, values)}
print(dict_a)
{'data': 5, 'science': 3, 'machine': 1, 'learning': 8}
```

We can also put a condition on the values when iterating over a list of tuples

```
words = ['data', 'science', 'machine', 'learning']
values = [5, 3, 1, 8]
dict_a = {i:j for i, j in zip(words, values) if j > 4}
print(dict_a)
{'data': 5, 'learning': 8}
```

Example: we can also apply transformations on key-value pairs

```
words = ['data', 'science', 'machine', 'learning']
values = [5, 3, 1, 8]

dict_b = {i.upper():j**2 for i, j in zip(words, values)}
```

```
print(dict_b)
{'DATA': 25, 'SCIENCE': 9, 'MACHINE': 1, 'LEARNING': 64}
```

Both keys and values are modified using simple Python methods.

Using items() in dictionary comprehension

We can use the items of an existing dictionary as an iterable in a dictionary comprehension. It allows us to create dictionaries based on existing dictionaries and modify both keys and values.

```
D={'DATA': 25, 'SCIENCE': 9, 'MACHINE': 1, 'LEARNING': 64}
D1 = {i.lower():j*2 for i, j in D.items()}
print(D1)
{'data': 50, 'science': 18, 'machine': 2, 'learning': 128}
```

Using enumerate() in dictionary comprehension

The enumerate function of Python can be used to create an iterable of tuples based on a list. Each tuple contains the items in the list with incrementing integer values.

```
names = ['John', 'Jane', 'Adam', 'Eva', 'Ashley']
print(list(enumerate(names)))
[(0, 'John'), (1, 'Jane'), (2, 'Adam'), (3, 'Eva'), (4, 'Ashley')]
```

We can use the enumerate function in a dictionary comprehension

```
names = ['John', 'Jane', 'Adam', 'Eva', 'Ashley']
dict_names = {i:len(j) for i, j in enumerate(names)}
print(dict_names)
{0: 4, 1: 4, 2: 4, 3: 3, 4: 6}
```

This example contains a slightly more complicated conditional than the previous ones.

Consider we have the following dictionary and list:

```
lst = ['data', 'science', 'artificial', 'intelligence']
dct = {'data': 5, 'science': 3, 'machine': 1, 'learning': 8}
```

We want to create a new dictionary using the list and dictionary defined above. The keys of the new dictionary will be the elements in the list so we will iterate over the elements in list. If the element is also in the dictionary, the value will be the value of that key in the dictionary. Otherwise, the value will be the length of the key.

```
D={i:dct[i] if i in dct else len(i) for i in lst}
print(D)
{'artificial': 10, 'data': 5, 'intelligence': 12, 'science': 3}
```

The word `artificial` is not in the dictionary so its value is the length of the word. The word `data` is in the dictionary, so its value is taken from the dictionary.

Dictionaries are very important data structures in Python and are used in many cases. The examples we did in this post will cover most of what you need to know about dictionary comprehensions. They will make you feel comfortable when working with and creating new dictionaries.

4.8.3 Construct dictionaries using loops

Using for loops

You can use a for loop to create a dictionary in three steps:

1. Initialize an empty dictionary.
2. Loop over an iterable or range of elements.
3. Add key-value pair with `D[key]=value` or `D.update({key:value})`

```
veggie=['spinach', 'broccoli', 'edamame', 'bell pepper', 'kale', 'cabbage',
        'celery', 'asparagus', 'lettuce']

num=[30,15,25,11,3,4,1,5,6]

D={}          #initialize empty dictionary

for k,v in zip(veggie,num): #loop over iterable and define key and value
    if 'a' in k and v < 10:
        D[k]=v             # or D.update({k:v}) to construct the dictionary

print(D)
{'kale': 3, 'cabbage': 4, 'asparagus': 5}
```

Using while loops

- Initialize an empty dictionary.
- Make a while loop
- Within the while loop add key-value pair with `D[key]=value` or `D.update({key:value})`

If you want to create a dictionary where the keys are 4 integer random numbers in range 1-10, and the values are the corresponding squares

```
import random
D={}
while len(D)!=4:
    num=random.randint(1,10)
    D[num]=num**2
print(D)
{10: 100, 8: 64, 5: 25, 1: 1}
```

Chapter 6. Writing your own Functions

In programming, a **function** is a self-contained block of code that encapsulates a specific task or related group of tasks. As you already know, Python gives you many built-in functions like `print()`, `type()`, `float()`, `int()`, `input()`, `list()`, `tuple()`, etc. You can also create your own functions, which are called *user-defined functions*.

Here some examples of built-in Python functions:

```
M=max([1,3,5])      #takes one list of numbers and returns the maximum value
print(M)

print(100*90)       #takes arguments and prints them to screen
```

Each of these built-in functions performs a specific task. The code that accomplishes the task is defined somewhere, but you don't need to know where. All you need to know about the function is:

- what **arguments** (if any) it takes
- what **values** (if any) it returns

Then you call a built-in function and pass the appropriate arguments (values). When you call a built-in function, e.g. `max([1,2,3])`, program execution goes off to the designated body of function code. When the function is finished, execution returns to your code where it left off. The function may or may not return data for your code to use, as the examples above do.

When you define your own Python function, it works just the same, but now, you know where the designated body of function code is and what it specifically does, because you wrote it. From somewhere in your code, you'll call your Python function and program execution will transfer to the designated body of function code. When the function is finished, execution returns to the location where the function was called.

Virtually all programming languages used today support a form of **user-defined functions**, although they aren't always called functions. In other languages, they might be called **subroutines**, **procedures**, or **subprograms**. There are several very good reasons why functions are important.

Reusability

Suppose you write some code that does something useful, and you find that the task performed by that code is one you need often, in many different locations within your script. You could just replicate the code over and over again, using your editor's copy-and-paste capability. However, the [Don't Repeat Yourself \(DRY\) Principle](#) of software development will convince you that a better solution is to **define a Python function that performs that task**. Anywhere in your script that you need to accomplish that task, you simply call the function. If you decide to change how that task works, then you only need to change the code in one location, which is the place where the function is defined. The changes will automatically be picked up anywhere the function is called.

Modularity

Functions allow **complex processes** to be broken up into smaller steps. Imagine, for example, that you have a program that reads in a file, processes the file contents, and then writes an output file. Instead of stringing all the code together, you can break it out into separate functions, each of which focuses on a specific task. Those tasks would be *read*, *process*, and *write*. After you define the functions, you would then simply need to call each of them.

6.1 Function Definition and Call – with return statement

The general syntax for defining a Python function with a return statement is:

```
def function_name(parameters):
    "function_docstring"
    code block
    return expression
```

Once a function has been defined, it can be called by name, supplying the necessary variables in parentheses in the same way as for a built-in function. A function with a return statement will return an object, which you can store in a variable, in this case var.

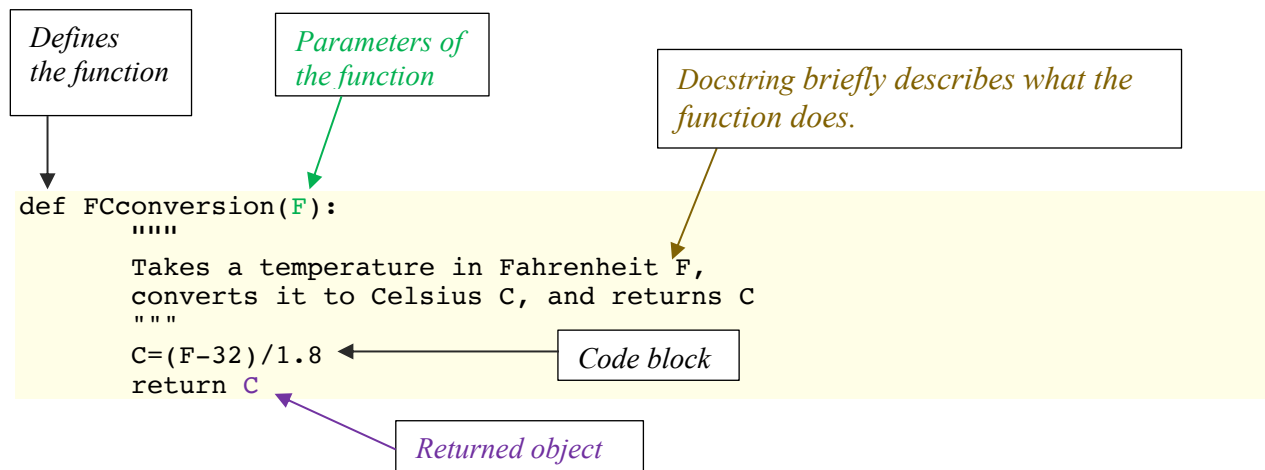
```
var=function_name(arguments)
print(var)
```

A parameter is a variable listed inside the parentheses in the function definition.

An argument is the value that is passed to the function when it is called.

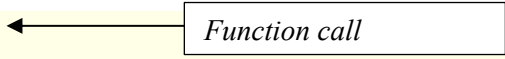
A return statement ends execution of the function and return an object to the caller, which you can store in a variable.

Let's make a function that implements the Fahrenheit to Celsius formula:



Function call. Use the function name followed by parentheses and supply the argument. The function returns an object which we can store in a variable.

```
y=FCconversion(100)
print(y)
```



In the function call, **100** is the argument, which is passed to the function, and it is assigned to the parameter **F** via a mechanism of argument passing.

The block of code in the function is executed.

The return statement passes back to the function call the object **C**, which can be stored in the variable **y**.

6.2 Parameters of a function

A parameter is a name between the function definition's parentheses. An argument can be passed to a function by specifying the value in the parentheses of the function's call. Functions can have zero, one, two ... parameters.

Example: Here's an example of a function that takes **one parameter** and returns one value.

```
def f1(par):
    value=par*par
    return value
```

Call the function and pass the value 10:

```
x=f1(10)
print(x)
100
```

Example: This function takes **two parameters** and returns one value.

```
def f2(par1, par2):
    return par1*par2
```

```
y=f2("first", 2)
print(y)
firstfirst
```

Example: This function takes **no parameter** and returns a value:

```
def f3():
    print('Hi')
    return 'Hello'
```

```
y=f3()  
print(y)
```

What is stored in variable y?

6.3 Functions can return multiple values.

A function can return **multiple** values. These multiple values can be returned as a tuple type.

```
def f4():  
    return 'Hello', 'World'  
  
x=f4()  
  
print(x)  
( 'Hello', 'World' )
```

Multiple values can also be returned via a list type.

6.4 Function definition and call - without return statement.

A **return** statement is used to return values. All the above functions f, f1, f2, f3 and f4 have a return statement.

In Python, it is possible to compose a function without a return statement. If we use the "return" keyword alone, or "return None", or we omit the keyword return, no value (class `NoneType`) is returned. This means that even if a calculation is performed inside the function, the result of the calculation cannot be assigned to a variable when you call the function.

```
def function_name(parameters):  
    "function_docstring"  
    code block
```

In the function call, write the function name and supply the arguments. Since there is no object returned, the *None* type is instead returned. Notice we do not store in a variable the *None* type.

```
function_name(arguments)
```

Let's modify the function `FCconvert`, and instead of returning the temperature in Celsius, it prints it.


```
def FCconversion(F):
    """
    Takes a temperature in Fahrenheit F,
    converts it to Celsius C, and prints C
    """
    C=(F-32)/1.8
    print(C)
```

The function call is now:

```
FCconversion(100)
```

Here other examples.

Let's define a function that calculates something and then prints out the calculated value

```
def printme(inpl):
    x=inpl*inpl
    print(x)

printme(2)
4
```

This function takes no parameters and prints

```
def print_none():
    print("I have no arguments!")
```

```
print_none()
I have no arguments!
```

This function takes two parameters and prints:

```
def print_two(arg1, arg2):
    print(arg1*arg2)
```

```
print_two("first", 2)
firstfirst
```

6.5 Local and Global Variables

If you define a variable inside of a function (local variable), that variable will not be available outside of that function:

```
def f5(arg1):
    myvar1=5          #local variable – defined inside a function
    return myvar*arg1

print(myvar1)
NameError: name 'myvar1' is not defined
```

Local variables can be used only inside the function in which they are declared.

You can define variables outside of a function (global variables). Global variables can be used throughout the program, including inside a function:

```
myvar=5              #global variable – defined outside a function
```

```
def f7(arg1):
    return myvar*arg1

print(myvar)
x=f7(5)
25
```

6.6 Using flow control in functions

If – else statements and loops can be used in the code block of a function.

Try these examples in a script called **func-if.py**

```
def c1(par):
    if par < 2:
        return "Success"
    else:
        return "Fail"

x1=c1(1)
print(x1)
x2=c1(3)
print(x2)
```

Run the script.

Here' another example

```
def c3(s1):
    if not ("Python" == s1):
        print("It really is! ")

c3("Fun")
```

Run the script.

Here examples on using loops. In a script called **my_upper.py** make a function called `string_upper`. The function takes one string as parameter, converts it to uppercase, and print each character uppercase.

```
def string_upper(mystr):
    for c in mystr:
        print(c.upper())

string_upper("music")
```

In a script called **conv_numstring.py**, make a function called `num_str` that takes one integer as parameter, and returns a string of the numbers from 0 to n-1. Example, if the integer value is 6, the returned string will be '012345'

```
def num_str(n):
    s1 = ''
    for i in range(n):
```

```

        s1 = s1 + str(i)
    return s1

s=num_str(6)
print(s)

```

6.7 Recursive Algorithms

A recursive algorithm calls itself.

```

def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(10)) # function call

```

A recursive algorithm must have a termination condition:

n == 0

And a reduction step where the function calls itself

factorial(n - 1)

Python by default has a limit to the depth of recursion available, to avoid absorbing all of the computer's memory. To see what the limit is type at the IPython console:

```

import sys
sys.getrecursionlimit()

```

and to set it :

```

sys.setrecursionlimit()

```

6.8 Functions in comprehensions

Functions that have a return statement can be called within comprehensions.

For example:

```

def FCconversion(F):
    """
    Takes a temperature in Fahrenheit F,
    converts it to Celsius C, and return C
    """
    C=(F-32)/1.8
    return(C)

```

We create a list of temperature values in Celsius from temperature values 1-100 in Fahrenheit:

```
L=[ FCconversion(i) for i in range(1,101)]
```

Chapter 7. Built-in Modules: math, random, sys

The Python interpreter has several built-in functions. They are loaded automatically as the interpreter starts and are always available. For example, `print()` and `input()` for Input/Output, type conversion functions `int()`, `float()`, `list()`, `tuple()`, `set()`, etc.

In addition to built-in functions, many pre-defined functions are also available as a part of libraries bundled with Python distributions. These functions are defined in modules. A module is a file containing definitions of functions, classes, variables, constants, or any other Python objects that logically belong together. Contents of this file can be made available to any other program. Built-in modules are written in C and integrated with the Python interpreter.

To display a list of all available modules, use the following command in the Python console/shell command line:

```
help('modules')
```

Pick one module you would like to know about, and which functions it contains, e.g. the `math` module, and type

```
help("math")
```

Import a module

We will start by learning how to use functions from a module.

If you want to use all functions and constants available from the `math` modules, you can use the:

generic import:

```
import modulename
modulename.function()
modulename.constant
```

If you want to only use specific functions, you can use the so-called

function import:

```
from modulename import function1, function2, constant
function1()
constant
```

7.1 The math module

Mathematical calculations are an essential part of most Python development. Whether you're working on a scientific project, a financial application, or any other type of programming endeavor, you just can't escape the need for math. For straightforward mathematical calculations in Python, you can use the built-in mathematical **operators**, such as addition (+), subtraction (-), division (/), and multiplication (*). But more advanced operations, such as exponential, logarithmic, trigonometric, or power functions, are not built in. Does that mean you need to implement all these functions from scratch? Fortunately, no. Python provides a [module](#) specifically designed for higher-level mathematical operations: the **math** module.

The **math** module consists of several basic math functions, which include trigonometric functions, logarithmic functions, angle conversion functions, etc. well-known mathematical constants, such π (pi) and the Euler number (e).

To display all functions, and constants of the math module

```
help("math")
```

Since the **math** module comes packaged with the Python release, you don't have to install it separately.

Using it is just a matter of importing the module:

You must import the math module before you can use its functions.

This is achieved through **import**.

Generic import

If you want to use all functions and constants available in the math module, you can use the **generic import**:

```
import math
math.log(x)    #you must also specify the module name
0.6931471805599453
```

`math.log(x)` returns the logarithm of base e.

You can access pi as follows:

```
math.pi
3.141592653589793
```

As you can see, the pi value is given to fifteen decimal places in Python. The number of digits provided depends on the underlying C compiler. Python prints the first fifteen digits by default, and `math.pi` always returns a float value.

You can calculate the circumference of a circle using $2\pi r$, where r is the radius of the circle:

```
r=10
circumference = 2 * math.pi * r
62.83185307179586
```

You can also rename the module name in your code

```
import math as m
m.log(x)    #you must now specify the new name of the module
0.6931471805599453
```

function import

If you want to only use specific functions and/or constants, you can use the so-called **function import**:

```
from math import log, sqrt, pi
```

This function import will only make the log and sqrt functions and pi constant available from the math module.

```
log(100)    #only the function name
4.605170185988092
```

```
sqrt(100)
100
```

```
r=10
area = pi * r * r
314.1592653589793
```

7.2 The random module

Random numbers are often used in scientific computing and statistics. A random number generator can be used to calculate random moves in calculations and can randomize data. Many options exist in the **random** module. You can use the `dir()` function to explore this module further.

To generate a random number:

```
import random
random.random()    #generate a random number over [0,1)
0.54393021341243169
```

```
random.random()    #generate again a random number over [0,1)
0.78493620464893827
```

```
random.randint(1,9)    # generate a random integer over [1,9]
```

5

The **choice()** function of the random module provides a quick way to randomly select an element from a list or a string:

```
seq= ['G', 'G', 'C', 'C', 'T', 'C', 'T', 'C', 'G', 'A', 'T']
s='GGCCTTCTC'
random.choice(s)
'C'
random.choice(s)
'T'
```

```
random.choice(seq)
'G'
```

```
random.choice(seq)
'A'
```

The **shuffle()** function of the random module will randomly shuffle elements of a list L

```
random.shuffle(seq) #list method, returns None
print(seq)
['G', 'G', 'A', 'T', 'C', 'T', 'A', 'C', 'T', 'T', 'G', 'C', 'C']
```

7.3 The sys module – argv attribute

The **sys** module is mostly useful for the experienced programmer. However, there are a few things even a beginner can make use of. In this course we will use a very important sys attribute, which is called **argv**. This attribute stores command line arguments as a list of strings. A python script can receive arguments from the command line (similar to the `$@` for bash). To see this in action, edit a python file named **argv_test.py** and write the following lines to your file:

```
from sys import argv    #use function import
print(argv)
print(argv[0])
print(argv[1])
print(argv[1:])
print(argv[2])
```

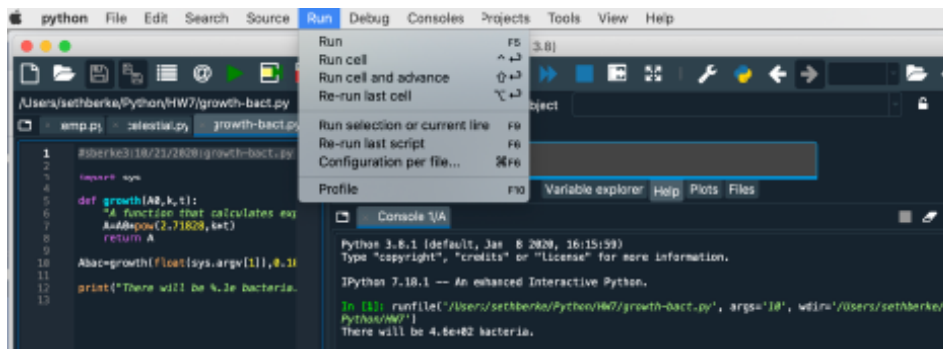
Notice we use function import, since we only need the argv attribute.

Let's pass these arguments `test1 test2 test3 100 200` to the script

To pass arguments to a script in Spyder do the following:

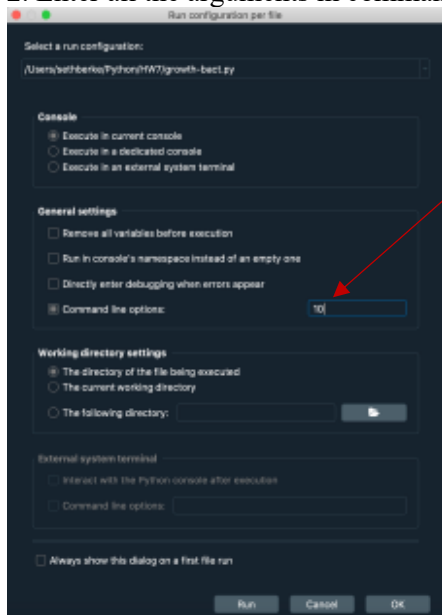
1. Click Run → Configuration per file





2. Enter all the arguments in command line options

test1 test2 test3 100 200



3. Run the code

```
['/Users/maria/argv_test.py', 'test1', 'test2', 'test3', '100', '200']
/Users/maria/argv_test.py
test1
['test1', 'test2', 'test3', '100', '200']
test2
```

Notice that `argv` stores a list of strings, and the first item of the string is the script name (including the pathname). The second item `argv[1]` is the first argument you pass to the script.

Passing arguments from IPython Console:

You can also pass arguments by running the script at IPython console like this:

```
runfile('path to the file', args='arg1 arg2 arg3')
```

You can use absolute path


```
runfile('/Users/maria/argv_test.py', args='test1 test2 test3 100 200')
```

or a relative path, relative to your current directory in IPython – use `pwd` if you want to know your current directory

```
runfile('argv_test.py', args='test1 test2 test3 100 200')
```

The arguments should be separated by a space, and all enclosed within single quotes.

Passing arguments from bash terminal:

You can also open a bash terminal, and run the python script like this

```
python3 argv_test.py test1 test2 test3 100 200
```

Chapter 8. Writing your own Modules

Modular programming design is used to break down a complex system into smaller parts or components, i.e., modules. These components can be independently created and tested. Modules contain functions, variables and other things that logically belong together. Some modules are built into Python (like the `math`, `random` and `sys` modules), other modules can be downloaded separately (like `Numpy`, `Matplotlib`, `pandas`), or you can write your own modules.

To write your own module, simply include in a python script some functions and variables that logically belong together. If you have a module that includes functions implementing the laws of classical physics, don't include in this module functions that deal with the human brain!

Also, always include a doc string in your modules. A module doc string is a text between triple quotes to be included at the beginning of a module. You cannot use Python reserved names, such as, for example, built-in function names – you will not be able to import the module. Also, names of modules must conform to the same rules as for variable names.

8.1 Anatomy of a module

Here we report the basic parts of a module. We call this module `geo.py`.

A Python module is a python file which you can import and use it into another Python program, like you do with, for example, the `math` module.

Docstring describes what the module is about

```
"""This module contains functions to calculate the square
and cube of a number, and some variables."""
```

```
a=10
b=5
```

Define variables

```
def square(n):
    "calculates square of a number"
    return n**2

def cube(n):
    "calculates cube of a number"
    return n**3

if __name__ == "__main__":
    c1=square(a)          #call function to test it
    c2=cube(b)            #call function to test it
    print("test: the square of", a, "is", c1)
    print("test: the cube of", b, "is", c2)
```

Define functions

Test functions within the if statement block

Run the module geo.py

```
test: the square of 10 is 100
test: the cube of 5 is 125
```

Notice the if statement `if __name__ == "__main__":` is True.

Now import this module into another python script and use its functions and variables.

Make a script called **use-geo.py**

```
import geo      #we use generic import
num=int(input("Enter an integer number: "))

x=geo.square(num)
print("the square of", num, "is", x)
print(geo.a)
Enter an integer number: 2
the square of 2 is 4
10
```

Notice that when we import geo, the `if __name__ == "__main__":` in the geo module is False, and the if code block in geo is not executed. When we import a module, Python executes the code in the module.

8.1.1 `__name__` and `"__main__"`

Before executing a Python script, Python interpreter defines a few special variables.

`__name__` is one such special variable.

If the file is executed as the main program (you run the script), the interpreter sets the `__name__` variable to have a string value `"__main__"`.

If this file is being imported from another module, `__name__` will be set to the module's name.

Advantages:

- Every Python module has its `__name__` defined and if this is `'__main__'`, it implies that the module is being run standalone by the user and we can do corresponding appropriate actions.
- If you import this script as a module in another script, the `__name__` is set to the name of the script/module.
- Python files can act as either reusable modules, or as standalone programs.
- **`if __name__ == "main"`**: is used to execute some code only if the file was run directly, and not imported.

Try this by adding the following statement to the **geo.py** module before the if-statement.

```
print("Variable __name__ is: ", __name__)
```

Now run the geo.py module:

```
Variable __name__ is: __main__
test: the square of 10 is 100
test: the cube of 5 is 125
```

Now run the use-geo.py, which imports the module geo.

```
Variable __name__ is: geo
Enter an integer number: 2
the square of 2 is 4
the cube of 2 is 8
10
```

We use `if __name__ == "__main__"` block to prevent (certain) code from being run when the module is imported. This if-statement is used to test the module functions, but also give the possibility of using a script as imported module or main program. We use this if statement to test that functions work correctly before importing the module in another python script.

8.1.2 Other special variables and the doc page of a module

Every python script and so also a module has special variables such as `__name__`, `__doc__`, `__file__`. These special variables have the double underscore as a part of their names. Python assigns to each script (and so to a module) these special variables.

At the IPython console type:

```
import geo
geo.__doc__ # variable __doc__ stores whatever you write within triple
quotes at the beginning of the script
This module contains functions to calculate the square \and cube of a
number, and a variable.
```

```
geo.__file__ #variable __file__ stores absolute path to script geo
'/Users/mprocop2/geo.py'
```

```
dir(geo)      #list the functions and constant defined in the geo module
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
 '__name__', '__package__', '__spec__', 'a', 'b', 'cube', 'square']
```

If you enter the help page, you will see that each function or constant you defined in geo.py is listed in the documentation page of geo. The description of each function is whatever you write in the docstring of each function in geo.py

```
help(geo)     #enter the documentation page of the geo module
Help on module geo:

NAME
    geo

DESCRIPTION
    This module contains functions to calculate the square
    and cube of a number, and some variables.

FUNCTIONS
    cube(n)
        calculates cube of a number

    square(n)
        calculates square of a number

DATA
    a = 10
    b = 5

FILE
    /Users/mprocop2/geo.py
```

8.2 Modules can be used in comprehension.

We want to make a list L1 of cubes out of list L, and we can use list comprehension, where the expression is the cube function from the geo module.

```
import geo

L=[2,6,8,9]

L1=[geo.cube(i) for i in L]
print(L1)
[8, 216, 512, 729]
```

Built-in Python modules can be used within comprehensions.

```
import math
```

```
L=[2,6,8,9]
De={i:math.exp(i) for i in L}
print(De)
{2: 7.38905609893065, 6: 403.4287934927351, 8: 2980.9579870417283, 9:
8103.083927575384}
```

Chapter 9. Reading and writing of files

Python is a useful programming language for processing data. The language has several built-in functions that make it easy to read, write, and manipulate data files.

9.1 Reading in data

To read data from a file you should do the following steps:

- Open the file object
- Read the contents and assign them to a variable
- Close the file object

9.1.1 Open File for Reading in Python

The first step to read a file in Python is to open the file you want to read. You can do so by using the *open()* function and specifying the path to the file.

Python's built-in *open()* function returns a file object, which you can store into a variable.

```
fileObject = open(filename, accessmode)
```

filename: a string that contains the pathname of the file that you want to access.

Access mode: determines the mode in which the file has to be opened, i.e., read, write, append, etc.

```
r      Opens a file for reading only. This is the default mode.
r+     Opens a file for both reading and writing.
```

The *r* flag at the end of the *open()* function tells Python that we only want to read in the data. We could change this flag if we wanted, for example, to read and write in the file at the same time.

Let's use the data file `SW.dat` – find the data file on Canvas. You can see the contents of a file by using the bash `cat` command at the IPython console:

```
201,AZ,1.602087638
401,CA,1.232179752
505,CO,1.338985021
2604,NV,1.724225207
2907,NM,1.70802772
4201,UT,1.385726584
```

At the IPython console write

```
r1=open("SW.dat",'r')
type(r1)
_io.TextIOWrapper
```

```
dir(r1) #see all the methods you can apply to a file object
```

After we open the file, we can read in data. There are three methods to read data from a file:

readlines(): Returns a list of lines (lines are stored as string type) of a file

readline(): Returns the next line (string type) of a file

read(): Returns the the entire contents of a file as a string.

In this course we will focus on the *readlines()* method

9.1.2 Reading in data with the readlines() method

Now let's make a script called **SW.py**. We will use the *readlines()* method to read in data.

```
obj=open('filename','r')
listname=obj.readlines()
obj.close()
```

```
r1=open("SW.dat",'r') #open file in r mode and store the file object in r1
L=r1.readlines() #read in all the lines of the file and store data in list L
r1.close() # close the file object
```

```
print(L)
print(type(L))
print(L[0])
print(type(L[0]))
```

Run the script

The *readlines()* method returns a list containing each line in the file as a list item.

```
['201,AZ,1.602087638\n', '401,CA,1.232179752\n', '505,CO,1.338985021\n',
'2604,NV,1.724225207\n', '2907,NM,1.70802772\n', '4201,UT,1.385726584\n']
<class 'list'>
201,AZ,1.602087638
<class 'str'>
```

Notice the print function removes the quotes of the string '201,AZ,1.602087638\n' and also the '\n' which is a new line character. The empty line in the output is coming from the print executing the '\n'.

9.1.3 Access the fields

Now let's see how we can access fields of a data set. Let's loop over the list storing the data, L, and print each item (line) to screen – modify the script to:

```
r1=open("SW.dat",'r') # open in r mode and store the file object in r1
L=r1.readlines() #read in all the lines of the file and store data in list L
r1.close() # close the file object

print(L)

for line in L:
    print(line)
['201,AZ,1.602087638\n', '401,CA,1.232179752\n', '505,CO,1.338985021\n',
'2604,NV,1.724225207\n', '2907,NM,1.70802772\n', '4201,UT,1.385726584\n']
201,AZ,1.602087638

401,CA,1.232179752

505,CO,1.338985021

2604,NV,1.724225207

2907,NM,1.70802772

4201,UT,1.385726584
```

Each item (line) of the list (L) is a string. The empty line is coming from the execution of \n at the end of each string. We can use the split() method to convert the string item (line) into a list (we split each line in fields). In this case since the field separator of the data file is a comma character and so we use split based on comma.

Modify the script to:

```
r1=open("SW.dat",'r') # open in r mode and store the file object in r1
L=r1.readlines() # read in all the lines of the file and store data in
list L
r1.close() # close the file object

print(L)

for line in L:
    print(line.split(','))
['201,AZ,1.602087638\n', '401,CA,1.232179752\n', '505,CO,1.338985021\n',
'2604,NV,1.724225207\n', '2907,NM,1.70802772\n', '4201,UT,1.385726584\n']
['201', 'AZ', '1.602087638\n']
['401', 'CA', '1.232179752\n']
['505', 'CO', '1.338985021\n']
```

```
[ '2604', 'NV', '1.724225207\n' ]
[ '2907', 'NM', '1.70802772\n' ]
[ '4201', 'UT', '1.385726584\n' ]
```

Now we have each line in a list, with the fields being the items. Notice that the fields are still string data type. For the second field it is ok because it contains text, but the first and second fields contain numbers, and we should convert into numeric types, before performing arithmetic calculations.

```
for line in listname:
    line.split(sep)[index] #index selects the field based on the field
separator sep
```

Modify the script to access and print the 3rd field

```
r1=open("SW.dat",'r') # open in r mode and store the file object in r1
L=r1.readlines()      # read in all the lines of the file and store data in
list L
r1.close()            # close the file object

for line in L:
    print(line.split(',')[2]) #access the 3rd field
1.602087638

1.232179752

1.338985021

1.724225207

1.70802772

1.385726584
```

Try to modify the script to calculate the average temperature anomaly (3rd field) – keep in mind that the 3rd field is still a string type. The print function strips the quotes and makes a new line when executing the '\n'.

9.1.4 Storing fields in lists or dictionaries

Based on what the initial data that we are reading in looks like, one can decide to store it in different formats. For example, one can store the data in a list or in lists, or one can store the data in a dictionary. Here are some examples of how to store the data from the file test.dat.

Example 1. Storing data in lists

We have seen how the split() method can be used to convert a string to a list. First remember that each item in the list L is a string corresponding to one line of the original file. You need to convert fields into the correct data type. In this case, for the second field it is ok because it contains text, but the first and second fields contain numbers, and we should convert into numeric types.

Individual fields of file SW.dat could be stored in 3 lists, L1, L2, and L3.

Using loops

```
r1=open("SW.dat",'r') # open the file in r mode and store the file object in
r1
L=r1.readlines()      # read in all the lines of the file and store data in
list L
r1.close()            # close the file object

L1=[]
L2=[]
L3=[]

for line in L:
    L1.append(int(line.split(',')[0]))    # 1st field, convert to integer
    L2.append(line.split(',')[1])        # 2nd field contains text
    L3.append(float(line.split(',')[2]))  # 3rd field, convert into float

print(L1)
print(L2)
print(L3)
[201, 401, 505, 2604, 2907, 4201]
['AZ', 'CA', 'CO', 'NV', 'NM', 'UT']
[1.602087638, 1.232179752, 1.338985021, 1.724225207, 1.70802772, 1.385726584]
```

Using list comprehensions

```
r1=open("SW.dat",'r') # open the file in r mode and store the file object in
r1
L=r1.readlines()      # read in all the lines of the file and store data in
list L
r1.close()            # close the file object

L1=[int(line.split(',')[0]) for line in L]    #1st field, convert to integer
L2=[line.split(',')[1] for line in L]        #2nd field contains text
L3=[float(line.split(',')[2]) for line in L]  #3rd field, convert into float

print(L1)
print(L2)
print(L3)
[201, 401, 505, 2604, 2907, 4201]
['AZ', 'CA', 'CO', 'NV', 'NM', 'UT']
[1.602087638, 1.232179752, 1.338985021, 1.724225207, 1.70802772, 1.385726584]
```

Example 2. Storing data in a dictionary

Let's store some data in a dictionary where the 1st field of file SW.dat is a key, and the 3rd field is the value.

Using loops

```
r1=open("SW.dat",'r') # open the file in r mode and store the file object in
r1
L=r1.readlines()      # read in all the lines of the file and store data in
list L
r1.close()            # close the file object

D={}
for line in L:
    k=int(line.split(',')[0])
    v=float(line.split(',')[2])
    D[k]=v

print(D)
{201: 1.602087638, 401: 1.232179752, 505: 1.338985021, 2604: 1.724225207,
2907: 1.70802772, 4201: 1.385726584}
```

Using dictionary comprehensions

```
r1=open("SW.dat",'r') #open the file in r mode and store the file object in r
L=r1.readlines()      #read in all the lines of the file and store data in
list L
r1.close()            #close the file object

D={line.split(',')[0]:float(line.split(',')[2]) for line in L}
print(D)
{'201': 1.602087638, '401': 1.232179752, '505': 1.338985021, '2604':
1.724225207, '2907': 1.70802772, '4201': 1.385726584}
```

9.1.5 Read line-by-line with the `readline()` method

The *readline()* function can be useful if you want to read a file line by line. This method reads a single line from the file and returns a **string**, subsequent calls to `.readline()` will return successive lines.

```
r1=open("SW.dat",'r')
s1=r1.readline()      #reads the file into memory line by line
print(r1)             #s1 stores the first line as a string type
s2=r1.readline()      #read the second line
print(s2)             #s2 stores the second line as a string type
```

Now try to read in data with the *read()* method.

9.2 Writing data into a file

To write data into a file, you need to follow these steps

- Open the file object
- Write/Append the content into the file
- Close the file object

Same as for reading in data, to write data into a file, you first need to open a file by using the Python's built-in `open()` function. Then you can use different methods to write data into a file. In this course we will use the `write()` method

```
fileObject = open(filename, 'w')
fileObject.write(string)
fileObject.close()
```

filename: is a string that contains the name of the file that you want to access.

```
w    Opens a file for writing only. Overwrites the file if the file exists.
      If the file does not exist, creates a new file for writing.

a    Opens a file for appending. If the file does not exist,
      it creates a new file for writing.
```

The `write()` method writes any **string** to an open file. The `write()` method does not add a newline character ('\n') to the end of the string.

In a script called my-write.py:

```
my_str='abcd'
out2=open("out_str.dat", 'w')
out2.write(my_str+'\n') # writes four characters: a,b,c,d + a new line
character
out2.close()
```

Run the script – Now look at the contents of the file `out_str.dat` with the `cat` command at the IPython console. Remember that the `write` method takes in a string type, and you would need to perform type conversion in some cases, like this one

```
f2 = open('file2', 'w')
for i in range(1,5):
    f2.write(str(i)+"\n")
f2.close()
```

9.2.1 Formatting operator % in the write method.

Since the `write` method takes a string as argument, you can use the string formatting operator `%` to write formatted text into a file.

To write formatted output use same syntax that you would use to print formatted output with the `print` function. But in the `write` method you should specify `\n` to make a new line.

```
f3=open('file3', 'w')
for i in range(1,5):
    f3.write("This is integer %d\n" %i)
```

```
f3.close()
```

9.2.2 The close() method

The method **close()** closes the opened file. A closed file cannot be read or written any more.

You *always* need to close your files after you're done writing to them. During the I/O process, data is held in a temporary location before being written to the file. Python doesn't write data to the file—until it's sure you're done writing. One way to do this is to close the file. If you write to a file without closing, the data won't make it to the target file.

There is a way in Python to avoid having to close the files. This is done as it follows:

```
with open("file", "mode") as variable:
    # Read or write to the file
```

For example:

```
with open('test.dat','r') as myfile:
    dat2=myfile.readlines()
```

or

```
with open('out.dat','w') as fout:
    fout.writelines(my_list)
```

Notes by Maria Procopio

References:

- <https://www.w3schools.com/python/default.asp>
- <https://realpython.com/>
- <https://docs.python.org/3/tutorial/index.html>
- Previous Introduction to Computing Notes by C. Fitch and A. Damjanovic