Lab Manal

# AI201L- Programming for Artificial Intelligence Lab

Prepared by: Zain Ul Abideen
Ghulam Ishaq Khan Institute of Engineering and Technology, Pakistan

| Lab Breakdown | |
|---|---|
| **Lab** | **Contents/Topics** |
| **Lab#01:** | Anaconda Installation. Basic data types, Variables, Mathematical Operators, input/output |
| **Lab#02:** | Lists and list function in python and Tuple |
| **Lab#03:** | Conditional Statements, Loops |
| **Lab#04:** | Dictionary in Python |
| **Lab#05:** | Functions in Python |
| **Lab#06:** | OOP in Python |
| | **Mid-term exam**. |
| **Lab#07:** | OOP Concepts in Python |
| **Lab#08:** | Python Exception handling |
| **Lab#09:** | Python Version Control |
| **Lab#10:** | NumPy |
| **Lab#11:** | Pandas |
| **Lab#12:** | Matplotlib |
| | **OEL** |
| | **Final Exam** |

# I. Table of Contents

## II. Lab Outline

| **AI201L Programming for AI Lab (1 CH)** |
|---|
| **Pre-Requisite:** NO<br>Instructor: **Zain Ul Abideen**<br>Email: *zain-ul-abideen@giki.edu.pk*<br>Office: S-04 New Academic Block (AI department). Ext. 0000<br>Office Hours: 9:00 pm ~ 12:30 pm |

**Lab Introduction**

This course will introduce you to the field of Artificial Intelligence and the fundamentals of python programming. AI201 is specifically designed for students with prior programming experience and touches upon a variety of fundamental topics to advance topics and some necessary libraries which used in AI domain. This course uses Python programming platform to understand the principals of basic computer language to advance. We do this by demonstrating Python primitive data types, relational operators, logical operators, control statements, iterative statements, list, tuple, dictionary, file handling, exception handling, OOP concepts, git, NumPy, Pandas and Matplotlib. By the end of the course, you will understand the basics to advance of Python programming language. The principles you learn here will be developed further as you progress through the Artificial Intelligence discipline.

**Mapping of CLOs and PLOs**

| • Sr. No | • Course Learning Outcomes+ | • PLOs* | • Blooms Taxonomy |
|---|---|---|---|
| • CLO_1 | • Utilize the basic techniques of python programming language. | • P L O 1 | • P2 (Set) |
| • CLO_2 | Implement programming structures to design solutions for the given problems. | • PLO 1 | • P3 (Guided Response) |
| • CLO_3 | Able to utilize python library in python program | • PLO 5 | • P3 (Guided Response) |
| • | • +Please add the prefix "Upon successful completion of this course, the student will be able to"<br>• *PLOs are for BS (CE) only | | |

**CLO Assessment Mechanism (Tentative)**

| **Assessment tools** | **CLO_1** | **CLO_2** | **CLO_3** |
|---|---|---|---|
| **Lab Performance** | **50%** | **50%** | **30%** |
| **Project** | - | **15%** | **40%** |
| **Midterm Exam** | **20%** | - | |
| **Final Exam** | **30%** | **35%** | **30%** |

**Overall Grading Policy (Tentative)**

| **Assessment Items** | **Percentage** |
|---|---|
| **Lab Performance** | **35%** |
| **Midterm Exam** | **25%** |
| **Project** | **5%** |
| **Final Exam** | **35%** |

**Text and Reference Books**

**Text books:**

- Python Crash Course, 2nd Edition, A Hands-On, Project-Based Introduction to Programming, by Eric Matthes, No Starch Press, 2019
- Lab Manual for AI201L

# III. System Requirements

**HARDWARE REQUIREMENT**

- Core i3 or Above
- 32- or 64-bit computer
- 4 GB RAM
- 3 GB HDD

**SOFTWARE REQUIREMENT**

- Anaconda, PyCharm, Visual Studio, Jupyter notebook, Jupyterlab, gitbash

**Anaconda System Requirements link:**

https://docs.conda.io/projects/conda/en/latest/user-guide/install/index.html

## IV. Evaluation Rubrics

| **Weekly Evaluation** | |
|---|---|
| **Rubrics** | **Marks (10)** |
| Code writing | 1 |
| **Error/exception Handling** | 1 |
| **Code Comments / Necessary Documentation** | 3 |
| **Correctness** | 5 |

**Detailed Rubric of weekly Evaluation:**

**Code writing (No credit, if not relevant implementation):**

* Meaningful/relevant variable/function names: 1

**Error/exception Handling:**

* Runs without exception: 0.5
* Displays proper messages/operations clearly: 0.5

**Code Comments / Viva:**

* Clarify meaning where needed: 1
* Knowledge about the topic: 2

**Correctness:**

* Accurate methodology: 1
* All tasks are done: 3
* Passes all test cases: 1

## V. Instruction For Students

- Students are expected to install lab software on their personal computers
- Students are expected to read respective lab contents before attending it
- Since the contents of a lab are covered in class before practicing in the lab hence, students are expected to read, understand and run all examples and sample problems before attending the lab
- Reach the lab on time. Late comers shall not be allowed to attend the lab
- Students need to maintain 100% attendance in lab if not a strict action will be taken.
- Wear your student card in the lab
- Mobile phones, USBs, and other electronic devices are strictly prohibited in the lab
- In case of hardware/power issues, immediately inform lab attendant and do not attempt to fix it by yourself
- In case of queries during lab task implementation, raise your hand, inform your instructor and wait for your turn

# 1. Lab#01 - Introduction to Python

## Learning Objectives

- Installation of Anaconda Software
- Environment of Jupyter and Run the Program in Jupyter
- To learn the basic data types and variables
- To learn the mathematical operators
- To learn the input and output

## Outcomes:

- Students should be able to install the Anaconda Software
- Students should be able Run their program in Jupyter
- Students should be able to know Python variables their types and arithmetic operators.
- Students should be able to use input and print methods in the programs.

## 1.1 Anaconda Software and WHY

Anaconda is a distribution of packages built for data science. It comes with conda, a package, and environment manager. Anaconda is a distribution of packages built for data science. It comes with conda, a package, and environment manager. For problem solvers, We recommend installing and using the Anaconda distribution of Python. This section details the installation of the Anaconda distribution of Python on Windows 10. We think the Anaconda distribution of Python is the best option for problem solvers who want to use Python. Anaconda is free (although the download is large which can take time) and can be installed on school or work computers where you don't have administrator access or the ability to install new programs. Anaconda comes bundled with about 600 packages pre-installed including NumPy, Matplotlib and SymPy. These three packages are very useful for problem solvers and will be discussed in subsequent chapters. We usually used conda to create environments for isolating our projects that use different versions of Python and/or different version of packages. We also use it to install, uninstall, and update packages in our project environments. When you download Anaconda first time it comes with conda, Python, and over 150 scientific packages and their dependencies. Anaconda is a fairly large download (~500 MB) because it comes with the most common data science packages in Python, for people who are conservative about disk space, there is also Miniconda, a smaller distribution that includes only conda and Python. You can still install any of the available packages with conda that comes by default with the standard version

## 1.2 Installation of Anaconda

Follow the steps below to install the Anaconda distribution of Python on Windows.

Steps:

- Visit Anaconda.com/downloads
- Select Windows
- Download the .exe installer
- Open and run the .exe installer
- Open the Anaconda Navigator and run some Python code

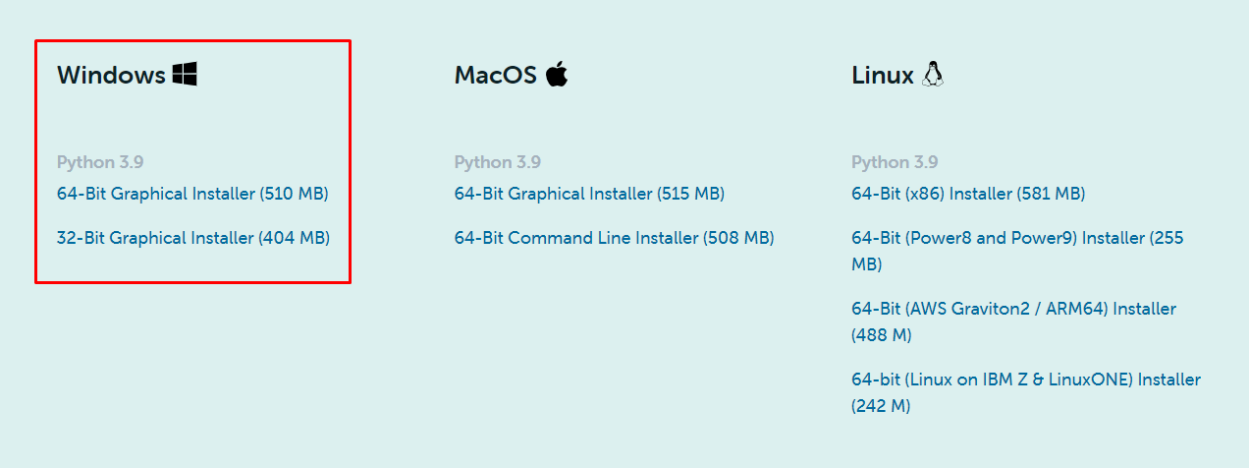
### 1.2.1 Visit Anaconda.com/downloads

Go to the following link: https://www.anaconda.com/products/individual

Go end of the page and the Anaconda Downloads Page will look something like this:

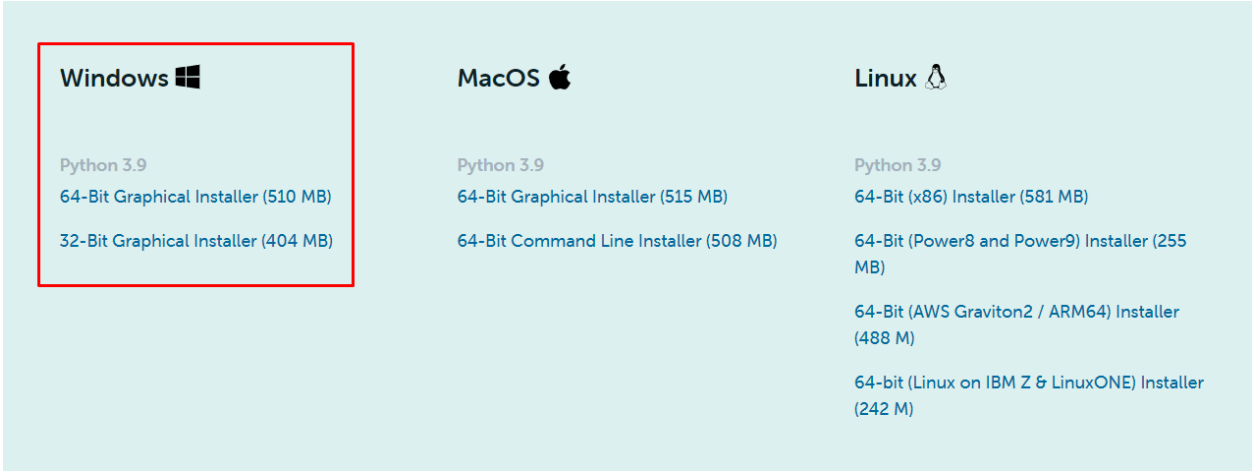

### 1.2.2 Select Windows

Select Windows where the three operating systems are listed.

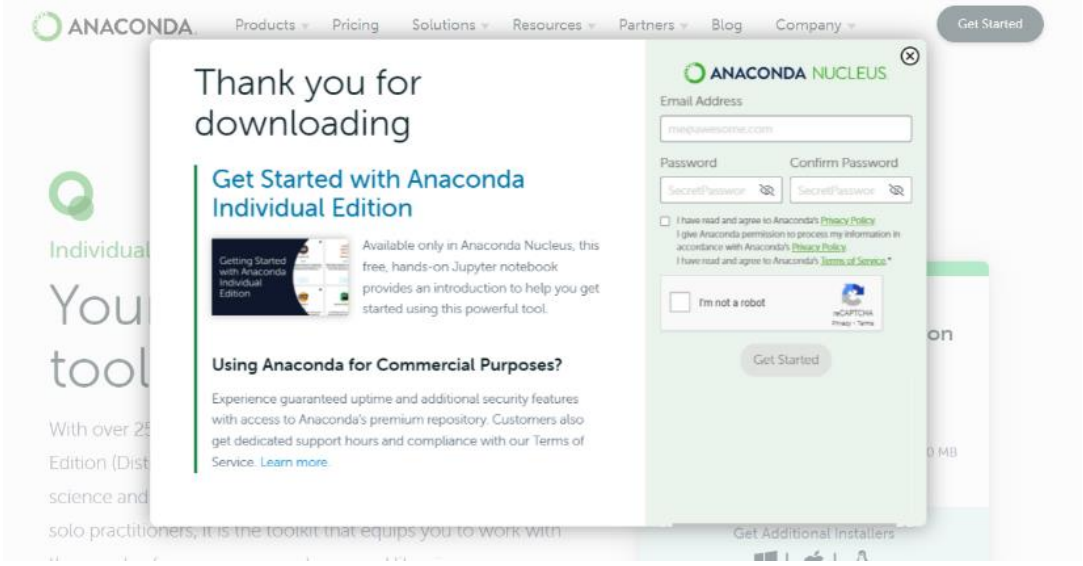



### 1.2.3 Downloads

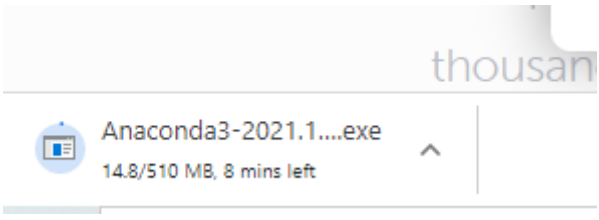
Download the most recent Python 3 release. If you are unsure if your computer is running a 64-bit or 32-bit version of Windows, select 64-bit as 64-bit Windows is most common.




You may be prompted to enter your email. You can still download Anaconda if you click [No Thanks] and don't enter your Work Email address.

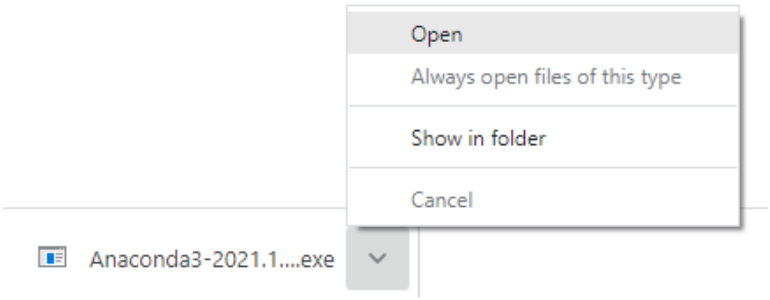



The download is quite large (over 500 MB) so it may take a while to for Anaconda to download.



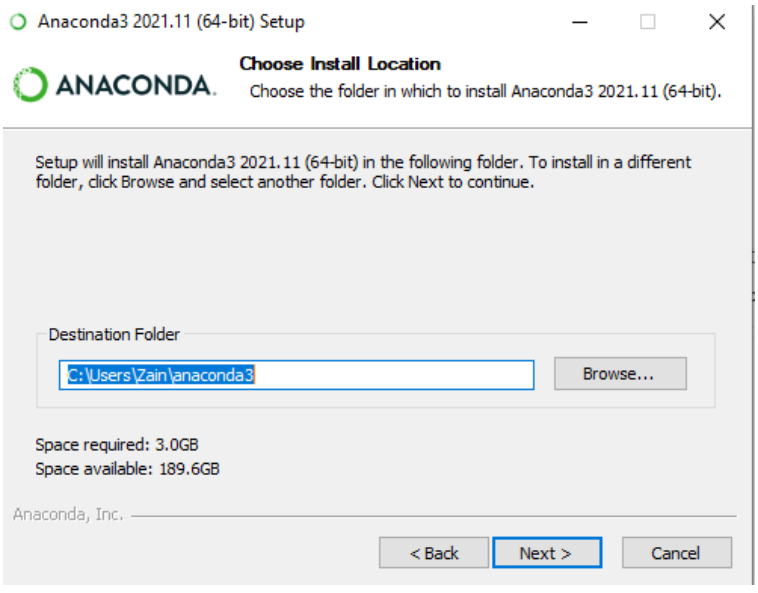

### 1.2.4 Open and run the installer

Once the download completes, open and run the .exe installer.

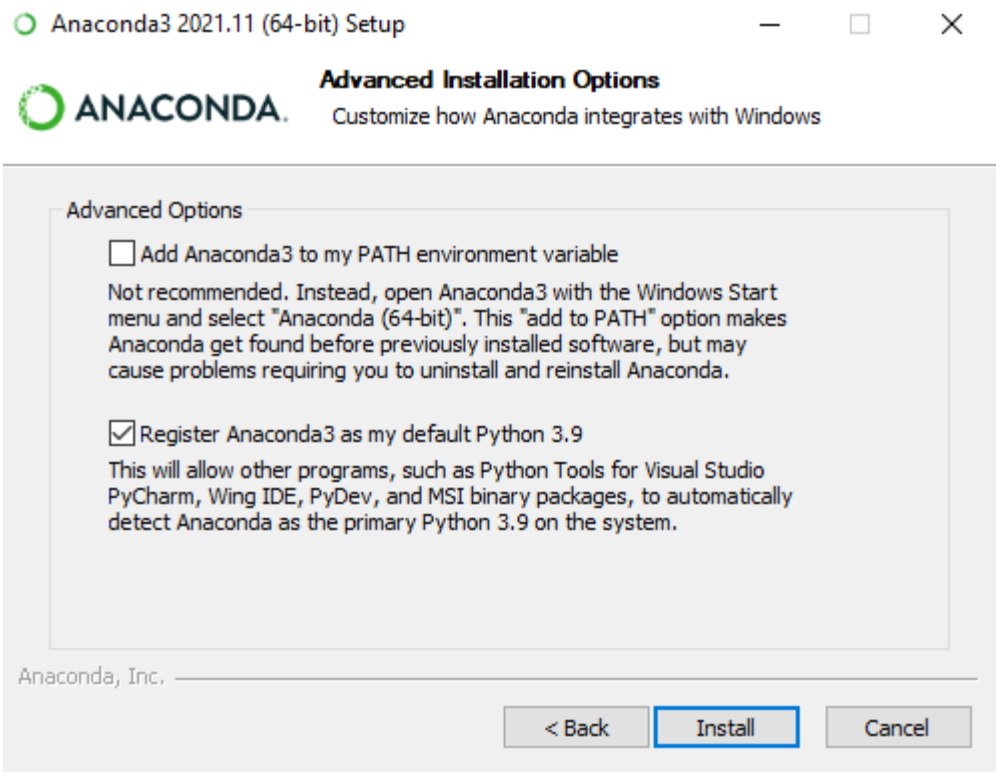



At the beginning of the install, you need to click Next to confirm the installation.


Then agree to the license.

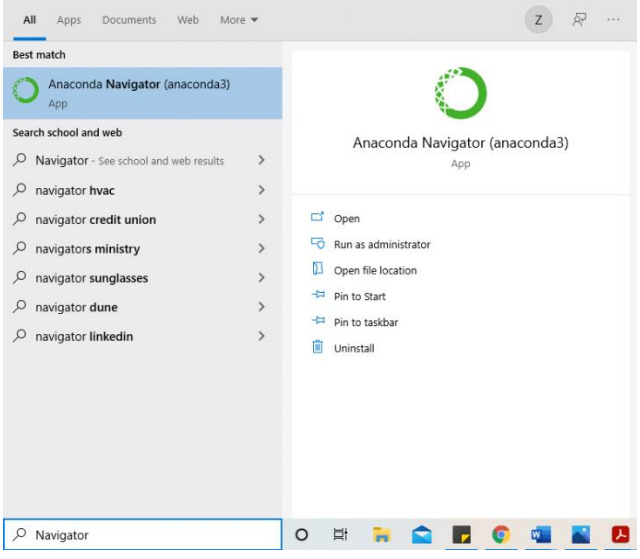


At the Advanced Installation Options screen, I recommend that you do not check "Add Anaconda to my PATH environment variable".

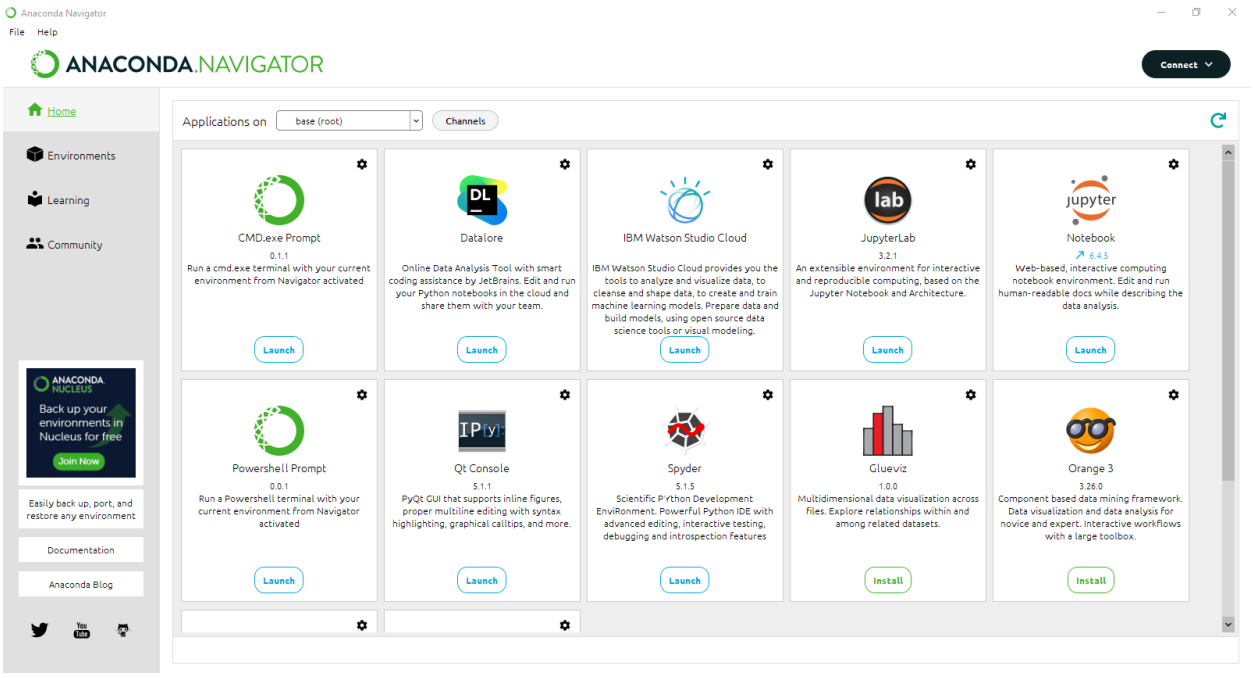



### 1.2.5 Open the Anaconda Navigator from the Windows start menu

After the installation of Anaconda is complete, you can go to the Windows start menu and search the Navigator and select the Anaconda Navigator.

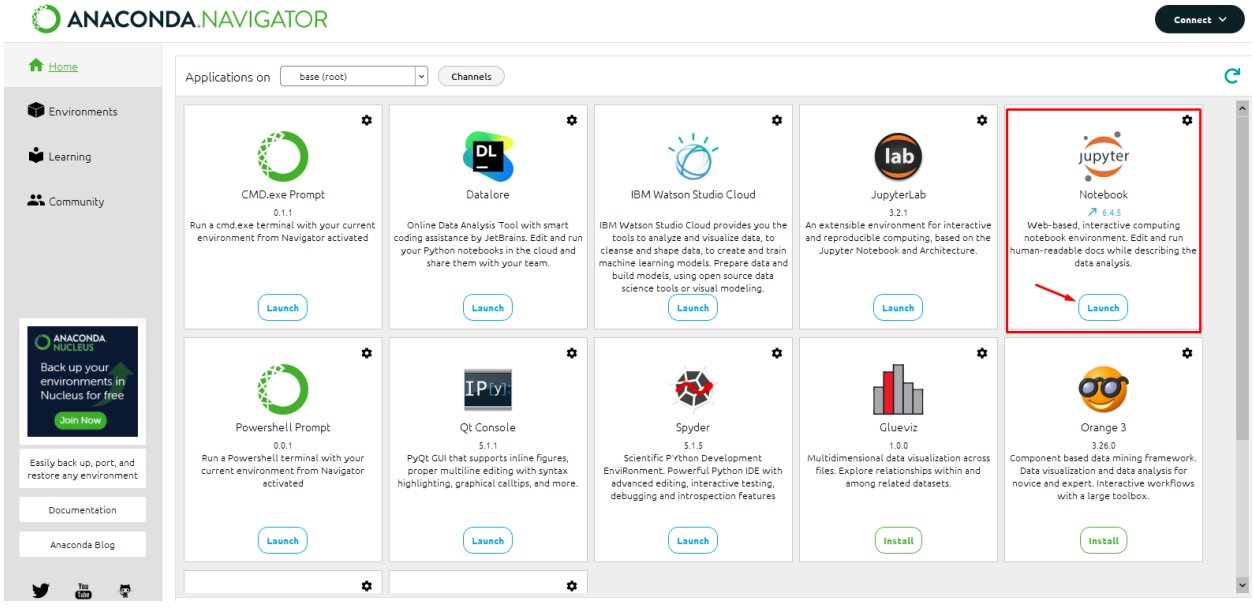


After installation the Anaconda Navigator looks like:

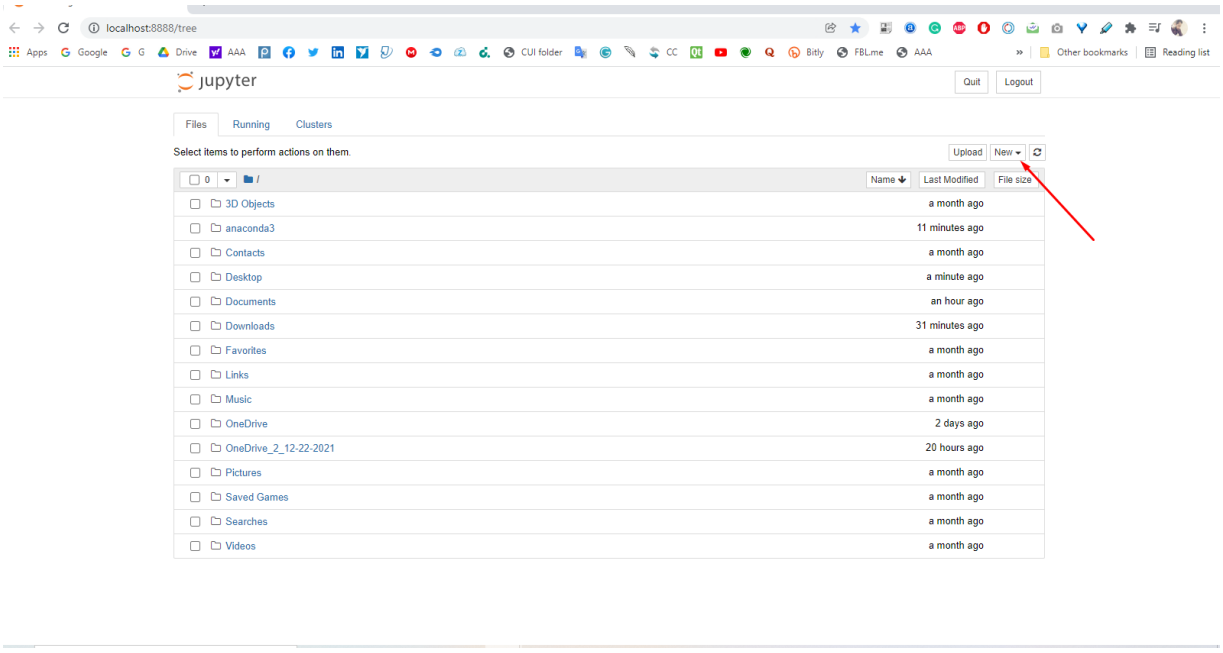


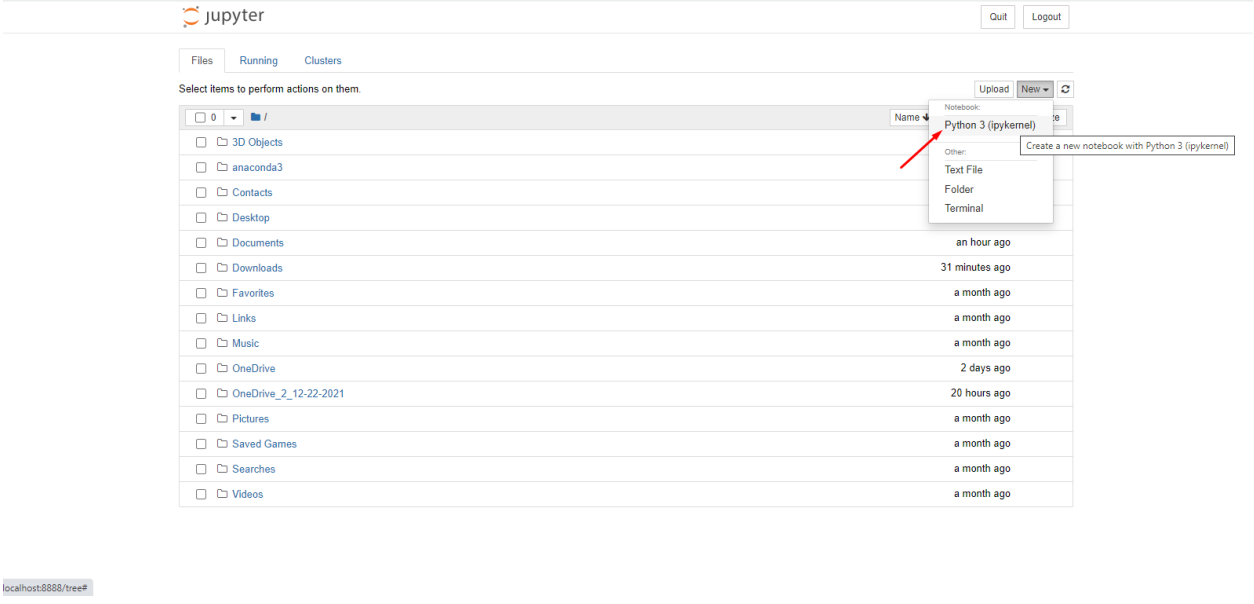

## 1.3 Hello Word in Jupyter notebook

Open the Notebook from Anaconda Navigator and launch the Jupyter notebook.



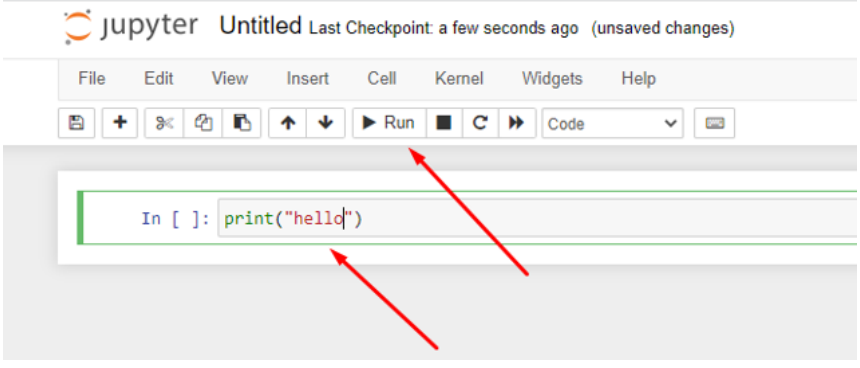

When Jupyter notebook in load in any browser. Then click on new and create a notebook file by clicking on python3.9 (your python version)

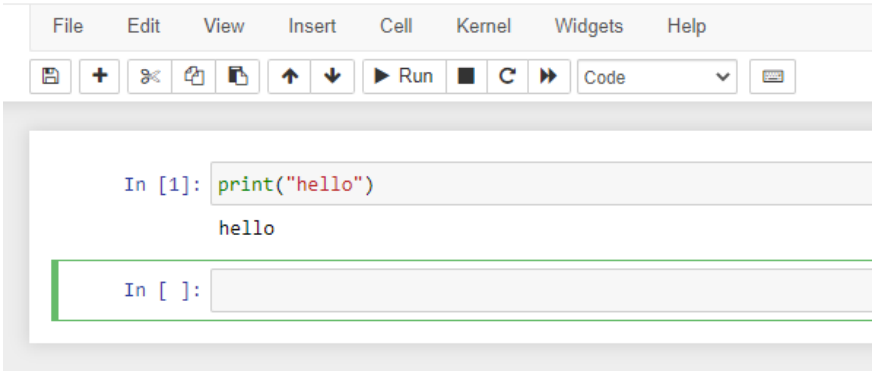



localhost:8888/tree#

After creating the notebook, write print ("hello") and click on run and check the output.






## 1.4 Basic Data Types in Python

You'll learn about several basic numeric, string, and Boolean types that are built into Python. By the end of this lab, you'll be familiar with what objects of these types look like, and how to represent them.

You'll also get an overview of Python's built-in functions. These are pre-written chunks of code you can call to do useful things.

### 1.4.1 Integers

In Python 3, there is effectively no limit to how long an integer value can be. Of course, it is constrained by the amount of memory your system has, as are all things, but beyond that an integer can be as long as you need it to be:

```
In [2]: print(123123123123123123123123123123123123123123123123 + 1)
123123123123123123123123123123123123123123123124
```

Python interprets a sequence of decimal digits without any prefix to be a decimal number:

```
In [3]: print(10)
10
```

The following strings can be prepended to an integer value to indicate a base other than 10:

| Prefix | Interpretation | Base |
|---|---|---|
| `0b` (zero + lowercase letter `'b'`)<br>`0B` (zero + uppercase letter `'B'`) | Binary | 2 |
| `0o` (zero + lowercase letter `'o'`)<br>`0O` (zero + uppercase letter `'O'`) | Octal | 8 |
| `0x` (zero + lowercase letter `'x'`)<br>`0X` (zero + uppercase letter `'X'`) | Hexadecimal | 16 |

**For example:**

```
In [4]: print(0o10)
8

In [5]: print(0x10)
16

In [7]: print(0b10)
2
```

For more information on integer values with non-decimal bases, see the following Wikipedia sites: Binary, Octal, and Hexadecimal. The underlying type of a Python integer, irrespective of the base used to specify it, is called int. Use type method to check the type of any variable.

```
In [1]: type(10)
Out[1]: int

In [2]: type(0o10)
Out[2]: int

In [3]: type(0x10)
Out[3]: int

In [ ]:
```

### 1.4.2 Floating-Point Numbers

The float type in Python designates a floating-point number. float values are specified with a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation:

```
In [4]: 4.2
Out[4]: 4.2

In [5]: type(4.2)
Out[5]: float

In [6]: 4.
Out[6]: 4.0

In [7]: .2
Out[7]: 0.2

In [8]: .4e7
Out[8]: 4000000.0

In [9]: type(.4e7)
Out[9]: float

In [10]: 4.2e-4
Out[10]: 0.00042
```

### 1.4.3 Complex Numbers

Complex numbers are specified as <real part>+<imaginary part>j. For example:

```
In [11]: 2+3j
Out[11]: (2+3j)

In [12]: type(2+3j)
Out[12]: complex
```

### 1.4.4 Strings

Strings are sequences of character data. The string type in Python is called str. String literals may be delimited using either single or double quotes. All the characters between the opening delimiter and matching closing delimiter are part of the string:

```
In [13]: print("im a string ")
         im a string

In [14]: type("im a string ")
Out[14]: str
```

What if you want to include a quote character as part of the string itself? Your first impulse might be to try something like this:

```
In [15]: print('this string contain a single qoute(')charachter.')
           File "<ipython-input-15-519429805632>", line 1
             print('this string contain a single qoute(')charachter.')
                                                        ^
         SyntaxError: invalid syntax
```

As you can see, that doesn't work so well. The string in this example opens with a single quote, so Python assumes the next single quote, the one in parentheses which was intended to be part of the string, is the closing delimiter. The final single quote is then a stray and causes the syntax error shown.

If you want to include either type of quote character within the string, the simplest way is to delimit the string with the other type. If a string is to contain a single quote, delimit it with double quotes and vice versa:

```
In [16]: print("this string contain a single qoute(')charachter.")
         this string contain a single qoute(')charachter.

In [17]: print('this string contain a single qoute(")charachter.')
         this string contain a single qoute(")charachter.
```

### 1.4.4.1 Escape Sequences in Strings

Sometimes, you want Python to interpret a character or sequence of characters within a string differently. This may occur in one of two ways:

- You may want to suppress the special interpretation that certain characters are usually given within a string.
- You may want to apply special interpretation to characters in a string which would normally be taken literally.

You can accomplish this using a backslash (\) character. A backslash character in a string indicates that one or more characters that follow it should be treated specially. (This is referred to as an escape sequence, because the backslash causes the subsequent character sequence to "escape" its usual meaning.)

Let's see how this works.

#### 1.4.4.1.1 Suppressing Special Character Meaning

You have already seen the problems you can come up against when you try to include quote characters in a string. If a string is delimited by single quotes, you can't directly specify a single quote character as part of the string because, for that string, the single quote has special meaning—it terminates the string:

```
In [15]: print('this string contain a single qoute(')charachter.')
  File "<ipython-input-15-519429805632>", line 1
    print('this string contain a single qoute(')charachter.')
                                                ^
SyntaxError: invalid syntax
```

Specifying a backslash in front of the quote character, in a string "escape" it and causes Python to suppress its usual special meaning. It is then interpreted simply as a literal single quote character:

```
In [18]: print('this string contain a single qoute(\')charachter.')
this string contain a single qoute(')charachter.
```

The same works in a string delimited by double quotes as well:

```
In [19]: print("this string contain a single qoute(\")charachter.")
this string contain a single qoute(")charachter.
```

The following is a table of escape sequences which cause Python to suppress the usual special interpretation of a character in a string:

| Escape Sequence | Usual Interpretation of Character(s) After Backslash | "Escaped" Interpretation |
|---|---|---|
| \' | Terminates string with single quote opening delimiter | Literal single quote (') character |
| \" | Terminates string with double quote opening delimiter | Literal double quote (") character |
| \<newline> | Terminates input line | Newline is ignored |
| \\ | Introduces escape sequence | Literal backslash (\) character |

Ordinarily, a newline character terminates line input. So, pressing Shift + Enter (jupyter) in the middle of a string will cause Python to think it is incomplete:

```
In [22]: print('a
           File "<ipython-input-22-785b7114aeab>", line 1
             print('a
                   ^
         SyntaxError: EOL while scanning string literal
```

To break up a string over more than one line, include a backslash before each newline, and the newlines will be ignored:

```
In [23]: print('a\
         b\
         c')
         abc
```

To include a literal backslash in a string, escape it with a backslash:

```
In [24]: print('food\\bar')
         food\bar
```

#### 1.4.4.1.2 Applying Special Meaning to Characters

Next, suppose you need to create a string that contains a tab character in it. Some text editors may allow you to insert a tab character directly into your code. But many programmers consider that poor practice, for several reasons:

- The computer can distinguish between a tab character and a sequence of space characters, but you can't. To a human reading the code, tab and space characters are visually indistinguishable.
- Some text editors are configured to automatically eliminate tab characters by expanding them to the appropriate number of spaces.
- Some Python REPL environments will not insert tabs into code.

In Python (and almost all other common computer languages), a tab character can be specified by the escape sequence \t:

```
In [25]: print('food\tbar')
         food    bar
```

The escape sequence \t causes the t character to lose its usual meaning, that of a literal t. Instead, the combination is interpreted as a tab character.

Here is a list of escape sequences that cause Python to apply special meaning instead of interpreting literally:

| Escape Sequence | "Escaped" Interpretation |
|---|---|
| \a | ASCII Bell (BEL) character |
| \b | ASCII Backspace (BS) character |
| \f | ASCII Formfeed (FF) character |
| \n | ASCII Linefeed (LF) character |
| \N{<name>} | Character from Unicode database with given <name> |
| \r | ASCII Carriage Return (CR) character |
| \t | ASCII Horizontal Tab (TAB) character |
| \uxxxx | Unicode character with 16-bit hex value xxxx |
| \Uxxxxxxxx | Unicode character with 32-bit hex value xxxxxxxx |
| \v | ASCII Vertical Tab (VT) character |
| \ooo | Character with octal value ooo |
| \xhh | Character with hex value hh |

**Examples:**

```
In [26]: print('a\tb')
         a       b

In [27]: print("a\141\x61")
         aaa

In [28]: print('\u2192\N{rightwards arrow}')
         →→

In [ ]:
```

This type of escape sequence is typically used to insert characters that are not readily generated from the keyboard or are not easily readable or printable.

### 1.4.4.2 Raw Strings

A raw string literal is preceded by r or R, which specifies that escape sequences in the associated string are not translated. The backslash character is left in the string:

```
In [29]: print('foo\nbar')
         foo
         bar

In [30]: print(r'foo\nbar')
         foo\nbar

In [31]: print('foo\\nbar')
         foo\nbar

In [32]: print(R'foo\nbar')
         foo\nbar
```

### 1.4.4.3 Triple-Quoted Strings

There is yet another way of delimiting strings in Python. Triple-quoted strings are delimited by matching groups of three single quotes or three double quotes. Escape sequences still work in triple-quoted strings, but single quotes, double quotes, and newlines can be included without escaping them. This provides a convenient way to create a string with both single and double quotes in it:

```
In [33]: print('''this string has single(')and double (") qoutes.''')
         this string has single(')and double (") qoutes.
```

Because newlines can be included without escaping them, this also allows for multiline strings:

```
In [35]: print("""this is
                string that
                spans across several
                lines""")
         this is
                string that
                spans across several
                lines
```

### 1.4.5 Boolean Type

the Python Boolean type has only two possible values:

- True
- False

No other value will have bool as its type. You can check the type of True and False with the built-in `type()`:

```
In [37]: type(False)
Out[37]: bool

In [38]: type(True)
Out[38]: bool
```

The `type()` of both False and True is bool.

## 1.5 Variables

### 1.5.1 Variable Assignment

Think of a variable as a name attached to a particular object. In Python, variables need not be declared or defined in advance, as is the case in many other programming languages. To create a variable, you just assign it a value and then start using it. Assignment is done with a single equals sign (=):

This is read or interpreted as “n is assigned the value 300.” Once this is done, n can be used in a statement or expression, and its value will be substituted:

```
In [39]: n=300

In [40]: print(n)
         300
```

Just as a literal value can be displayed directly from the jupyter notebook cell in a REPL session without the need for `print()`, so can a variable:

```
In [41]: n
Out[41]: 300
```

Later, if you change the value of n and use it again, the new value will be substituted instead:

```
In [42]: n=100

In [43]: print(n)
         100

In [44]: n
Out[44]: 100

In [ ]:
```

Python also allows chained assignment, which makes it possible to assign the same value to several variables simultaneously:

```
In [45]: a=b=c=300

In [46]: print(a,b,c)
         300 300 300
```

The chained assignment above assigns 300 to the variables a, b, and c simultaneously.

### 1.5.2 Variable Types in Python

In many programming languages, variables are statically typed. That means a variable is initially declared to have a specific data type, and any value assigned to it during its lifetime must always have that type.

Variables in Python are not subject to this restriction. In Python, a variable may be assigned a value of one type and then later re-assigned a value of a different type:

```
In [47]: var=2.35

In [48]: print(var)
         2.35

In [49]: var="now im a string"

In [50]: print(var)
         now im a string
```

### 1.5.3 Object References

What is actually 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.

**Consider this code:**

```
In [51]: print(300)
         300

In [ ]:
```

When presented with the statement print(300), the interpreter does the following:

- Creates an integer object
- Gives it the value 300
- Displays it to the console

You can see that an integer object is created using the built-in `type()` function:

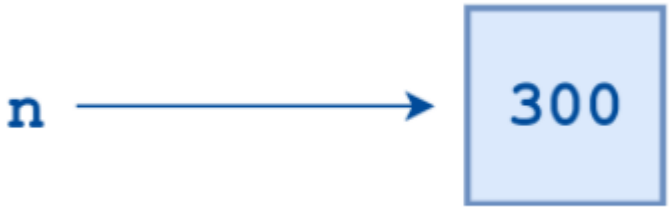
```
In [52]: type(300)
Out[52]: int
```

A Python variable is a symbolic name that is a reference or pointer to an object. Once an object is assigned to a variable, you can refer to the object by that name. But the data itself is still contained within the object.

**For example:**

```
In [53]: n=300

In [ ]:
```

This assignment creates an integer object with the value 300 and assigns the variable n to point to that object.




The following code verifies that n points to an integer object:

```
In [54]: print(n)
         300

In [55]: type(n)
Out[55]: int
```

Now consider the following statement:

```
In [56]: m=n

In [ ]:
```

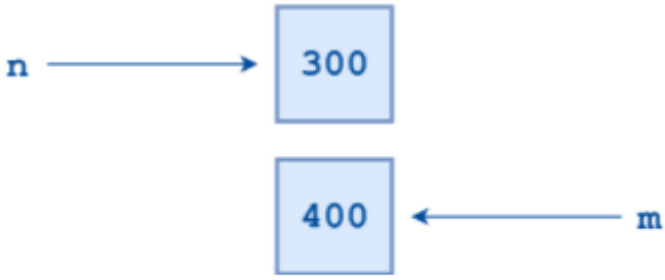
What happens when it is executed? Python does not create another object. It simply creates a new symbolic name or reference, m, which points to the same object that n points to.


Multiple References to a Single Object

Next, suppose you do this:

```
In [57]: m=400
```

Now Python creates a new integer object with the value 400, and m becomes a reference to it.

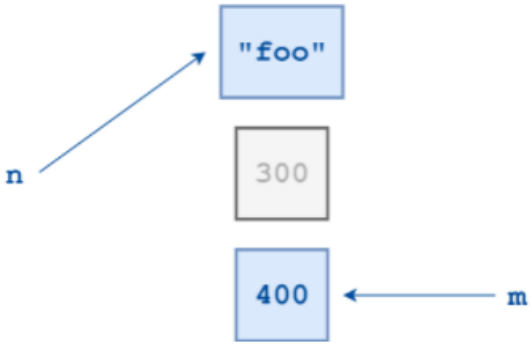



References to Separate Objects

Lastly, suppose this statement is executed next:

```
In [58]: n="foo"
```

Now Python creates a string object with the value "foo" and makes n reference that.




Orphaned Object

There is no longer any reference to the integer object 300. It is orphaned, and there is no way to access it.

An object's life begins when it is created, at which time at least one reference to it is created. During an object's lifetime, additional references to it may be created, as you saw above, and references to it may be deleted as well. An object stays alive, as it were, so long as there is at least one reference to it.

When the number of references to an object drops to zero, it is no longer accessible. At that point, its lifetime is over. Python will eventually notice that it is inaccessible and reclaim the allocated memory so it can be used for something else. In computer lingo, this process is referred to as garbage collection.

### 1.5.4 Object Identity

In Python, every object that is created is given a number that uniquely identifies it. It is guaranteed that no two objects will have the same identifier during any period in which their lifetimes overlap. Once an object's reference count drops to zero and it is garbage collected, as happened to the 300 object above, then its identifying number becomes available and may be used again.

The built-in Python function id() returns an object's integer identifier. Using the id() function, you can verify that two variables indeed point to the same object:

```
[59]: n=300
      m=n

[60]: id(n)

[60]: 2259583284464

[61]: m=400
      id(m)

[61]: 2259583284976
```

After the assignment m = n, m and n both point to the same object, confirmed by the fact that id(m) and id(n) return the same number. Once m is reassigned to 400, m and n point to different objects with different identities.

### 1.5.5 Variable Names

The examples you have seen so far have used short, terse variable names like m and n. But variable names can be more verbose. In fact, it is usually beneficial if they are because it makes the purpose of the variable more evident at first glance.

Officially, variable names in Python can be any length and can consist of uppercase and lowercase letters (A-Z, a-z), digits (0-9), and the underscore character (_). An additional restriction is that, although a variable name can contain digits, the first character of a variable name cannot be a digit.

For example, all of the following are valid variable names:

```
In [63]: name="Bob"
         age=45
         has_W2=True
         print(name,age,has_W2)

         Bob 45 True
```

---

But this one is not, because a variable name can't begin with a digit:

```
In [64]: 1099_filed=False

           File "<ipython-input-64-67c3f2339ef3>", line 1
             1099_filed=False
                  ^
         SyntaxError: invalid decimal literal
```

Note that case is significant. Lowercase and uppercase letters are not the same. Use of the underscore character is significant as well. Each of the following defines a different variable:

```
[67]: age=1
      Age=2
      aGe=3
      a_g_e=4
      _age=5
      age_=6
      _Age_=7
      print(age,Age,aGe,a_g_e,_age,age_,_Age_)

      1 2 3 4 5 6 7
```

There is nothing stopping you from creating two different variables in the same program called age and Age, or for that matter agE. But it is probably ill-advised.

It would certainly be likely to confuse anyone trying to read your code, and even you yourself, after you'd been away from it awhile.

It is worthwhile to give a variable a name that is descriptive enough to make clear what it is being used for.

For example, suppose you are tallying the number of people who have graduated college. You could conceivably choose any of the following:

```
[68]: numberofcollegegreduates=2300
      NUMBEROFCOLLEGEGREDUATES=4000
      numberOfCollegeGreduates=4500
      NumberOfCollegeGreduates=4500
      number_of_college_greduates=3500
      print(numberofcollegegreduates,NUMBEROFCOLLEGEGREDUATES,
            numberOfCollegeGreduates,NumberOfCollegeGreduates,
            number_of_college_greduates)

      2300 4000 4500 4500 3500
```

The most commonly used methods of constructing a multi-word variable name are the last three examples:

- **Camel Case:** Second and subsequent words are capitalized, to make word boundaries easier to see. (Presumably, it struck someone at some point that the capital letters strewn throughout the variable name vaguely resemble camel humps.)

  Example: numberOfCollegeGraduates

- **Pascal Case:** Identical to Camel Case, except the first word is also capitalized.

  Example: NumberOfCollegeGraduates

- **Snake Case:** Words are separated by underscores.

  Example: number_of_college_graduate

## 1.6 Reserved Words (Keywords)

There is one more restriction on identifier names. The Python language reserves a small set of keywords that designate special language functionality. No object can have the same name as a reserved word. Python keywords mention below:

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

You can check python reserve keywords list by importing keyword mention below:

```
[20]: #check list of keywords
      import keyword
      print(keyword.kwlist)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'con
e', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', '
ise', 'return', 'try', 'while', 'with', 'yield']
```

Trying to create a variable with the same name as any reserved word results in an error.

```
In [70]: for=3
```

```
  File "<ipython-input-70-bb96f6012aea>", line 1
    for=3
        ^
SyntaxError: invalid syntax
```

## 1.7 Arithmetic Operations

In Python, operators are special symbols that designate that some sort of computation should be performed. The values that an operator acts on are called operands.

Here is an example:

```
In [71]: a=10
         b=20
         print(a+b)
```

```
30
```

In this case, the + operator adds the operands a and b together. An operand can be either a literal value or a variable that references an object:

```
In [72]: a=10
         b=20
         a+b-5
```

```
Out[72]: 25
```

A sequence of operands and operators, like a + b - 5, is called an expression. Python supports many operators for combining data objects into expressions.

### 1.7.1 Arithmetic Operators

The arithmetic operators supported by Python are mentioned below:

| Operator | Example | Meaning | Result |
|---|---|---|---|
| + (unary) | `+a` | **Unary Positive** | `a`<br>In other words, it doesn't really do anything. It mostly exists for the sake of completeness, to complement **Unary Negation**. |
| + (binary) | `a + b` | **Addition** | Sum of `a` and `b` |
| - (unary) | `-a` | **Unary Negation** | Value equal to `a` but opposite in sign |
| - (binary) | `a - b` | **Subtraction** | `b` subtracted from `a` |
| * | `a * b` | **Multiplication** | Product of `a` and `b` |
| / | `a / b` | **Division** | Quotient when `a` is divided by `b`.<br>The result always has type `float`. |
| % | `a % b` | **Modulo** | Remainder when `a` is divided by `b` |
| // | `a // b` | **Floor Division** (also called **Integer Division**) | Quotient when `a` is divided by `b`, rounded to the next smallest whole number |
| ** | `a ** b` | **Exponentiation** | `a` raised to the power of `b` |

Here are some examples of these operators in use:

```
In [77]: a=4
         b=3
         print(a+b)
         print(a-b)
         print(a*b)
         print(a%b)
         print(a/b)
         print(a**b)
         print(-a)
         print(+b)
```

```
7
1
12
1
1.3333333333333333
64
-4
3
```

The result of standard division (/) is always a float, even if the dividend is evenly divisible by the divisor:

```
In [78]: 10/5
Out[78]: 2.0

In [79]: type(10/5)
Out[79]: float
```

When the result of floor division (//) is positive, it is as though the fractional portion is truncated off, leaving only the integer portion. When the result is negative, the result is rounded down to the next smallest (greater negative) integer:

```
[80]: 10/4
[80]: 2.5
[81]: 10//4
[81]: 2
[82]: 10//-4
[82]: -3
[83]: -10//4
[83]: -3
[84]: -10//-4
```

### 1.7.2 Comparison Operators

The Comparison Operators shown below.

| Operator | Example | Meaning | Result |
|---|---|---|---|
| `==` | `a == b` | **Equal to** | `True` if the value of `a` is equal to the value of `b`<br>`False` otherwise |
| `!=` | `a != b` | **Not equal to** | `True` if `a` is not equal to `b`<br>`False` otherwise |
| `<` | `a < b` | **Less than** | `True` if `a` is less than `b`<br>`False` otherwise |
| `<=` | `a <= b` | **Less than or equal to** | `True` if `a` is less than or equal to `b`<br>`False` otherwise |
| `>` | `a > b` | **Greater than** | `True` if `a` is greater than `b`<br>`False` otherwise |
| `>=` | `a >= b` | **Greater than or equal to** | `True` if `a` is greater than or equal to `b`<br>`False` otherwise |

Here are examples of the comparison operators in use:

```
[85]: a=10
      b=20
      a==b

[85]: False

[86]: a!=b

[86]: True

[87]: a>=b

[87]: False

[88]: a<=b

[88]: True
```

### 1.7.3 Logical Operators

The logical operators (not, or, and) modify and join together expressions evaluated in Boolean context to create more complex conditions.

| Operator | Description | Example |
|---|---|---|
| and | Returns True if both statements are true | x < 5 and x < 10 |
| or | Returns True if one of the statements is true | x < 5 or x < 4 |
| not | Reverse the result, returns False if the result is true | not(x < 5 and x < 10) |

Examples:

```
In [90]: a=True
         b=False
         print("a and b is  ",a and b)
         print("a or b is ",a or b)
         print ("not a is ",not a)

         a and b is   False
         a or b is  True
         not a is  False
```

### 1.7.4 Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

| Operator | Description | Example |
|---|---|---|
| is | Returns True if both variables are the same object | x is y |
| is not | Returns True if both variables are not the same object | x is not y |

### 1.7.5 Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

| Operator | Name | Description |
|---|---|---|
| & | AND | Sets each bit to 1 if both bits are 1 |
| \| | OR | Sets each bit to 1 if one of two bits is 1 |
| ^ | XOR | Sets each bit to 1 if only one of two bits is 1 |
| ~ | NOT | Inverts all the bits |
| << | Zero fill left shift | Shift left by pushing zeros in from the right and let the leftmost bits fall off |
| >> | Signed right shift | Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off |

## 1.8 Tasks:

# 2. Lab#02 - Lists and Tuple functions in python

---

## Objectives:

- To learn and able to create, modify lists and access their elements
- To learn and able to use slicing in lists
- To learn and able to create and use tuples and tuples packing and unpacking.

## Outcomes:

- Students should be able to create, modify lists and access their elements
- Students should be able to use slicing in lists
- Student should be able to create and use tuples and tuples packing and unpacking

---

## 2.1 Lists

List is a collection of anything, it is like an array in many other programming languages, but it can be used for more things. Python lists are made by putting a comma-separated sequence of objects in square brackets ([]), as shown in the picture below.

```
[95]: a=['foo','bar','baz','quix']

[96]: print(a)
      ['foo', 'bar', 'baz', 'quix']

[97]: a
```

The important characteristics of Python lists are as follows:

- Lists are ordered.
- Lists can contain any arbitrary objects.
- List elements can be accessed by index.
- Lists can be nested to arbitrary depth.
- Lists are mutable.
- Lists are dynamic.

Each of these features is examined in more detail below.

### 2.1.1 Lists Are Ordered

A list is not merely a collection of objects. It is an ordered collection of objects. The order in which you specify the elements when you define a list is an innate characteristic of that list and is maintained for that list's lifetime.

Lists that have the same elements in a different order are not the same:

```
In [98]: a=['foo','bar','baz','quix']
         b=['bar','foo','baz','quix']
         a==b
Out[98]: False

In [99]: a is b
Out[99]: False
```

### 2.1.2 Lists Can Contain Arbitrary Objects

A list can contain any assortment of objects. The elements of a list can all be the same type:

```
[100]: a=[2,4,6,8]
       a
[100]: [2, 4, 6, 8]
```

Or the elements can be of varying types:

```
[101]: a=[1,2,'foo',2.34,True]
       a
[101]: [1, 2, 'foo', 2.34, True]
```

Lists can even contain complex objects, like functions, classes, and modules, which you will learn about in upcoming labs:

```
[103]: len
[103]: <function len(obj, /)>

[105]: def foo():
           pass
       foo
[105]: <function __main__.foo()>

[106]: import math
       math
[106]: <module 'math' (built-in)>

[107]: a=[int,len,foo,math]
       a
[107]: [int,
        <function len(obj, /)>,
        <function __main__.foo()>,
        <module 'math' (built-in)>]
```

A list can contain any number of objects, from zero to as many as your computer's memory will allow:

```
[22]: a=[]
      a

[22]: []

[23]: a=['foo']
      a

[23]: ['foo']

[24]: a=[1,2,3,4,5,6,7,7,8,45,56,567,67,67,6776,45,34,2345,5678,345,33445]
      print(a)

      [1, 2, 3, 4, 5, 6, 7, 7, 8, 45, 56, 567, 67, 67, 6776, 45, 34, 2345, 5678, 345, 33445]
```

(A list with a single object is sometimes referred to as a singleton list.)

List objects needn't be unique. A given object can appear in a list multiple time:

```
[25]: #may not have unique elements
      a=['bark','bark','meow','meow']
      print(a)

      ['bark', 'bark', 'meow', 'meow']
```

### 2.1.3 List Elements Can Be Accessed by Index

Individual elements in a list can be accessed using an index in square brackets. This is exactly analogous to accessing individual characters in a string. List indexing is zero-based as it is with strings.

Consider the following list:

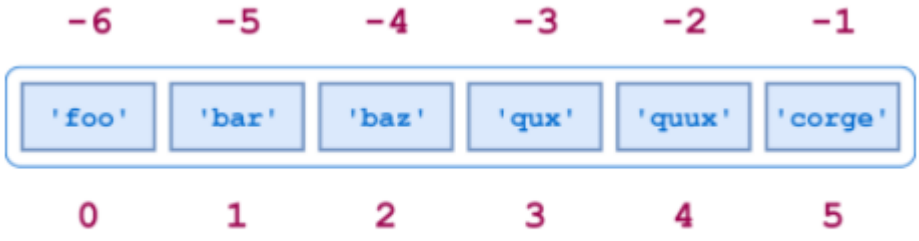
```
In [124]: #indexes
          a=['foo','bar','baz','qux','quux','corge']
```

The indices for the elements are shown below:

Here is Python code to access some elements of a list:

```
[27]: a[0]
[27]: 'foo'
[28]: a[2]
[28]: 'baz'
[29]: a[5]
[29]: 'corge'
```

Virtually everything about string indexing works similarly for lists. For example, a negative list index counts from the end of the list:




Negative List Indexing

```
[30]: a[-2]
[30]: 'quux'
[31]: a[-5]
[31]: 'bar'
```

### 2.1.4 List Slicing

Slicing also works. If a is a list, the expression a[m:n] returns the portion of a from index m to, but not including, index n:

**Example: a[start : end]**

```
[131]: #slicing
       a=['foo','bar','baz','qux','quux','corge']
       a[2:5]
[131]: ['baz', 'qux', 'quux']
```

Other features of string slicing work analogously for list slicing as well:

Both positive and negative indices can be specified:

```
In [132]: a[-5:-2]
Out[132]: ['bar', 'baz', 'qux']

In [133]: a[1:4]
Out[133]: ['bar', 'baz', 'qux']

In [134]: a[-5:-2]==a[1:4]
Out[134]: True
```

Omitting the first index starts the slice at the beginning of the list, and omitting the second index extends the slice to the end of the list:

```
[136]: print(a[:4],a[0:4])
       ['foo', 'bar', 'baz', 'qux'] ['foo', 'bar', 'baz', 'qux']

[137]: print(a[2:],a[2:len(a)])
       ['baz', 'qux', 'quux', 'corge'] ['baz', 'qux', 'quux', 'corge']

[138]: a[:4]+a[4:]

[138]: ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

[139]: a[:4]+a[4:]==a

[139]: True
```

**Example: a[start : end : increment]**

```
[140]: print(a[0:6:2])
       ['foo', 'baz', 'quux']

[141]: print(a[1:6:2])
       ['bar', 'qux', 'corge']
```

The syntax for reversing a list works the same way it does for strings:

```
[142]: a[::-1]

[142]: ['corge', 'quux', 'qux', 'baz', 'bar', 'foo']
```

The [:] syntax works for lists. However, there is an important difference between how this operation works with a list and how it works with a string.

If s is a string, s[:] returns a reference to the same object:

```
In [144]: s='food bar'
          s[:]
Out[144]: 'food bar'

In [145]: s[:]  is s
Out[145]: True
```

Conversely, if a is a list, a[:] returns a new object that is a copy of a:

```
In [148]: a=['foo','bar','baz','qux','quux','corge']
          a[:]
Out[148]: ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

In [149]: a[:]  is a
Out[149]: False
```

Several Python operators and built-in functions can also be used with lists in ways that are analogous to strings:

The in and not in operators:

```
In [151]: 'qux' in a
Out[151]: True

In [152]: 'thud ' not in a
Out[152]: True
```

The concatenation (+) and replication (*) operators:

```
In [153]: a+['graulr','graply']
Out[153]: ['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'graulr', 'graply']

In [155]: print(a*2)
          ['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'foo', 'bar', 'baz', 'qux', 'quux', 'corge']
```

The `len()`, `min()`, and `max()` functions:

```
[156]: len(a)

[156]: 6

[158]: max(a)

[158]: 'qux'

[159]: min(a)

[159]: 'bar'
```

It’s not an accident that strings and lists behave so similarly. They are both special cases of a more general object type called an iterable, which you will encounter in more detail in the upcoming labs on definite iteration.

By the way, in each example above, the list is always assigned to a variable before an operation is performed on it. But you can operate on a list literal as well:

```
[160]: ['foo', 'bar', 'baz', 'qux', 'quux', 'corge'][2]
[160]: 'baz'
[161]: ['foo', 'bar', 'baz', 'qux', 'quux', 'corge'][::-1]
[161]: ['corge', 'quux', 'qux', 'baz', 'bar', 'foo']
[162]: len(['foo', 'bar', 'baz', 'qux', 'quux', 'corge'][::-1])
[162]: 6
```

For that matter, you can do likewise with a string literal:

```
[163]: "python is most popular programming language"[::-1]
[163]: 'egaugnal gnimmargorp ralupop tsom si nohtyp'
```

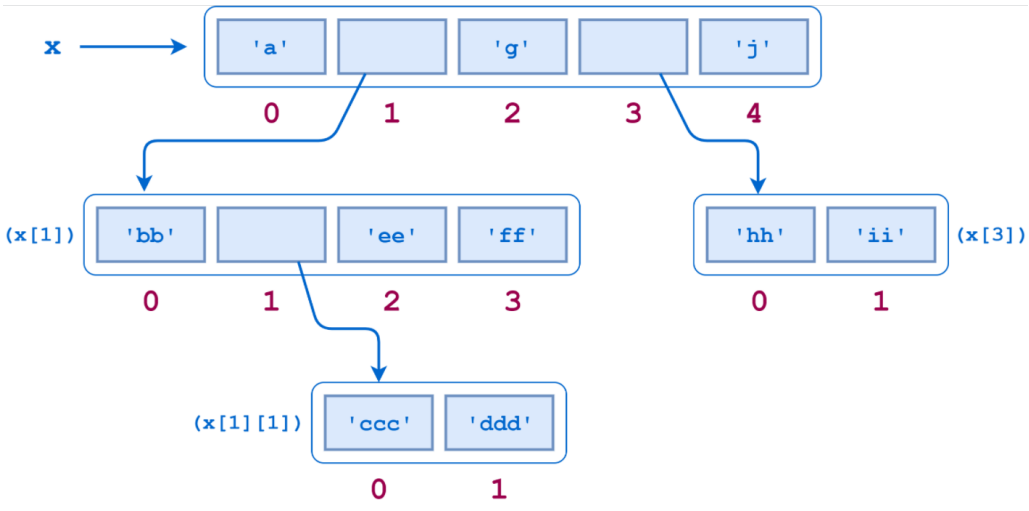
### 2.1.5 Lists Can Be Nested

You have seen that an element in a list can be any sort of object. That includes another list. A list can contain sub lists, which in turn can contain sub lists themselves, and so on to arbitrary depth.

Consider this (admittedly contrived) example:

```
In [165]: x=['a',['bb',['ccc','ddd'],'ee','ff'],'g',['hh','ii'],'j']
          x
Out[165]: ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
```

The object structure that x references is diagrammed below:



x[0], x[2], and x[4] are strings, each one character long:

```
[34]: print(x[0], x[2], x[4])
      a g j
```

But x[1] and x[3] are sub lists:

```
[166]: x[1]
[166]: ['bb', ['ccc', 'ddd'], 'ee', 'ff']
[167]: x[3]
```

To access the items in a sub list, simply append an additional index:

```
[168]: x[1][0]
[168]: 'bb'
[169]: x[1][1]
[169]: ['ccc', 'ddd']
[170]: x[1][2]
[170]: 'ee'
```

x[1][1] is yet another sub list, so adding one more index accesses its elements:

```
[170]: x[1][2]
[170]: 'ee'
[171]: x[1][1][0],x[1][1][1]
[171]: ('ccc', 'ddd')
```

There is no limit, short of the extent of your computer's memory, to the depth or complexity with which lists can be nested in this way.

All the usual syntax regarding indices and slicing applies to sub lists as well:

```
[172]: x[1][1][-1]
[172]: 'ddd'
[173]: x[1][1:3]
[173]: [['ccc', 'ddd'], 'ee']
[174]: x[3][::-1]
[174]: ['ii', 'hh']
```

However, be aware that operators and functions apply to only the list at the level you specify and are not recursive. Consider what happens when you query the length of x using len():

```
[175]: len(x)
[175]: 5
[176]: x[2]
[176]: 'g'
[177]: x[1]
[177]: ['bb', ['ccc', 'ddd'], 'ee', 'ff']
[178]: x[3]
[178]: ['hh', 'ii']
```

x has only five elements—three strings and two sublists. The individual elements in the sublists don't count toward x's length.

You'd encounter a similar situation when using the in operator:

```
[179]: 'ddd'in x
[179]: False
[180]: 'ddd' in x[1]
[180]: False
[181]: 'ddd' in x[1][1]
[181]: True
```

'ddd' is not one of the elements in x or x[1]. It is only directly an element in the sublist x[1][1]. An individual element in a sublist does not count as an element of the parent list(s).

### 2.1.6 Lists Are Mutable

Most of the data types you have encountered so far have been atomic types. Integer or float objects, for example, are primitive units that can't be further broken down. These types are immutable, meaning that they

can’t be changed once they have been assigned. It doesn’t make much sense to think of changing the value of an integer. If you want a different integer, you just assign a different one.

By contrast, the string type is a composite type. Strings are reducible to smaller parts—the component characters. It might make sense to think of changing the characters in a string. But you can’t. In Python, strings are also immutable.

The list is the first mutable data type you have encountered. Once a list has been created, elements can be added, deleted, shifted, and moved around at will. Python provides a wide range of ways to modify lists.

### 2.1.6.1 Modifying a Single List Value

A single value in a list can be replaced by indexing and simple assignment:

```
In [182]: m=['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
          m

Out[182]: ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

In [184]: m[1]=23
          m[-1]=29
          m

Out[184]: ['foo', 23, 'baz', 'qux', 'quux', 29]
```

You can’t do this with a string:

```
In [185]: #cant do this with string
          s="Disney"
          s[1]=x

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-185-0d716db0dc8d> in <module>
      1 #cant do this with string
      2 s="Disney"
----> 3 s[1]=x

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

A list item can be deleted with the del command:

```
In [186]: m=['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
          del m[3]
          m

Out[186]: ['foo', 'bar', 'baz', 'quux', 'corge']
```

### 2.1.6.2 Modifying Multiple List Values

What if you want to change several contiguous elements in a list at one time? Python allows this with slice assignment, which has the following syntax:

```
In [188]: a[m:n]=<iterable>
```

Again, for the moment, think of an iterable as a list. This assignment replaces the specified slice of a with <iterable>:

```
In [189]: a[1:4]
Out[189]: ['bar', 'baz', 'quux']

In [190]: a[1:4]=['pizza','burger','fries']

In [191]: a
Out[191]: ['foo', 'pizza', 'burger', 'fries', 'corge']

In [193]: a[1:6]=['bark']
          a
Out[193]: ['foo', 'bark']
```

The number of elements inserted need not be equal to the number replaced. Python just grows or shrinks the list as needed.

You can insert multiple elements in place of a single element—just use a slice that denotes only one element:

```
In [194]: a=[1,2,3]
          a[1:2]=[1.2,1.3,1.4]
          a
Out[194]: [1, 1.2, 1.3, 1.4, 3]
```

Note that this is not the same as replacing the single element with a list:

```
In [195]: a=[1,2,3]
          a[1]=[1.2,1.3,1.4]
          a
Out[195]: [1, [1.2, 1.3, 1.4], 3]
```

You can also insert elements into a list without removing anything. Simply specify a slice of the form [n:n] (a zero-length slice) at the desired index:

```
In [196]: a=[1,2,3]
          a[2:2]=[1.2,1.3,1.4]
          a
Out[196]: [1, 2, 1.2, 1.3, 1.4, 3]
```

You can delete multiple elements out of the middle of a list by assigning the appropriate slice to an empty list. You can also use the del statement with the same slice:

```
In [197]: a=['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
          a[1:5]=[]
          a
Out[197]: ['foo', 'corge']

In [199]: del a[1:5]
          a
Out[199]: ['foo']
```

### 2.1.6.3 Prepending or Appending Items to a List

Additional items can be added to the start or end of a list using the + concatenation operator or the += augmented assignment operator:

```
[200]: a=['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
       a+=['graply','grault']
       a

[200]: ['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'graply', 'grault']

[201]: a=[10,20]+a
       a

[201]: [10, 20, 'foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'graply', 'grault']
```

Note that a list must be concatenated with another list, so if you want to add only one element, you need to specify it as a singleton list:

```
In [202]: a+=20

----------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-202-1096ec335417> in <module>
----> 1 a+=20

TypeError: 'int' object is not iterable
```

```
In [212]: a += [20]
          print(a)

[10, 20, 'foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'graply', 'grault', 20, 20, 20, 'c', 'o', 'r', 'g', 'e', 'corge', 'corg
e', 'c', 'o', 'r', 'g', 'e', 20]
```

Note: Technically, it isn't quite correct to say a list must be concatenated with another list. More precisely, a list must be concatenated with an object that is iterable. Of course, lists are iterable, so it works to concatenate a list with another list.

Strings are iterable also. But watch what happens when you concatenate a string onto a list:

```
In [211]: a+='corge'
          print(a)

[10, 20, 'foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'graply', 'grault', 20, 20, 20, 'c', 'o', 'r', 'g', 'e', 'corge', 'corg
e', 'c', 'o', 'r', 'g', 'e']
```

This result is perhaps not quite what you expected. When a string is iterated through, the result is a list of its component characters. In the above example, what gets concatenated onto list a is a list of the characters in the string 'corge'.

If you really want to add just the single string 'corge' to the end of the list, you need to specify it as a singleton list:

```
In [210]: a+=['corge']
          print(a)

[10, 20, 'foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'graply', 'grault', 20, 20, 20, 'c', 'o', 'r', 'g', 'e', 'corge', 'corg
e']
```

### 2.1.6.4 Methods That Modify a List

Finally, Python supplies several built-in methods that can be used to modify lists. Information on these methods is detailed below.

- `a.append(<obj>)` Appends an object to a list.

`a.append(<obj>)` appends object <obj> to the end of list a:

```
In [215]: b=['a','b']
          b.append(1)
          b

Out[215]: ['a', 'b', 1]
```

Remember, list methods modify the target list in place. They do not return a new list:

```
In [217]: b=['a','b']
          x=b.append(1)
          print(x)

          None
```

Remember that when the + operator is used to concatenate to a list, if the target operand is an iterable, then its elements are broken out and appended to the list individually:

```
In [218]: a=['a,b']
          a=a+[1,2,3]
          a

Out[218]: ['a,b', 1, 2, 3]
```

The `.append()` method does not work that way! If an iterable is appended to a list with `.append()`, it is added as a single object:

```
In [221]: a=['a,b']
          a.append([1,2,3])
          a

Out[221]: ['a,b', [1, 2, 3]]
```

- `a.extend(<iterable>)` Extends a list with the objects from an iterable.

Yes, this is probably what you think it is. `.extend()` also adds to the end of a list, but the argument is expected to be an iterable. The items in <iterable> are added individually:

```
In [222]: a=['a,b']
          a.extend([1,2,3])
          a

Out[222]: ['a,b', 1, 2, 3]
```

In other words, .extend() behaves like the + operator. More precisely, since it modifies the list in place, it behaves like the += operator:

```
In [223]: a=['a,b']
          a+=([1,2,3])
          a

Out[223]: ['a,b', 1, 2, 3]
```

- `a.insert(<index>, <obj>)` Inserts an object into a list.

`a.insert(<index>, <obj>)` inserts object <obj> into list a at the specified <index>. Following the method call, a[<index>] is <obj>, and the remaining list elements are pushed to the right:

```
In [225]: a=['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
          a.insert(3,12345)
          a
Out[225]: ['foo', 'bar', 'baz', 12345, 'qux', 'quux', 'corge']
```

- `a.remove(<obj>)` Removes an object from a list.

`a.remove(<obj>)` removes object <obj> from list a. If <obj> isn't in a, an exception is raised:

```
In [226]: a=['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
          a.remove('baz')
          a
Out[226]: ['foo', 'bar', 'qux', 'quux', 'corge']

In [227]: a.remove('sam')
          ---------------------------------------------------------------------------
          ValueError                                Traceback (most recent call last)
          <ipython-input-227-37be4471fed9> in <module>
          ----> 1 a.remove('sam')

          ValueError: list.remove(x): x not in list
```

- `a.pop(index=-1)` Removes an element from a list.

This method differs from `.remove()` in two ways:

- You specify the index of the item to remove, rather than the object itself.
- The method returns a value: the item that was removed.

`a.pop()` simply removes the last item in the list:

```
In [229]: a=['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
          a.pop()
Out[229]: 'corge'

In [230]: a
Out[230]: ['foo', 'bar', 'baz', 'qux', 'quux']

In [231]: a.pop()
Out[231]: 'quux'

In [232]: a
Out[232]: ['foo', 'bar', 'baz', 'qux']
```

If the optional <index> parameter is specified, the item at that index is removed and returned. <index> may be negative, as with string and list indexing:

```
In [233]: a=['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
          a.pop(1)
Out[233]: 'bar'

In [234]: a.pop(-3)
Out[234]: 'qux'

In [235]: a
Out[235]: ['foo', 'baz', 'quux', 'corge']
```

<index> defaults to -1, so a.`pop(-1)` is equivalent to `a.pop()`.

## 2.2 Python Tuples

Python provides another type that is an ordered collection of objects, called a tuple.

Pronunciation varies depending on whom you ask. Some pronounce it as though it were spelled “too-ple” (rhyming with “Mott the Hoople”), and others as though it were spelled “tup-ple” (rhyming with “supple”). My inclination is the latter, since it presumably derives from the same origin as “quintuple,” “sextuple,” “octuple,” and so on, and everyone I know pronounces these latter as though they rhymed with “supple.”

### 2.2.1 Defining and Using Tuples

Tuples are identical to lists in all respects, except for the following properties:

- Tuples are defined by enclosing the elements in parentheses (()) instead of square brackets ([]).
- Tuples are immutable.

Here is a short example showing a tuple definition, indexing, and slicing:

```
In [236]: t=('foo', 'bar', 'baz', 'qux', 'quux', 'corge')
          t
Out[236]: ('foo', 'bar', 'baz', 'qux', 'quux', 'corge')

In [237]: t[0]
Out[237]: 'foo'

In [238]: t[-1]
Out[238]: 'corge'

In [239]: t[1::2]
Out[239]: ('bar', 'qux', 'corge')
```

Never fear! Our favorite string and list reversal mechanism works for tuples as well:

```
In [240]: t[::-1]
Out[240]: ('corge', 'quux', 'qux', 'baz', 'bar', 'foo')
```

Everything you’ve learned about lists—they are ordered, they can contain arbitrary objects, they can be indexed and sliced, they can be nested—is true of tuples as well. But they can’t be modified:

```
In [241]: t[2]='bark'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-241-8a53fd9cff43> in <module>
----> 1 t[2]='bark'

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

Why use a tuple instead of a list?

- Program execution is faster when manipulating a tuple than it is for the equivalent list. (This is probably not going to be noticeable when the list or tuple is small.)
- Sometimes you don’t want data to be modified. If the values in the collection are meant to remain constant for the life of the program, using a tuple instead of a list guards against accidental modification.

- There is another Python data type that you will encounter shortly called a dictionary, which requires as one of its components a value that is of an immutable type. A tuple can be used for this purpose, whereas a list can't be.

In a Python REPL session, you can display the values of several objects simultaneously by entering them directly at the >>> prompt, separated by commas:

```
In [242]: a='foo'
          b=45
          a,b,123
Out[242]: ('foo', 45, 123)
```

Python displays the response in parentheses because it is implicitly interpreting the input as a tuple.

There is one peculiarity regarding tuple definition that you should be aware of. There is no ambiguity when defining an empty tuple, nor one with two or more elements. Python knows you are defining a tuple:

```
In [243]: t=()
          type(t)
Out[243]: tuple

In [244]: t=(1,2,3)
          type(t)
Out[244]: tuple
```

But what happens when you try to define a tuple with one item:

```
In [246]: t=(2)
          type(t)
Out[246]: int
```

Doh! Since parentheses are also used to define operator precedence in expressions, Python evaluates the expression (2) as simply the integer 2 and creates an int object. To tell Python that you really want to define a singleton tuple, include a trailing comma (,) just before the closing parenthesis:

```
In [248]: t=(2,)
          type(t)
Out[248]: tuple

In [249]: t[0]
Out[249]: 2

In [250]: t[-1]
Out[250]: 2
```

You probably won't need to define a singleton tuple often, but there has to be a way.

When you display a singleton tuple, Python includes the comma, to remind you that it's a tuple:

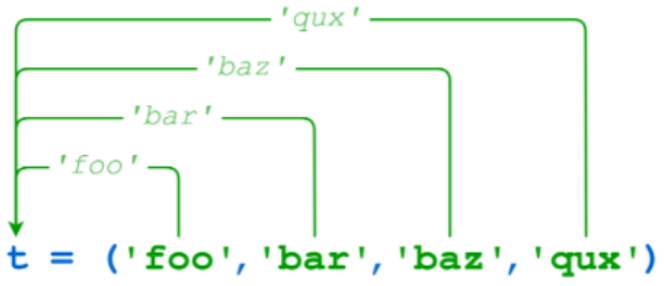
```
In [251]: print(t)
          (2,)
```

### 2.2.2 Tuple Assignment, Packing, and Unpacking

As you have already seen above, a literal tuple containing several items can be assigned to a single object:

```
In [236]: t=('foo', 'bar', 'baz', 'qux', 'quux', 'corge')
          t
```

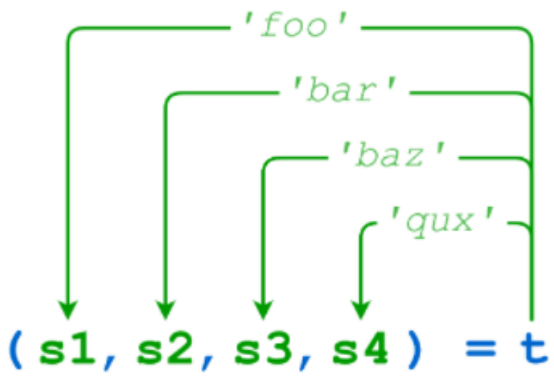
When this occurs, it is as though the items in the tuple have been "packed" into the object:




Tuple Packing

```
In [236]: t=('foo', 'bar', 'baz', 'qux', 'quux', 'corge')
          t
Out[236]: ('foo', 'bar', 'baz', 'qux', 'quux', 'corge')

In [237]: t[0]
Out[237]: 'foo'

In [238]: t[-1]
Out[238]: 'corge'
```

If that "packed" object is subsequently assigned to a new tuple, the individual items are "unpacked" into the objects in the tuple:




Tuple Unpacking

```
In [261]: t=('foo', 'bar', 'baz', 'qux')
          (s1,s2,s3,s4)=t

In [262]: s1
Out[262]: 'foo'

In [263]: s2
Out[263]: 'bar'

In [264]: s3
Out[264]: 'baz'
```

When unpacking, the number of variables on the left must match the number of values in the tuple:

```
In [265]: t=('foo', 'bar', 'baz', 'qux')
          (s1,s2,s3)=t

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-265-258a4bdb5022> in <module>
      1 t=('foo', 'bar', 'baz', 'qux')
----> 2 (s1,s2,s3)=t

ValueError: too many values to unpack (expected 3)

In [266]: t=('foo', 'bar', 'baz', 'qux')
          (s1,s2,s3,s4,s5)=t

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-266-1e560363fa7e> in <module>
      1 t=('foo', 'bar', 'baz', 'qux')
----> 2 (s1,s2,s3,s4,s5)=t

ValueError: not enough values to unpack (expected 5, got 4)
```

Packing and unpacking can be combined into one statement to make a compound assignment:

```
In [267]: (s1,s2,s3,s4)=('foo', 'bar', 'baz', 'qux')

In [268]: s1
Out[268]: 'foo'

In [269]: s4
Out[269]: 'qux'
```

Again, the number of elements in the tuple on the left of the assignment must equal the number on the right:

```
In [270]: (s1,s2,s3,s4,s5)=('foo', 'bar', 'baz', 'qux')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-270-856e25dbc0f5> in <module>
----> 1 (s1,s2,s3,s4,s5)=('foo', 'bar', 'baz', 'qux')

ValueError: not enough values to unpack (expected 5, got 4)
```

## 2.3 Tasks:

# 3. Lab#03 - Conditional Statements, Loop (while, for), break and continue

---

## Objectives:

- To learn and to use conditional statements (if, elif, else)
- To learn and able to use loop (for loop, while loop)
- To learn and able to use the break and continue statement
- To learn and able to use loop with lists

## Outcomes:

- Students should be able to know the if, elif and else.
- Students should be able to use loops for loop and while loop in the programs.
- Students should be able to use break and continue statement with loop.
- Students should be able to use loop with lists

---

## 3.1 Conditional Statements

In a Python program, the if statement is how you perform this sort of decision-making. It allows for conditional execution of a statement or group of statements based on the value of an expression.

### 3.1.1 Introduction to the if Statement

We'll start by looking at the most basic type of if statement. In its simplest form, it looks like this:

```
In [ ]: if <expr>:
            <statement>
```

In the form shown above:

- <expr> is an expression evaluated in a Boolean context, as discussed in the section on Logical Operators in the Operators and Expressions in Python tutorial.
- <statement> is a valid Python statement, which must be indented. (You will see why very soon.)

If <expr> is true (evaluates to a value that is "truthy"), then <statement> is executed. If <expr> is false, then <statement> is skipped over and not executed.

Note that the colon (:) following <expr> is required. Some programming languages require <expr> to be enclosed in parentheses, but Python does not.

Here are several examples of this type of if statement:

```
In [15]: x = 0
         y = 5

         if x < y:        # Truthy
             print('yes-1')

         if y < x:                          # Falsy
             print('yes-2')

         if x:                              # Falsy
             print('yes-3')

         if y:                              # Truthy
             print('yes-4')

         if x or y:                         # Truthy
             print('yes-5')

         if x and y:                        # Falsy
             print('yes-6')
```

```
yes-1
yes-4
yes-5
```

### 3.1.2 Grouping Statements: Indentation and Blocks

In all the examples shown above, each if <expr>: has been followed by only a single <statement>. There needs to be some way to say "If <expr> is true, do all of the following things."

The usual approach taken by most programming languages is to define a syntactic device that groups multiple statements into one compound statement or block. A block is regarded syntactically as a single entity. When it is the target of an if statement, and <expr> is true, then all the statements in the block are executed. If <expr> is false, then none of them are.

Virtually all programming languages provide the capability to define blocks, but they don't all provide it in the same way. Let's see how Python does it.

**In a Python program, contiguous statements that are indented to the same level are considered to be part of the same block.**

```
In [ ]: if <expr>:
            statement1
            statement2
            statement3
            .
            .
            statement n
        <following statement>
```

Here, all the statements at the matching indentation level (lines 2 to 5) are considered part of the same block. The entire block is executed if <expr> is true or skipped over if <expr> is false. Either way, execution proceeds with <following_statement> (line 6) afterward.

Notice that there is no token that denotes the end of the block. Rather, the end of the block is indicated by a line that is indented less than the lines of the block itself.

Consider the following code.

```
In [17]:
1  x = 3
2
3  if x % 2 == 0:
4      print('Expression was true')
5      print('Executing statement in suite')
6      print('...')
7      print('Even Number')
8      print('Done.')
9  print('After conditional')
```

```
After conditional
```

The four `print()` statements on lines 4 to 8 are indented to the same level as one another. They constitute the block that would be executed if the condition were true. But it is false, so all the statements in the block are skipped. After the end of the compound if statement has been reached (whether the statements in the block on lines 4 to 8 are executed or not), execution proceeds to the first statement having a lesser indentation level: the `print()` statement on line 9.

Blocks can be nested to arbitrary depth. Each indent defines a new block, and each out dent ends the preceding block. The resulting structure is straightforward, consistent, and intuitive.

Consider this complex example and check the output generated when this code is run is shown below:

```
In [21]:
1  x = 8
2  if x % 2 == 0:       #  x
3      print('Outer condition is true')      #  x
4
5      if x > 20:                            #  x
6          print('Inner condition 1')        #        x
7
8      print('Between inner conditions')     #  x
9
10     if x < 20:                            #  x
11         print('Inner condition 2')        #  x
12
13     print('End of outer condition')       #  x
14 print('After outer condition')            #  x
```

```
Outer condition is true
Between inner conditions
Inner condition 2
End of outer condition
After outer condition
```

### 3.1.3 The else and elif Clauses.

Now you know how to use an if statement to conditionally execute a single statement or a block of several statements. It's time to find out what else you can do.

Sometimes, you want to evaluate a condition and take one path if it is true but specify an alternative path if it is not. This is accomplished with an else clause:

```
In [ ]:
if<expr>:
    <statement>
else:
    <statement>
```

If <expr> is true, the first suite is executed, and the second is skipped. If <expr> is false, the first suite is skipped and the second is executed. Either way, execution then resumes after the second suite. Both suites are defined by indentation, as described above.

In this example, x is less than 50, so the first suite (lines 4 to 5) is executed, and the second suite (lines 7 to 8) are skipped:

```
In [22]:
1 x = 20
2
3 if x < 50:
4     print('(first suite)')
5     print('x is small')
6 else:
7     print('(second suite)')
8     print('x is large')
```

```
(first suite)
x is small
```

Here, on the other hand, x is greater than 50, so the first suite is passed over, and the second suite executed:

```
In [23]:
1 x = 70
2
3 if x < 50:
4     print('(first suite)')
5     print('x is small')
6 else:
7     print('(second suite)')
8     print('x is large')
```

```
(second suite)
x is large
```

There is also syntax for branching execution based on several alternatives. For this, use one or more elif (short for else if) clauses. Python evaluates each <expr> in turn and executes the suite corresponding to the first that is true. If none of the expressions are true, and an else clause is specified, then its suite is executed:

```
In [ ]:
if<expr>:
    <statement>
elif:
    <statement>
elif:
    <statement>
elif:
    <statement>
else:
     <statement>
```

An arbitrary number of elif clauses can be specified. The else clause is optional. If it is present, there can be only one, and it must be specified last:

In [25]:
```
name = 'Ali'
if name == 'Ali':
    print('Hello Ali')
elif name == 'Asad':
    print('Hello Asad')
elif name == 'Nimra':
    print('Hello Nimra')
elif name == 'Saa':
    print('Hello Saa')
else:
    print("I don't know who you are!")

```

```
Hello Ali
```

At most, one of the code blocks specified will be executed. If an else clause isn't included, and all the conditions are false, then none of the blocks will be executed.

Multiple statements may be specified on the same line as an elif or else clause as well

Ans shown below:

In [28]:
```
x = 2
if x == 1: print('foo'); print('bar'); print('baz')
elif x == 2: print('qux'); print('quux')
else: print('corge'); print('grault')
```

```
qux
quux
```

### 3.1.4 Conditional Expressions (Python's Ternary Operator)

Python supports one additional decision-making entity called a conditional expression. (It is also referred to as a conditional operator or ternary operator in various places in the Python documentation.)

In its simplest form, the syntax of the conditional expression is as follows:

In [ ]:
```
<expr1> if <conditional_expr> else <expr2>
```

This is different from the if statement forms listed above because it is not a control structure that directs the flow of program execution. It acts more like an operator that defines an expression. In the above example, <conditional_expr> is evaluated first. If it is true, the expression evaluates to <expr1>. If it is false, the expression evaluates to <expr2>.

Notice the non-obvious order: the middle expression is evaluated first, and based on that result, one of the expressions on the ends is returned. Here are some examples that will hopefully help clarify:

In [36]:
```
a = 2
b = 3
if a > b:
    m = a
else:
    m = b
print(m)
```

```
3
```

In [37]:
```
m = a if a > b else b
print(m)
```

```
3
```

Remember that the conditional expression behaves like an expression syntactically. It can be used as part of a longer expression. The conditional expression has lower precedence than virtually all the other operators, so parentheses are needed to group it by itself.

In the following example, the + operator binds more tightly than the conditional expression, so 1 + x and y + 2 are evaluated first, followed by the conditional expression. The parentheses in the second case are unnecessary and do not change the result:

In [40]:
```
x = y = 40

z = 1 + x if x > y else y + 2
print(z)


z = (1 + x) if x > y else (y + 2)
print(z)
```

```
42
42
```

If you want the conditional expression to be evaluated first, you need to surround it with grouping parentheses. In the next example, (x if x > y else y) is evaluated first. The result is y, which is 40, so z is assigned 1 + 40 + 2 = 43:

In [41]:
```
x = y = 40

z = 1 + (x if x > y else y) + 2
print(z)
```

```
43
```

## 3.2 Loops in Python

### 3.2.1 While loop

The format of a basic while loop is shown below:

In [ ]:
```
while <expr> :
    <statement>
```

<statement(s)> represents the block to be repeatedly executed, often referred to as the body of the loop. This is denoted with indentation, just as in an if statement.

The controlling expression, <expr>, typically involves one or more variables that are initialized prior to starting the loop and then modified somewhere in the loop body.

When a while loop is encountered, <expr> is first evaluated in Boolean context. If it is true, the loop body is executed. Then <expr> is checked again, and if still true, the body is executed again. This continues until <expr> becomes false, at which point program execution proceeds to the first statement beyond the loop body.

Consider this loop:

In [42]:
```
1 n = 5
2 while n > 0:
3     n -= 1
4     print(n)
```

```
4
3
2
1
0
```

Here's what's happening in this example:

- n is initially 5. The expression in the while statement header on line 2 is n > 0, which is true, so the loop body executes. Inside the loop body on line 3, n is decremented by 1 to 4, and then printed.
- When the body of the loop has finished, program execution returns to the top of the loop at line 2, and the expression is evaluated again. It is still true, so the body executes again, and 3 is printed.
- This continues until n becomes 0. At that point, when the expression is tested, it is false, and the loop terminates. Execution would resume at the first statement following the loop body, but there isn't one in this case.
-

Note that the controlling expression of the while loop is tested first before anything else happens. If it's false to start with, the loop body will never be executed at all:

In [43]:
```
1 n = 0
2 while n > 0:
3     n -= 1
4     print(n)
```

In the example above, when the loop is encountered, n is 0. The controlling expression n > 0 is already false, so the loop body never executes.

Here's another while loop involving a list, rather than a numeric comparison:

In [282]:
```
a=['foo','bar','baz']
while a:
    print(a.pop(-1))
```

```
baz
bar
foo
```

When a list is evaluated in Boolean context, it is truthy if it has elements in it and falsy if it is empty. In this example, a is true as long as it has elements in it. Once all the items have been removed with the `.pop()` method and the list is empty, a is false, and the loop terminates.

## 3.2.2 For loop

### 3.2.2.1 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
- An action to be performed at the end of each iteration.

This type of loop has the following form:

```
C

for (i = 1; i <= 10; i++)
    <loop body>
```

This loop is interpreted as follows:

- Initialize i to 1.
- Continue looping as long as i <= 10.
- Increment i by 1, after iteration of each loop.

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 from shown above. These for loops are also featured in the C++, Java, PHP, and Perl languages.

In python

```
In [289]: fruits = ["apple", "banana", "cherry"]
          for x in fruits:
              print(x)

          apple
          banana
          cherry
```

### 3.2.2.2 The range() Function

`range(<end>)` returns an iterable that yields integers starting with 0, up to but not including <end>:

In Python, iterable means an object can be used in iteration.

```
In [290]: #range fuction

          x=range(5)
          x

Out[290]: range(0, 5)

In [291]: type(x)

Out[291]: range
```

Note that `range()` returns an object of class range, not a list or tuple of the values. Because a range object is an iterable, you can obtain the values by iterating over them with a for loop:

```
In [294]: for n in x:
              print(n)
```

```
0
1
2
3
4
```

The `range()` function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6):

```
In [295]: for x in range(2,6):
              print(x)
```

```
2
3
4
5
```

The `range()` function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, 3): **range(start, end, increment)**

```
In [296]: for x in range(2,30,3):
              print(x)
```

```
2
5
8
11
14
17
20
23
26
29
```

### 3.2.2.3 Loop for list

To iterate over a list, you use the for loop statement as follows:

```
for item in list:
    # process the item
```

In this syntax, the for loop statement assigns an individual element of the list to the item variable in each iteration.

Inside the body of the loop, you can manipulate each list element individually.

For example, the following defines a list of cities and uses a for loop to iterate over the list:

```
In [1]:
cities = ['New York', 'Beijing', 'Cairo', 'Mumbai', 'Mexico']

for city in cities:
    print(city)
```

```
New York
Beijing
Cairo
Mumbai
Mexico
```

```
In [2]:
cities = ['New York', 'Beijing', 'Cairo', 'Mumbai', 'Mexico']

for city in cities[:3]:
    print(city)
```

```
New York
Beijing
Cairo
```

In this example, the for loop assigns an individual element of the cities list to the city variable and prints out the city in each iteration.

### 3.2.3 Using Python for loop to iterate over a list with index

Sometimes, you may want to access indexes of elements inside the loop. In these cases, you can use the `enumerate()` function.

The `enumerate()` function returns a tuple that contains the current index and element of the list.

The following example defines a list of cities and uses a for loop with the `enumerate()` function to iterate over the list:

```
In [2]:
cities = ['New York', 'Beijing', 'Cairo', 'Mumbai', 'Mexico']

for item in enumerate(cities):
    print(item)
```

```
(0, 'New York')
(1, 'Beijing')
(2, 'Cairo')
(3, 'Mumbai')
(4, 'Mexico')
```

To access the index, you can unpack the tuple within the for loop statement like this:

```
In [3]:
cities = ['New York', 'Beijing', 'Cairo', 'Mumbai', 'Mexico']

for index, city in enumerate(cities):
    print(f"{index}: {city}")
```

```
0: New York
1: Beijing
2: Cairo
3: Mumbai
4: Mexico
```

The `enumerate()` function allows you to specify the starting index which defaults to zero.

The following example uses the `enumerate()` function with the index that starts from one:

```
In [4]:
1 cities = ['New York', 'Beijing', 'Cairo', 'Mumbai', 'Mexico']
2
3 for index, city in enumerate(cities,1):
4     print(f"{index}: {city}")
```

```
1: New York
2: Beijing
3: Cairo
4: Mumbai
5: Mexico
```

## 3.3 Break and Continue

In Python, break and continue statements can alter the flow of a normal loop.

Loops iterate over a block of code until the test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression.

The break and continue statements are used in these cases.

## 3.4 Break Statement

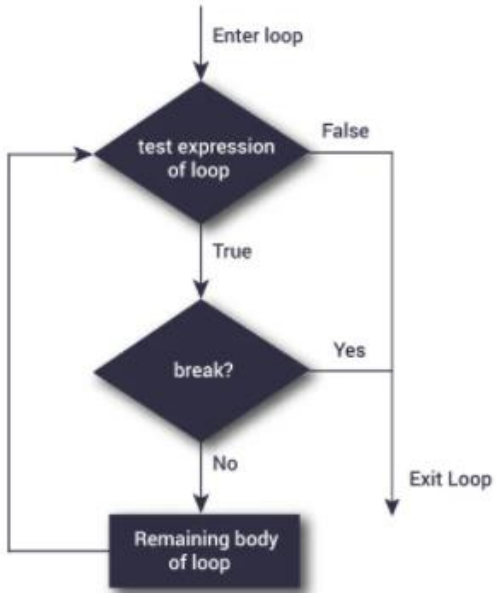
The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

If the break statement is inside a nested loop (loop inside another loop), the break statement will terminate the innermost loop.

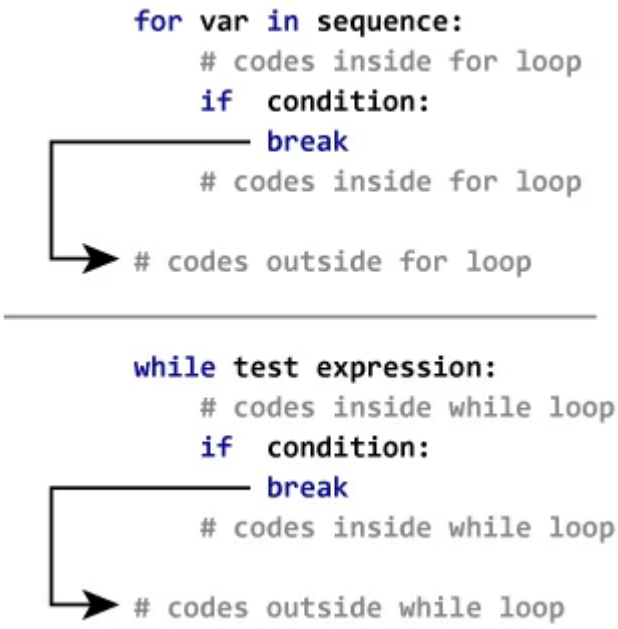
### 3.4.1 Syntax of break

Just use "`break`" in your program.

### 3.4.2 Flowchart of break



The working of break statement in for loop and while loop is shown below.




### 3.4.3 Example: Python break

```
[1]: # Use of break statement inside the loop

for val in "string":
    if val == "i":
        break
    print(val)

print("The end")
```

```
s
t
r
The end
```

In this program, we iterate through the "string" sequence. We check if the letter is i, upon which we break from the loop. Hence, we see in our output that all the letters up till i gets printed. After that, the loop terminates.

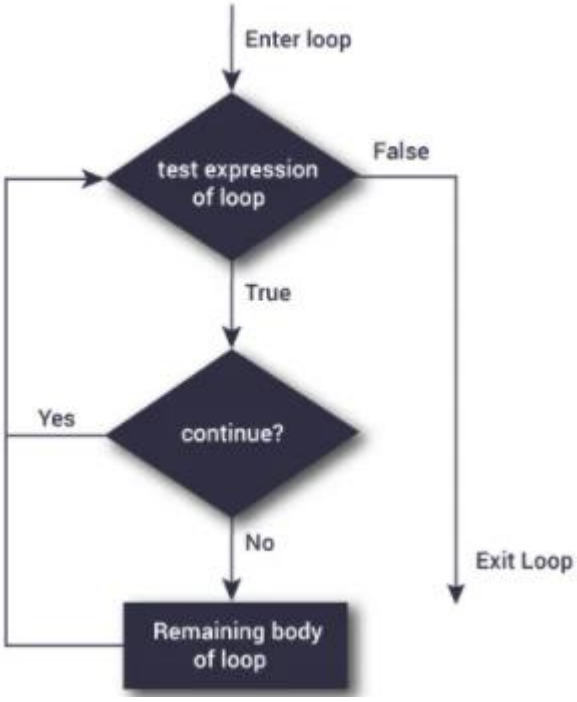
## 3.5 Continue statement

The `continue` statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

### 3.5.1 Syntax of Continue

Just use "`continue`" in your program.

### 3.5.2 Flowchart of continue




The working of the continue statement in for and while loop is shown below.

```
for var in sequence:
    # codes inside for loop
    if  condition:
        continue
    # codes inside for loop

# codes outside for loop
```

---

```
while test expression:
    # codes inside while loop
    if  condition:
        continue
    # codes  inside while loop

# codes outside while loop
```

### 3.5.3 Example: Python continue

```
[2]: # Program to show the use of continue statement inside loops

for val in "string":
    if val == "i":
        continue
    print(val)

print("The end")
```

```
s
t
r
n
g
The end
```

This program is same as the above example except the `break` statement has been replaced with `continue`.

We continue with the loop, if the string is i, not executing the rest of the block. Hence, we see in our output that all the letters except i gets printed.

## 3.6 Tasks:

# 4. Lab#04 - Dictionary in python

---

## Objectives:

- To learn and to use create, modify dictionary and access their elements
- To learn and able to use built-in functions of dictionary
- To learn and able to use the dictionary with loop

## Outcomes:

- Students should be able to create, modify dictionary and access their elements
- Students should be able to use built-in functions of dictionary in different programs.
- Students should be able to iterate the dictionary and able to use dictionary with loop

.

---

## 4.1 Dictionary

Dictionaries are Python's implementation of a data structure that is more generally known as an associative array. A dictionary consists of a collection of key-value pairs. Each key-value pair maps the key to its associated value.

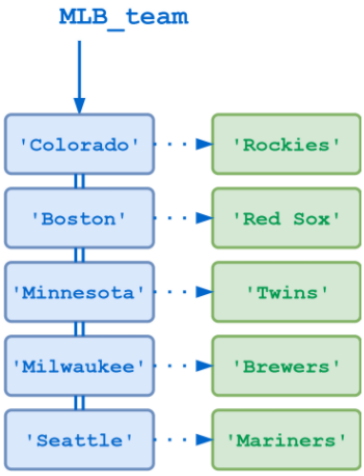
You can define a dictionary by enclosing a comma-separated list of key-value pairs in curly braces ({}). A colon (:) separates each key from its associated value:

```
d = {
    <key>: <value>,
    <key>: <value>,
      .
      .
      .
    <key>: <value>
}
```

The following defines a dictionary that maps a location to the name of its corresponding Major League Baseball team:

```
In [304]: MLB_teams ={
              'clorado':'rockies',
              'boston' :'red socks',
              'minnisota':'twins',
              'seattle':'mariners'
          }
```

---



Dictionary Mapping Location to MLB Team

You can also construct a dictionary with the built-in `dict()` function. The argument to `dict()` should be a sequence of key-value pairs. A list of tuples works well for this:

```
d = dict([
    (<key>, <value>),
    (<key>, <value),
      .
      .
      .
    (<key>, <value>)
])
```

MLB_team can then also be defined this way:

```
In [307]: MLB_teams =dict([
              ('clorado','rockies'),
              ('boston' ,'red socks'),
              ('minnisota','twins'),
             ('seattle','mariners')
          ])
```

If the key values are simple strings, they can be specified as keyword arguments. So here is yet another way to define MLB_team:

```
In [310]: MLB_teams =dict(
              clorado='rockies',
              boston='red socks',
              minnisota='twins',
             seattle='mariners'
          )
```

Once you've defined a dictionary, you can display its contents, the same as you can do for a list. All three of the definitions shown above appear as follows when displayed:

```
In [312]: type(MLB_teams)
Out[312]: dict

In [313]: MLB_teams
Out[313]: {'clorado': 'rockies',
           'boston': 'red socks',
           'minnisota': 'twins',
           'seattle': 'mariners'}
```

The entries in the dictionary display in the order they were defined. But that is irrelevant when it comes to retrieving them. Dictionary elements are not accessed by numerical index:

```
In [314]: MLB_teams[1]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-314-d43f6a177fce> in <module>
----> 1 MLB_teams[1]

KeyError: 1
```

## 4.2 Accessing Dictionary Values

Of course, dictionary elements must be accessible somehow. If you don't get them by index, then how do you get them?

A value is retrieved from a dictionary by specifying its corresponding key in square brackets ([]):

```
In [315]: MLB_teams['minnisota']
Out[315]: 'twins'

In [316]: MLB_teams['clorado']
Out[316]: 'rockies'
```

If you refer to a key that is not in the dictionary, Python raises an exception:

```
In [317]: MLB_teams['torronto']
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-317-076a7f75f4a7> in <module>
----> 1 MLB_teams['torronto']

KeyError: 'torronto'
```

Adding an entry to an existing dictionary is simply a matter of assigning a new key and value:

```
In [318]: MLB_teams['kensas city']='Royals'

In [319]: MLB_teams
Out[319]: {'clorado': 'rockies',
           'boston': 'red socks',
           'minnisota': 'twins',
           'seattle': 'mariners',
           'kensas city': 'Royals'}
```

If you want to update an entry, you can just assign a new value to an existing key:

```
In [320]: MLB_teams['clorado']='seaHawks'

In [321]: MLB_teams
Out[321]: {'clorado': 'seaHawks',
           'boston': 'red socks',
           'minnisota': 'twins',
           'seattle': 'mariners',
           'kensas city': 'Royals'}
```

To delete an entry, use the del statement, specifying the key to delete:

```
In [322]: del MLB_teams['clorado']

In [323]: MLB_teams
Out[323]: {'boston': 'red socks',
           'minnisota': 'twins',
           'seattle': 'mariners',
           'kensas city': 'Royals'}
```

## 4.3 Dictionary Keys vs. List Indices

You may have noticed that the interpreter raises the same exception, Key Error, when a dictionary is accessed with either an undefined key or by a numeric index:

```
In [324]: MLB_teams['torronto']
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-324-076a7f75f4a7> in <module>
----> 1 MLB_teams['torronto']

KeyError: 'torronto'

In [325]: MLB_teams[1]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-325-d43f6a177fce> in <module>
----> 1 MLB_teams[1]

KeyError: 1
```

In fact, it's the same error. In the latter case, [1] looks like a numerical index, but it isn't.

You will see later in this tutorial that an object of any immutable type can be used as a dictionary key. Accordingly, there is no reason you can't use integers:

```
In [326]: d={0:'a',1:'b',2:'c',3:'d'}
          d
Out[326]: {0: 'a', 1: 'b', 2: 'c', 3: 'd'}

In [327]: d[0]
Out[327]: 'a'

In [328]: d[2]
Out[328]: 'c'
```

In the expressions MLB_team[1], d[0], and d[2], the numbers in square brackets appear as though they might be indices. But they have nothing to do with the order of the items in the dictionary. Python is interpreting them as dictionary keys. If you define this same dictionary in reverse order, you still get the same values using the same keys:

```
In [329]: d={3:'d',2:'c',1:'b',0:'a'}
          d
Out[329]: {3: 'd', 2: 'c', 1: 'b', 0: 'a'}

In [330]: d[0]
Out[330]: 'a'

In [331]: d[2]
Out[331]: 'c'
```

The syntax may look similar, but you can't treat a dictionary like a list:

```
In [332]: type(d)
Out[332]: dict

In [333]: d[-1]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-333-1bab3c6dc314> in <module>
----> 1 d[-1]

KeyError: -1

In [334]: d[0:1]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-334-e181e8d8d20f> in <module>
----> 1 d[0:1]

TypeError: unhashable type: 'slice'
```

```
In [335]: d.append('apple')

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-335-93e59ce8b48f> in <module>
----> 1 d.append('apple')

AttributeError: 'dict' object has no attribute 'append'
```

## 4.4 Building a Dictionary Incrementally

Defining a dictionary using curly braces and a list of key-value pairs, as shown above is fine if you know all the keys and values in advance. But what if you want to build a dictionary on the fly?

You can start by creating an empty dictionary, which is specified by empty curly braces. Then you can add new keys and values one at a time:

```
In [336]: person={}
          type(person)

Out[336]: dict

In [337]: person['fname']='joe'
          person['lname']='fonbon'
          person['age']='21'
          person['pets']={'dog':'fido','cat':'sox'}
```

Once the dictionary is created in this way, its values are accessed the same way as any other dictionary:

```
In [338]: person
Out[338]: {'fname': 'joe',
           'lname': 'fonbon',
           'age': '21',
           'pets': {'dog': 'fido', 'cat': 'sox'}}

In [340]: person['fname']
Out[340]: 'joe'

In [341]: person['pets']
Out[341]: {'dog': 'fido', 'cat': 'sox'}
```

Retrieving the values in the sub list or sub dictionary requires an additional index or key:

```
In [342]: person['pets']['cat']

Out[342]: 'sox'
```

Just as the values in a dictionary don't need to be of the same type, the keys don't either:

```
In [343]: foo={42:'aaa',2.78:'bb',True:'ccc'}
          foo

Out[343]: {42: 'aaa', 2.78: 'bb', True: 'ccc'}
```

Here, one of the keys is an integer, one is a float, and one is a Boolean. It's not obvious how this would be useful, but you never know.

Notice how versatile Python dictionaries are. In MLB_team, the same piece of information (the baseball team name) is kept for each of several different geographical locations. person, on the other hand, stores varying types of data for a single person.

You can use dictionaries for a wide range of purposes because there are so few limitations on the keys and values that are allowed. But there are some. Read on!

## 4.5 Restrictions on Dictionary Keys

Almost any type of value can be used as a dictionary key in Python. You just saw this example, where integer, float, and Boolean objects are used as keys:

```
In [343]: foo={42:'aaa',2.78:'bb',True:'ccc'}
          foo

Out[343]: {42: 'aaa', 2.78: 'bb', True: 'ccc'}
```

You can even use built-in objects like types and functions:

```
In [344]: d={int:1,bool:2,float:3}

In [345]: d

Out[345]: {int: 1, bool: 2, float: 3}
```

However, there are a couple restrictions that dictionary keys must abide by.

First, a given key can appear in a dictionary only once. Duplicate keys are not allowed. A dictionary maps each key to a corresponding value, so it doesn't make sense to map a particular key more than once.

You saw above that when you assign a value to an already existing dictionary key, it does not add the key a second time, but replaces the existing value:

```
[38]: MLB_teams ={'clorado': 'seaHawks',
       'boston': 'red socks',
       'minnisota': 'twins',
       'seattle': 'mariners',
       'kensas city': 'Royals',
        'minnisota': 'timberwloves'
                 }

      MLB_teams['minnisota'] = 'Tinimina'
      MLB_teams

[38]: {'clorado': 'seaHawks',
       'boston': 'red socks',
       'minnisota': 'Tinimina',
       'seattle': 'mariners',
       'kensas city': 'Royals'}
```

Similarly, if you specify a key a second time during the initial creation of a dictionary, the second occurrence will override the first:

```
In [350]: MLB_teams ={'clorado': 'seaHawks',
           'boston': 'red socks',
           'minnisota': 'twins',
           'seattle': 'mariners',
           'kensas city': 'Royals',
            'minnisota': 'timberwloves'
                     }

In [351]: MLB_teams

Out[351]: {'clorado': 'seaHawks',
           'boston': 'red socks',
           'minnisota': 'timberwloves',
           'seattle': 'mariners',
           'kensas city': 'Royals'}
```

Secondly, a dictionary key must be of a type that is immutable. You have already seen examples where several of the immutable types you are familiar with—integer, float, string, and Boolean—have served as dictionary keys.

A tuple can also be a dictionary key, because tuples are immutable:

```
In [352]: d={(1,1):'a',(1,2):'b',(2,1):'c',(2,2):'d'}
          d[(1,1)]

Out[352]: 'a'

In [353]: d[(2,1)]

Out[353]: 'c'
```

However, neither a list nor another dictionary can serve as a dictionary key, because lists and dictionaries are mutable:

```
In [354]: d={[1,1]:'a',[1,2]:'b',[2,1]:'c',[2,2]:'d'}

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-354-19d9fe6e98ce> in <module>
----> 1 d={[1,1]:'a',[1,2]:'b',[2,1]:'c',[2,2]:'d'}

TypeError: unhashable type: 'list'
```

## 4.6 Operators and Built-in Functions

You have already become familiar with many of the operators and built-in functions that can be used with strings, lists, and tuples. Some of these work with dictionaries as well.

For example, the in and not in operators return True or False according to whether the specified operand occurs as a key in the dictionary:

```
In [355]: MLB_teams ={'clorado': 'seaHawks',
           'boston': 'red socks',
           'minnisota': 'twins',
           'seattle': 'mariners',
           'kensas city': 'Royals',
            'minnisota': 'timberwloves'
                      }

In [356]: 'boston' in MLB_teams
Out[356]: True

In [357]: 'torronto' in MLB_teams
Out[357]: False
```

:

In the second case, due to short-circuit evaluation, the expression `MLB_team['Toronto']` is not evaluated, so the KeyError exception does not occur.

The `len()` function returns the number of key-value pairs in a dictionary:

```
In [358]: len(MLB_teams)
Out[358]: 5
```

## 4.7 Built-in Dictionary Methods

As with strings and lists, there are several built-in methods that can be invoked on dictionaries. In fact, in some cases, the list and dictionary methods share the same name. (In the discussion on object-oriented programming, you will see that it is perfectly acceptable for different types to have methods with the same name.

The following is an overview of methods that apply to dictionaries:

- `MLB_teams.clear()` Clears a dictionary.

`MLB_teams.clear()` empties dictionary d of all key-value pairs:

```
In [359]: MLB_teams.clear()

In [360]: MLB_teams
Out[360]: {}
```

- `MLB_teams.get(<key>[, <default>])` Returns the value for a key if it exists in the dictionary.

The Python `dictionary .get()` method provides a convenient way of getting the value of a key from a dictionary without checking ahead of time whether the key exists, and without raising an error.

`MLB_teams.get(<key>)` searches dictionary d for <key> and returns the associated value if it is found. If <key> is not found, it returns None:

```
In [361]: MLB_teams ={'clorado': 'seaHawks',
           'boston': 'red socks',
           'minnisota': 'twins',
           'seattle': 'mariners',
           'kensas city': 'Royals',
            'minnisota': 'timberwloves'
                     }
          MLB_teams.get('clorado')

Out[361]: 'seaHawks'

In [362]: MLB_teams.get('boston')

Out[362]: 'red socks'

In [363]: MLB_teams.get('pak')

In [364]: print(MLB_teams.get('pak'))
          None
```

If <key> is not found and the optional <default> argument is specified, that value is returned instead of None:

→ `MLB_teams.items()` Returns a list of key-value pairs in a dictionary.

`MLB_teams.items()` returns a list of tuples containing the key-value pairs in mlb_teams. The first item in each tuple is the key, and the second item is the key's value:

```
In [365]: MLB_teams ={'clorado': 'seaHawks',
           'boston': 'red socks',
           'minnisota': 'twins',
           'seattle': 'mariners',
           'kensas city': 'Royals',
            'minnisota': 'timberwloves'
                     }

In [366]: list(MLB_teams.items())

Out[366]: [('clorado', 'seaHawks'),
           ('boston', 'red socks'),
           ('minnisota', 'timberwloves'),
           ('seattle', 'mariners'),
           ('kensas city', 'Royals')]

In [367]: list(MLB_teams.items())[1][0]

Out[367]: 'boston'

In [368]: list(MLB_teams.items())[1][1]

Out[368]: 'red socks'
```

- `MLB_teams.keys()` Returns a list of keys in a dictionary.

`MLB_teams.keys()` returns a list of all keys in MLB_teams:

```
In [369]: MLB_teams ={'clorado': 'seaHawks',
           'boston': 'red socks',
           'minnisota': 'twins',
           'seattle': 'mariners',
           'kensas city': 'Royals',
            'minnisota': 'timberwloves'
                     }
          list(MLB_teams.keys())

Out[369]: ['clorado', 'boston', 'minnisota', 'seattle', 'kensas city']
```

- `MLB_teams.values()` Returns a list of values in a dictionary.

`MLB_teams.values()` returns a list of all values in MLB_teams:

```
In [370]: MLB_teams ={'clorado': 'seaHawks',
           'boston': 'red socks',
           'minnisota': 'twins',
           'seattle': 'mariners',
           'kensas city': 'Royals',
            'minnisota': 'timberwloves'
                     }
          list(MLB_teams.values())

Out[370]: ['seaHawks', 'red socks', 'timberwloves', 'mariners', 'Royals']
```

Any duplicate values in d will be returned as many times as they occur:

```
In [371]: #duplicate values
          d={3:'d',2:'d',1:'d',0:'d'}
          list(d.values())

Out[371]: ['d', 'd', 'd', 'd']
```

- `d.pop(<key>[, <default>])` Removes a key from a dictionary, if it is present, and returns its value.

If <key> is present in d, `d.pop(<key>)` removes <key> and returns its associated value:

```
In [372]: d={3:'a',2:'b',1:'c',0:'d'}
          d.pop(3)

Out[372]: 'a'

In [373]: d

Out[373]: {2: 'b', 1: 'c', 0: 'd'}
```

`d.pop(<key>)` raises a KeyError exception if <key> is not in d:

```
In [374]: d.pop('z')

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-374-693100771e54> in <module>
----> 1 d.pop('z')

KeyError: 'z'
```

If <key> is not in d, and the optional <default> argument is specified, then that value is returned, and no exception is raised:

```
In [375]: d={'a':10,'b':20,'c':30}

In [376]: d.pop('z',-1)

Out[376]: -1

In [377]: d

Out[377]: {'a': 10, 'b': 20, 'c': 30}
```

- `d.popitem()` Removes a key-value pair from a dictionary.

`d.popitem()` removes the last key-value pair added from d and returns it as a tuple:

```
In [378]: d.popitem()
Out[378]: ('c', 30)

In [379]: d
Out[379]: {'a': 10, 'b': 20}
```

If d is empty, `d.popitem()` raises a KeyError exception:

```
In [380]: d={}

In [381]: d.popitem()
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call las
<ipython-input-381-83c64cff336b> in <module>
----> 1 d.popitem()

KeyError: 'popitem(): dictionary is empty'
```

- `d.update(<obj>)` Merges a dictionary with another dictionary or with an iterable of key-value pairs.

If <obj> is a dictionary, `d.update(<obj>)` merges the entries from <obj> into d. For each key in <obj>:

If the key is not present in d, the key-value pair from <obj> is added to d.

If the key is already present in d, the corresponding value in d for that key is updated to the value from <obj>.

Here is an example showing two dictionaries merged together:

```
In [383]: d1={'a':10,'b':20,'c':30}
          d2={'b':200,'d':400}

In [384]: d1.update(d2)
          d1
Out[384]: {'a': 10, 'b': 200, 'c': 30, 'd': 400}
```

In this example, key 'b' already exists in d1, so its value is updated to 200, the value for that key from d2. However, there is no key 'd' in d1, so that key-value pair is added from d2.

<obj> may also be a sequence of key-value pairs, similar to when the `dict()` function is used to define a dictionary. For example, <obj> can be specified as a list of tuples:

```
In [387]: d1={'a':10,'b':20,'c':30}
          d1.update([('b',200),('d',400)])
          d1
Out[387]: {'a': 10, 'b': 200, 'c': 30, 'd': 400}
```

Or the values to merge can be specified as a list of keyword arguments:

```
In [390]: d1={'a':10,'b':20,'c':30}
          d1.update(b=200,c=300)
          d1
Out[390]: {'a': 10, 'b': 200, 'c': 300}
```

## 4.8 Using Python for loop to iterate over a dictionary

[6]:
```
for i in d1.keys():
    print(i)

print('--------')

for i in d1.values():
    print(i)
```

```
a
b
ali
d
--------
10
20
30
[1, 2, 3]
```

[11]:
```
for key,val in d1.items():
    print(f'{key} -> {val} ')

print('\n -------- \n')

for key,val in d1.items():
    if type(val) == list:
        print(f'{key} -> {val} ')
```

```
a -> 10
b -> 20
ali -> 30
d -> [1, 2, 3]

 --------

d -> [1, 2, 3]
```

[19]:
```
print(d1)

for key,val in d1.items():
    if type(val) == int:
        d1[key] = val**2

print(d1)
```

```
{'a': 10, 'b': 20, 'ali': 30, 'd': [1, 2, 3]}
{'a': 100, 'b': 400, 'ali': 900, 'd': [1, 2, 3]}
```

## 4.9 Tasks:

# 5. Lab#05 - Functions in python

---

## Objectives:

- To learn and able to create function and return
- To learn and able to create function with positional arguments and keywords arguments
- To learn and able to create function with default parameters
- To learn and able to use argument tuple packing and unpacking
- To learn and able to use argument dictionary packing and unpacking

## Outcomes:

- Students should be able to create function and return.
- Students should be able to create function with positional arguments and keywords arguments
- Students should be able to create function with default parameters
- Students should be able to use argument tuple packing and unpacking
- Students should be able to use argument dictionary packing and unpacking

---

## 5.1 Functions

The usual syntax for defining a Python function is as follows:

```
def <function_name>([<parameters>]):
    <statement(s)>
```

The components of the definition are explained in the table below:

| Component | Meaning |
|---|---|
| `def` | The keyword that informs Python that a function is being defined |
| `<function_name>` | A valid Python identifier that names the function |
| `<parameters>` | An optional, comma-separated list of parameters that may be passed to the function |
| `:` | Punctuation that denotes the end of the Python function header (the name and parameter list) |
| `<statement(s)>` | A block of valid Python statements |

The final item, <statement(s)>, is called the body of the function. The body is a block of statements that will be executed when the function is called. The body of a Python function is defined by indentation in accordance with the off-side rule. This is the same as code blocks associated with a control structure, like an if or while statement.

The syntax for calling a Python function is as follows:

```
<function_name>([<arguments>])
```

<arguments> are the values passed into the function. They correspond to the <parameters> in the Python function definition. You can define a function that doesn't take any arguments, but the parentheses are still required. Both a function definition and a function call must always include parentheses, even if they're empty.

As usual, you'll start with a small example and add complexity from there. Keeping the time-honored mathematical tradition in mind, you'll call your first Python function `f()`. Here's a script file, foo.py, that defines and calls `f()`:

```
def f():
    s='inside f()'
    print(s)
print("before calling f()")
f()
print("after calling f()")
```

```
before calling f()
inside f()
after calling f()
```

Here's how this code works:

- Line 1 uses the def keyword to indicate that a function is being defined. Execution of the def statement merely creates the definition of `f()`. All the following lines that are indented (lines 2 to 3) become part of the body of `f()` and are stored as its definition, but they aren't executed yet.

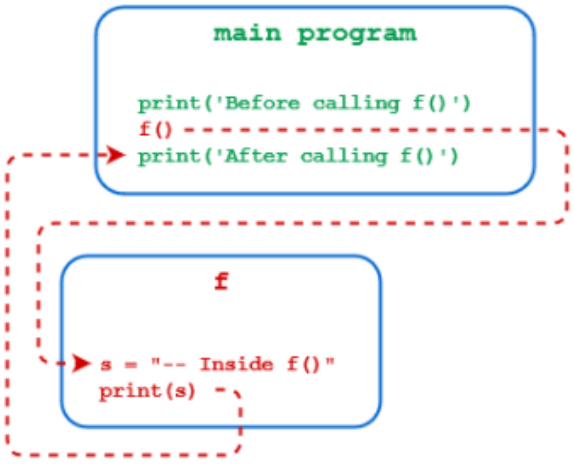
- Line 4 is a bit of whitespace between the function definition and the first line of the main program. While it isn't syntactically necessary, it is nice to have. To learn more about whitespace around top-level Python function definitions, check out Writing Beautiful Pythonic Code With PEP 8.

- Line 5 is the first statement that isn't indented because it isn't a part of the definition of `f()`. It's the start of the main program. When the main program executes, this statement is executed first.

- Line 6 is a call to `f()`. Note that empty parentheses are always required in both a function definition and a function call, even when there are no parameters or arguments. Execution proceeds to `f()` and the statements in the body of f() are executed.

- Line 7 is the next line to execute once the body of `f()` has finished. Execution returns to this `print()` statement.

The sequence of execution (or control flow) for foo.py is shown in the following diagram:




Occasionally, you may want to define an empty function that does nothing. This is referred to as a stub, which is usually a temporary placeholder for a Python function that will be fully implemented at a later time. Just as a block in a control structure can't be empty, neither can the body of a function. To define a stub function, use the pass statement:

```
In [392]: def f():
              pass
          f()
```

As you can see above, a call to a stub function is syntactically valid but doesn't do anything.

## 5.2 Argument Passing

So far in this tutorial, the functions you've defined haven't taken any arguments. That can sometimes be useful, and you'll occasionally write such functions. More often, though, you'll want to pass data into a function so that its behavior can vary from one invocation to the next. Let's see how to do that.

### 5.2.1 Positional Arguments

The most straightforward way to pass arguments to a Python function is with positional arguments (also called required arguments). In the function definition, you specify a comma-separated list of parameters inside the parentheses:

```
In [ ]: def f(qty,items,price):
            print(f'{qty}{items} cost ${price:.2f}')
```

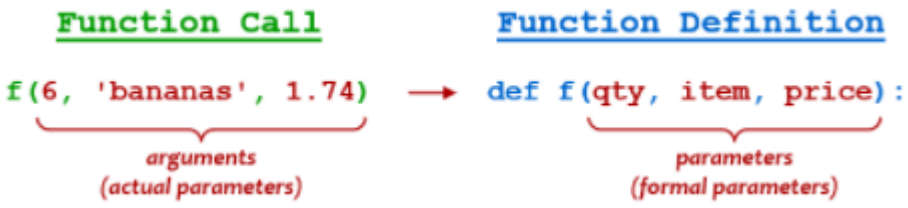
When the function is called, you specify a corresponding list of arguments:

```
In [394]: f(6,'bananas',1.75)
          6bananas cost $1.75
```

The parameters (qty, item, and price) behave like variables that are defined locally to the function. When the function is called, the arguments that are passed (6, 'bananas', and 1.74) are bound to the parameters in order, as though by variable assignment:

| Parameter | | Argument |
|---|---|---|
| qty | ← | 6 |
| item | ← | bananas |
| price | ← | 1.74 |

In some programming texts, the parameters given in the function definition are referred to as **formal parameters**, and the arguments in the function call are referred to as **actual parameters**:




Although positional arguments are the most straightforward way to pass data to a function, they also afford the least flexibility. For starters, the order of the arguments in the call must match the order of the parameters in the definition. There's nothing to stop you from specifying positional arguments out of order, of course:

```
In [394]: f(6,'bananas',1.75)

          6bananas cost $1.75
```

The function may even still run, as it did in the example above, but it's very unlikely to produce the correct results. It's the responsibility of the programmer who defines the function to document what the appropriate arguments should be, and it's the responsibility of the user of the function to be aware of that information and abide by it.

With positional arguments, the arguments in the call and the parameters in the definition must agree not only in order but in number as well. That's the reason positional arguments are also referred to as required arguments. You can't leave any out when calling the function:

```
In [395]: #too few arguments
          f(6,'bananas')

          ---------------------------------------------------------------------------
          TypeError                                 Traceback (most recent call last)
          <ipython-input-395-5c87b4378311> in <module>
                1 #too few arguments
          ----> 2 f(6,'bananas')

          TypeError: f() missing 1 required positional argument: 'price'
```

Nor can you specify extra ones:

```
In [396]: #too many argument
          f(6,'bananas',1.75,4)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-396-e7ac2c7b9869> in <module>
      1 #too many argument
----> 2 f(6,'bananas',1.75,4)

TypeError: f() takes 3 positional arguments but 4 were given
```

Positional arguments are conceptually straightforward to use, but they're not very forgiving. You must specify the same number of arguments in the function call as there are parameters in the definition, and in exactly the same order. In the sections that follow, you'll see some argument-passing techniques that relax these restrictions.

### 5.2.2 Keyword Arguments

When you're calling a function, you can specify arguments in the form <keyword>=<value>. In that case, each <keyword> must match a parameter in the Python function definition. For example, the previously defined function `f()` may be called with **keyword arguments** as follows:

```
In [398]: f(qty=6,items='bananas',price=1.75)
          6bananas cost $1.75
```

Referencing a keyword that doesn't match any of the declared parameters generates an exception:

```
In [399]: #wrong arg
          f(qty=6,item='bananas',price=1.75)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-399-830a85f627bf> in <module>
      1 #wrong arg
----> 2 f(qty=6,item='bananas',price=1.75)

TypeError: f() got an unexpected keyword argument 'item'
```

Using keyword arguments lifts the restriction on argument order. Each keyword argument explicitly designates a specific parameter by name, so you can specify them in any order and Python will still know which argument goes with which parameter:

```
In [401]: f(items='bananas',price=1.75,qty=6)
          6bananas cost $1.75
```

Like with positional arguments, though, the number of arguments and parameters must still match:

```
In [402]: #too few args
          f(items='bananas',price=1.75)

          ---------------------------------------------------------------------------
          TypeError                                 Traceback (most recent call last)
          <ipython-input-402-ca72c721db1c> in <module>
                1 #too few args
          ----> 2 f(items='bananas',price=1.75)

          TypeError: f() missing 1 required positional argument: 'qty'
```

So, keyword arguments allow flexibility in the order that function arguments are specified, but the number of arguments is still rigid.

You can call a function using both positional and keyword arguments:

```
In [404]: f(6,price=1.75,items='banana')

          6banana cost $1.75
```

When positional and keyword arguments are both present, all the positional arguments must come first:

```
In [406]: #positional args follow key
          f(6,items='banana',1.75)

            File "<ipython-input-406-66445a69f0ea>", line 2
              f(6,items='banana',1.75)
                                 ^
          SyntaxError: positional argument follows keyword argument
```

Once you've specified a keyword argument, there can't be any positional arguments to the right of it.

### 5.2.3 Default Parameters

If a parameter specified in a Python function definition has the form <name>=<value>, then <value> becomes a default value for that parameter. Parameters defined this way are referred to as **default or optional parameters**. An example of a function definition with default parameters is shown below:

```
In [407]: #default perameter
          def f(qty=6,items='banana',price=17.4):
              print(f'{qty}{items} cost ${price:.2f}')
```

When this version of `f()` is called, any argument that's left out assumes its default value:

```
In [407]: #default perameter
          def f(qty=6,items='banana',price=17.4):
              print(f'{qty}{items} cost ${price:.2f}')

In [408]: f(4,'apples',2.4)

          4apples cost $2.40

In [409]: f('grapes',3.5)

          grapes3.5 cost $17.40

In [410]: f(9,2.4)

          92.4 cost $17.40
```

### 5.2.4 Mutable Default Parameter Values

Things can get weird if you specify a default parameter value that is a mutable object. Consider this Python function definition:

```
In [411]: def f(my_list=[]):
              my_list.append("###")
              return my_list
```

f() takes a single list parameter, appends the string '###' to the end of the list, and returns the result:

```
In [412]: f(['foo','bar','baz'])
Out[412]: ['foo', 'bar', 'baz', '###']

In [413]: f([1,2,3,4,5])
Out[413]: [1, 2, 3, 4, 5, '###']
```

The default value for parameter my_list is the empty list, so if `f()` is called without any arguments, then the return value is a list with the single element '###':

```
In [414]: f()
Out[414]: ['###']
```

Everything makes sense so far. Now, what would you expect to happen if `f()` is called without any parameters a second and a third time? Let's see:

```
In [414]: f()
Out[414]: ['###']

In [415]: f()
Out[415]: ['###', '###']

In [416]: f()
Out[416]: ['###', '###', '###']
```

Oops! You might have expected each subsequent call to also return the singleton list ['###'], just like the first. Instead, the return value keeps growing. What happened?

In Python, default parameter values are defined only once when the function is defined (that is, when the def statement is executed). The default value isn't re-defined each time the function is called. Thus, each time you call `f()` without a parameter, you're performing `.append()` on the same list.

You can demonstrate this with `id()`:

```
In [419]: def f(my_list=[]):
              print(id(my_list))
              my_list.append("###")
              return my_list

In [420]: f()
          2259583565632
Out[420]: ['###']

In [421]: f()
          2259583565632
Out[421]: ['###', '###']
```

The object identifier displayed confirms that, when my_list is allowed to default, the value is the same object with each call. Since lists are mutable, each subsequent `.append()` call causes the list to get longer. This is a common and pretty well-documented pitfall when you're using a mutable object as a parameter's default value. It potentially leads to confusing code behavior, and is probably best avoided.

As a workaround, consider using a default argument value that signals no argument has been specified. Most any value would work, but None is a common choice. When the sentinel value indicates no argument is given, create a new empty list inside the function:

```
In [437]: def f(my_list=None):
              if my_list is None:
                  my_list=[]
              my_list.append("###")
              return my_list

In [438]: f()
Out[438]: ['###']

In [439]: f()
Out[439]: ['###']

In [440]: f(['foo','bar','baz'])
Out[440]: ['foo', 'bar', 'baz', '###']
```

## 5.3 The return Statement

What's a Python function to do then? After all, in many cases, if a function doesn't cause some change in the calling environment, then there isn't much point in calling it at all. How should a function affect its caller?

Well, one possibility is to use function return values. A return statement in a Python function serves two purposes:

It immediately terminates the function and passes execution control back to the caller.

It provides a mechanism by which the function can pass data back to the caller.

### 5.3.1 Exiting a Function

Within a function, a return statement causes immediate exit from the Python function and transfer of execution back to the caller:

```
In [441]: def f():
              print("foo")
              print("bar")
          f()
          foo
          bar
```

In this example, the return statement is actually superfluous. A function will return to the caller when it falls off the end—that is, after the last statement of the function body is executed. So, this function would behave identically without the return statement.

However, return statements don't need to be at the end of a function. They can appear anywhere in a function body, and even multiple times. Consider this example:

```
In [442]: def f(x):
              if x<0:
                  return
              if x>0:
                  return
              print(x)

          f(6)
          f(65)
          f(2.45)
```

The first two calls to `f()` don't cause any output, because a return statement is executed and the function exits prematurely, before the `print()` statement on line 6 is reached.

This sort of paradigm can be useful for error checking in a function. You can check several error conditions at the start of the function, with return statements that bail out if there's a problem:

```
def f():
    if error_cond1:
        return
    if error_cond2:
        return
    if error_cond3:
        return

    <normal processing>
```

If none of the error conditions are encountered, then the function can proceed with its normal processing.

### 5.3.2 Returning Data to the Caller

In addition to exiting a function, the return statement is also used to pass data back to the caller. If a return statement inside a Python function is followed by an expression, then in the calling environment, the function call evaluates to the value of that expression:

```
In [443]: def f():
              return 'foo'
          s=f()
          print(s)

          foo
```

Here, the value of the expression `f()` on line 5 is 'foo', which is subsequently assigned to variable s.

A function can return any type of object. In Python, that means pretty much anything whatsoever. In the calling environment, the function call can be used syntactically in any way that makes sense for the type of object the function returns.

For example, in this code, `f()` returns a dictionary. In the calling environment then, the expression `f()` represents a dictionary, and `f()['baz']` is a valid key reference into that dictionary:

```
In [445]: def f():
              return dict(foo=1,bar=2,baz=3)
          f()
Out[445]: {'foo': 1, 'bar': 2, 'baz': 3}
```

In the next example, f() returns a string that you can slice like any other string:

```
In [446]: def f():
              return  'foo bar'
          f()[2:4]
Out[446]: 'o '
```

Here, f() returns a list that can be indexed or sliced:

```
In [447]: def f():
              return ['foo','bar','baz']
          f()
Out[447]: ['foo', 'bar', 'baz']

In [448]: f()[2]
Out[448]: 'baz'

In [449]: f()[::-1]
Out[449]: ['baz', 'bar', 'foo']
```

If multiple comma-separated expressions are specified in a return statement, then they're packed and returned as a tuple:

```
In [450]: def f():
              return 'foo','bar','baz','quix'
          type(f())
Out[450]: tuple

In [451]: t=f()
          t
Out[451]: ('foo', 'bar', 'baz', 'quix')

In [452]: a=b=c=d=e=f=f()
          print(a,b,c,d,e,f)
          ('foo', 'bar', 'baz', 'quix') ('foo', 'bar', 'baz', 'quix') ('foo', 'bar', 'baz', 'quix') ('foo', 'bar', 'baz', 'quix') ('foo',
          'bar', 'baz', 'quix') ('foo', 'bar', 'baz', 'quix')
```

When no return value is given, a Python function returns the special Python value None:

```
In [453]: def f():
              return
          print(f())
          None
```

The same thing happens if the function body doesn't contain a return statement at all and the function falls off the end:

```
In [455]: def g():
              pass
          print(g())
          None
```

Recall that None is falsy when evaluated in a Boolean context.

Since functions that exit through a bare return statement or fall off the end return None, a call to such a function can be used in a Boolean context:

```
In [453]: def f():
              return
          print(f())
```

```
None
```

```
In [455]: def g():
              pass
          print(g())
```

```
None
```

```
In [457]: if f() or g():
              print("yes")
          else:
              print("no")
```

```
no
```

Here, calls to both `f()` and `g()` are falsy, so `f()` or `g()` is as well, and the else clause executes.

### 5.3.4 Revisiting Side Effects

Suppose you want to write a function that takes an integer argument and doubles it. That is, you want to pass an integer variable to the function, and when the function returns, the value of the variable in the calling environment should be twice what it was.

As you now know, Python integers are immutable, so a Python function can't change an integer argument by side effect:

```
In [463]: def double(x):
              x*=2
              print(x)
          x=5
          double(x)
```

```
10
```

However, you can use a return value to obtain a similar effect. Simply write `double()` so that it takes an integer argument, doubles it, and returns the doubled value. Then, the caller is responsible for the assignment that modifies the original value:

```
In [465]: def double(x):
              return x*2
              print(x)
          x=5
          double(x)
          x
```

```
Out[465]: 5
```

This is arguably preferable to modifying by side effect. It's very clear that x is being modified in the calling environment because the caller is doing so itself. Anyway, it's the only option, because modification by side effect doesn't work in this case.

Still, even in cases where it's possible to modify an argument by side effect, using a return value may still be clearer. Suppose you want to double every item in a list. Because lists are mutable, you could define a Python function that modifies the list in place:

```
In [467]: def double_list(x):
              i=0
              while i< len(x):
                  x[i] *=2
                  i+=1
          a=[1,2,3,4,5]
          double_list(a)
          a
Out[467]: [2, 4, 6, 8, 10]
```

Unlike `double()` in the previous example, `double_list()` actually works as intended. If the documentation for the function clearly states that the list argument's contents are changed, then this may be a reasonable implementation.

However, you can also write `double_list()` to pass the desired list back by return value and allow the caller to make the assignment, similar to how `double()` was re-written in the previous example:

```
In [468]: def double_list(x):
              r=[]
              for i in x:
                  r.append(i*2)
              return r

          a=[1,2,3,4,5]
          double_list(a)
          a
Out[468]: [1, 2, 3, 4, 5]
```

Either approach works equally well. As is often the case, this is a matter of style, and personal preferences vary. Side effects aren't necessarily consummate evil, and they have their place, but because virtually anything can be returned from a function, the same thing can usually be accomplished through return values as well.

## 5.4 Variable-Length Argument Lists

In some cases, when you're defining a function, you may not know beforehand how many arguments you'll want it to take. Suppose, for example, that you want to write a Python function that computes the average of several values. You could start with something like this:

```
In [470]: def avg(a,b,c):
              return (a+b+c)/3
```

All is well if you want to average three values:

```
In [471]: avg(1,2,3)
Out[471]: 2.0
```

However, as you've already seen, when positional arguments are used, the number of arguments passed must agree with the number of parameters declared. Clearly then, all isn't well with this implementation of `avg()` for any number of values other than three:

```
In [472]: avg(1,2,3,4)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-472-71776da85a6e> in <module>
----> 1 avg(1,2,3,4)

TypeError: avg() takes 3 positional arguments but 4 were given
```

You could try to define `avg()` with optional parameters:

```
In [474]: def avg(a,b=0,c=0,d=0,e=0):
              pass
```

This allows for a variable number of arguments to be specified. The following calls are at least syntactically correct:

```
In [ ]: avg(1)
        avg(1,2)
        avg(1,2,3)
        avg(1,2,3,4)
        avg(1,2,3,4,5)
```

But this approach still suffers from a couple of problems. For starters, it still only allows up to five arguments, not an arbitrary number. Worse yet, there's no way to distinguish between the arguments that were specified and those that were allowed to default. The function has no way to know how many arguments were actually passed, so it doesn't know what to divide by:

```
In [ ]: def avg(a,b=0,c=0,d=0,e=0):
            return(a+b+c+d+e)/#divided by what?
```

Evidently, this won't do either. You could write `avg()` to take a single list argument:

```
In [478]: def avg(a):
              total=0
              for v in a:
                  total +=v
              return total/len(a)
          avg([1,2,3])
          avg([1,2,3,4,5])
```

```
Out[478]: 3.0
```

At least this works. It allows an arbitrary number of values and produces a correct result. As an added bonus, it works when the argument is a tuple as well:

```
In [479]: t=(1,2,3,4,5)
          avg(t)
```

```
Out[479]: 3.0
```

The drawback is that the added step of having to group the values into a list or tuple is probably not something the user of the function would expect, and it isn't very elegant. Whenever you find Python code that looks inelegant, there's probably a better option.

In this case, indeed there is! Python provides a way to pass a function a variable number of arguments with argument tuple packing and unpacking using the asterisk (*) operator.

### 5.4.1 Argument Tuple Packing

When a parameter name in a Python function definition is preceded by an asterisk (*), it indicates argument tuple packing. Any corresponding arguments in the function call are packed into a tuple that the function can refer to by the given parameter name. Here's an example:

```
In [480]: def f(*args):
              print(args)
              print(type(args),len(args))
              for x in args:
                  print(x)
          f(1,2,3)
```

```
(1, 2, 3)
<class 'tuple'> 3
1
2
3
```

In the definition of `f()`, the parameter specification *args indicates tuple packing. In each call to `f()`, the arguments are packed into a tuple that the function can refer to by the name args. Any name can be used, but args is so commonly chosen that it's practically a standard.

Using tuple packing, you can clean up `avg()` like this:

```
In [483]: def avg (*args):
              total=0
              for i in args:
                  total +=i
              return total/len(args)
          avg(1,2,3)
```

```
Out[483]: 2.0
```

---

Better still, you can tidy it up even further by replacing the for loop with the built-in Python function `sum()`, which sums the numeric values in any iterable:

```
In [485]: def avg (*args):
              return sum(args)/len(args)
          avg(1,2,3)
```

```
Out[485]: 2.0
```

---

Now, `avg()` is concisely written and works as intended.

Still, depending on how this code will be used, there may still be work to do. As written, `avg()` will produce a TypeError exception if any arguments are non-numeric:

```
In [486]: #non numeric
          avg(1,2,'foo')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call l
<ipython-input-486-045654dedfbb> in <module>
      1 #non numeric
----> 2 avg(1,2,'foo')

<ipython-input-485-6e3be61b2fa0> in avg(*args)
      1 def avg (*args):
----> 2     return sum(args)/len(args)
      3 avg(1,2,3)
```

To be as robust as possible, you should add code to check that the arguments are of the proper type. Later in this tutorial series, you'll learn how to catch exceptions like TypeError and handle them appropriately.

### 5.4.2 Argument Tuple Unpacking

An analogous operation is available on the other side of the equation in a Python function call. When an argument in a function call is preceded by an asterisk (*), it indicates that the argument is a tuple that should be unpacked and passed to the function as separate values:

```
In [487]: def f (x,y,z):
              print(f'x={x}')
              print(f'y={y}')
              print(f'z={z}')
          f(1,2,3)

          x=1
          y=2
          z=3
```

```
In [488]: t=('foo','bar','baz')
          f(*t)

          x=foo
          y=bar
          z=baz
```

In this example, *t in the function call indicates that t is a tuple that should be unpacked. The unpacked values 'foo', 'bar', and 'baz' are assigned to the parameters x, y, and z, respectively.

Although this type of unpacking is called tuple unpacking, it doesn't only work with tuples. The asterisk (*) operator can be applied to any iterable in a Python function call. For example, a list or set can be unpacked as well:

```
In [489]: a=['foo','bar','baz']
          type(a)

Out[489]: list
```

```
In [490]: f(*a)

          x=foo
          y=bar
          z=baz
```

```
In [491]: s={1,2,3}
          f(*s)

          x=1
          y=2
          z=3
```

You can even use tuple packing and unpacking at the same time:

```
In [492]: def f (*args):
              print(type(args),args)
          a=['foo','bar','baz']
          f(*a)

          <class 'tuple'> ('foo', 'bar', 'baz')
```

Here, f(*a) indicates that list a should be unpacked and the items passed to f() as individual values. The parameter specification *args causes the values to be packed back up into the tuple args.

### 5.4.3 Argument Dictionary Packing

Python has a similar operator, the double asterisk (**), which can be used with Python function parameters and arguments to specify dictionary packing and unpacking. Preceding a parameter in a Python function definition by a double asterisk (**) indicates that the corresponding arguments, which are expected to be key=value pairs, should be packed into a dictionary:

```
In [495]: def f (**kwargs):
              print(kwargs)
              print(type(kwargs))
              for key,val in kwargs.items():
                  print(key,'->',val)
          f(foo=1,bar=2,baz=3)

          {'foo': 1, 'bar': 2, 'baz': 3}
          <class 'dict'>
          foo -> 1
          bar -> 2
          baz -> 3
```

In this case, the arguments `foo=1, bar=2`, and `baz=3` are packed into a dictionary that the function can reference by the name kwargs. Again, any name can be used, but the peculiar kwargs (which is short for keyword args) is nearly standard. You don't have to adhere to it, but if you do, then anyone familiar with Python coding conventions will know straightaway what you mean.

### 5.4.4 Argument Dictionary Unpacking

Argument dictionary unpacking is analogous to argument tuple unpacking. When the double asterisk (**) precedes an argument in a Python function call, it specifies that the argument is a dictionary that should be unpacked, with the resulting items passed to the function as keyword arguments:

```
In [499]: def f(a,b,c):
              print(F'a={a}')
              print(F'b={b}')
              print(F'c={c}')
          d={'a':'foo','b':25,'c':'qux'}
          f(**d)

          a=foo
          b=25
          c=qux
```

The items in the dictionary d are unpacked and passed to f() as keyword arguments. So, f(**d) is equivalent to `f(a='foo', b=25, c='qux')`:

```
In [500]: f(a='foo',b=25,c='qux')

          a=foo
          b=25
          c=qux
```

In fact, check this out:

```
In [501]: f(**dict(a='foo',b=25,c='qux'))

          a=foo
          b=25
          c=qux
```

Here, `dict(a='foo', b=25, c='qux')` creates a dictionary from the specified key/value pairs. Then, the double asterisk operator (**) unpacks it and passes the keywords to `f()`.

### 5.4.5 Putting It All Together

Think of *args as a variable-length positional argument list, and **kwargs as a variable-length keyword argument list.

All three—standard positional parameters, *args, and **kwargs—can be used in one Python function definition. If so, then they should be specified in that order:

```
In [503]: def f(a,b,*args,**kwargs):
              print(F'a={a}')
              print(F'b={b}')
              print(F'args={args}')
              print(F'kwargs={kwargs}')
          f(1,2,'foo','bar','baz','qux',x=100,y=200,z=300)

          a=1
          b=2
          args=('foo', 'bar', 'baz', 'qux')
          kwargs={'x': 100, 'y': 200, 'z': 300}
```

This provides just about as much flexibility as you could ever need in a function interface!

### 5.4.6 Multiple Unpacking's in a Python Function Call

Python version 3.5 introduced support for additional unpacking generalizations, as outlined in PEP 448. One thing these enhancements allow is multiple unpacking's in a single Python function call:

```
In [506]: def f(*args):
              for i in args:
                  print(i)
          a=[1,2,3]
          t=(4,5,6)
          s={7,8,9}
          f(*a,*t,*s)

          1
          2
          3
          4
          5
          6
          8
          9
          7
```

You can specify multiple dictionary unpackings in a Python function call as well:

```
Python                                                    >>>
>>> def f(**kwargs):
...     for k, v in kwargs.items():
...             print(k, '->', v)
...

>>> d1 = {'a': 1, 'b': 2}
>>> d2 = {'x': 3, 'y': 4}

>>> f(**d1, **d2)
a -> 1
b -> 2
x -> 3
y -> 4
```

By the way, the unpacking operators * and ** don't apply only to variables, as in the examples above. You can also use them with literals that are iterable:

```
In [507]: def f (*args):
              for i in args:
                  print(i)
          f(*[1,2,3],*[4,5,6])

1
2
3
4
5
6
```

```
In [508]: def f(**kwargs):
              for k,v in kwargs.items():
                  print(k,'->',v)
          f(**{'a':1,'b':2},**{'x':3,'y':4})

a -> 1
b -> 2
x -> 3
y -> 4
```

Here, the literal lists `[1, 2, 3]` and `[4, 5, 6]` are specified for tuple unpacking, and the literal dictionaries `{'a': 1, 'b': 2}` and `{'x': 3, 'y': 4}` are specified for dictionary unpacking.

## 5.5 Docstrings

When the first statement in the body of a Python function is a string literal, it's known as the function's docstring. A docstring is used to supply documentation for a function. It can contain the function's purpose, what arguments it takes, information about return values, or any other information you think would be useful.

The following is an example of a function definition with a docstring:

```
In [ ]:
def avg(*args):
    return sum(args) /len(args)
```

Technically, docstrings can use any of Python's quoting mechanisms, but the recommended convention is to triple-quote using double-quote characters ("""), as shown above. If the docstring fits on one line, then the closing quotes should be on the same line as the opening quotes.

Multi-line docstrings are used for lengthier documentation. A multi-line docstring should consist of a summary line, followed by a blank line, followed by a more detailed description. The closing quotes should be on a line by themselves:

```
def foo(bar=0,baz=1):
    pass
```

Docstring formatting and semantic conventions are detailed in PEP 257.

When a docstring is defined, the Python interpreter assigns it to a special attribute of the function called `__doc__`. This attribute is one of a set of specialized identifiers in Python that are sometimes called magic attributes or magic methods because they provide special language functionality.

You can access a function's docstring with the expression <function_name>.__doc__. The docstrings for the above examples can be displayed as follows:

```
print(avg.__doc__)
None

In [512]:
print(foo.__doc__)
None
```

In the interactive Python interpreter, you can type help(<function_name>) to display the docstring for <function_name>:

```
help(avg)
Help on function avg in module __main__:

avg(*args)


In [514]:
help(foo)
Help on function foo in module __main__:

foo(bar=0, baz=1)
```

## 5.6 lambda function

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

The identity function, a function that returns its argument, is expressed with a standard Python function definition using the keyword def as follows:

```
In [515]: def identity(x):
              return x
```

`identity()` takes an argument x and returns it upon invocation.

In contrast, if you use a Python lambda construction, you get the following:

```
In [516]: lambda x:x

Out[516]: <function __main__.<lambda>(x)>
```

In the example above, the expression is composed of:

- **The keyword:** lambda
- **A bound variable:** x
- **A body:** x

You can write a slightly more elaborated example, a function that adds 1 to an argument, as follows:

```
In [517]: lambda x:x+1

Out[517]: <function __main__.<lambda>(x)>
```

You can apply the function above to an argument by surrounding the function and its argument with parentheses:

```
In [518]: (lambda x:x+1)(2)

Out[518]: 3
```

Because a lambda function is an expression, it can be named. Therefore you could write the previous code as follows:

```
In [520]: add_one=(lambda x:x+1)(2)
          add_one

Out[520]: 3
```

The above lambda function is equivalent to writing this:

```
In [ ]: def add_one(x):
            return (x)
```

These functions all take a single argument. You may have noticed that, in the definition of the lambdas, the arguments don't have parentheses around them. Multi-argument functions (functions that take more than one argument) are expressed in Python lambdas by listing arguments and separating them with a comma (,) but without surrounding them with parentheses:

```
In [523]: full_name=lambda first,last:f'Full name: {first.title()}{last.title()}'
          full_name('guido','van rossum')

Out[523]: 'Full name: GuidoVan Rossum'
```

The lambda function assigned to full_name takes two arguments and returns a string interpolating the two parameters first and last. As expected, the definition of the lambda lists the arguments with no parentheses, whereas calling the function is done exactly like a normal Python function, with parentheses surrounding the arguments.

## 5.7 Built-in Function

### 5.7.1 map()

`map()` loops over the items of an input iterable (or iterables) and returns an iterator that results from applying a transformation function to every item in the original input iterable.

According to the documentation, `map()` takes a function object and an iterable (or multiple iterables) as arguments and returns an iterator that yields transformed items on demand. The function's signature is defined as follows:

```
map(function, iterable[, iterable1, iterable2,..., iterableN])
```

`map()` applies function to each item in iterable in a loop and returns a new iterator that yields transformed items on demand. function can be any Python function that takes a number of arguments equal to the number of iterables you pass to `map()`.

This first argument to `map()` is a **transformation function**. In other words, it's the function that transforms each original item into a new (transformed) item.

The operation that `map()` performs is commonly known as a **mapping** because it maps every item in an input iterable to a new item in a resulting iterable. To do that, `map()` applies a transformation function to all the items in the input iterable.

To better understand `map()`, suppose you need to take a list of numeric values and transform it into a list containing the square value of every number in the original list. In this case, you can use a for loop and code something like this:

```
In [524]: numbers=[1,2,3,4,5]
          squared=[]
          for num in numbers:
              squared.append(num**2)
          squared
Out[524]: [1, 4, 9, 16, 25]
```

When you run this loop on numbers, you get a list of square values. The for loop iterates over numbers and applies a power operation on each value. Finally, it stores the resulting values in squared.

You can achieve the same result without using an explicit loop by using `map()`. Take a look at the following reimplementation of the above example:

```
In [527]: def squared(number):
              return number **2
          numbers=[1,2,3,4,5]
          squared=map(squared,numbers)
          list(squared)
Out[527]: [1, 4, 9, 16, 25]
```

`square()` is a transformation function that maps a number to its square value. The call to `map()` applies `square()` to all of the values in numbers and returns an iterator that yields square values. Then you call `list()` on `map()` to create a list object containing the square values.

Since `map()` is written in C and is highly optimized, its internal implied loop can be more efficient than a regular Python for loop. This is one advantage of using `map()`.

A second advantage of using `map()` is related to memory consumption. With a for loop, you need to store the whole list in your system's memory. With `map()`, you get items on demand, and only one item is in your system's memory at a given time.

## 5.8 Tasks:

# 6. Lab#06 - OOP in python

---

## Objectives:

- To learn and able to create class, constructor and object
- To learn and able to create instance variable, class variable and member functions
- To learn and able to use encapsulation (public, private and protected)
- To learn and able to use setter and getter

## Outcomes:

- Students should be able to create class, constructor and object
- Students should be able to create instance variable, class variable and member functions
- Students should be able to use encapsulation (public, private and protected)
- Students should be able to use setter and getter

---

## 6.1 OOP

Object-oriented programming (OOP) is a method of structuring a program by bundling related properties and behaviors into individual objects. In this tutorial, you'll learn the basics of object-oriented programming in Python.

Conceptually, objects are like the components of a system. Think of a program as a factory assembly line of sorts. At each step of the assembly line a system component processes some material, ultimately transforming raw material into a finished product.

## 6.2 What Is Object-Oriented Programming in Python?

Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties and behaviors are bundled into individual objects.

For instance, an object could represent a person with properties like a name, age, and address and behaviors such as walking, talking, breathing, and running. Or it could represent an email with properties like a recipient list, subject, and body and behaviors like adding attachments and sending.

Put another way, object-oriented programming is an approach for modeling concrete, real-world things, like cars, as well as relations between things, like companies and employees, students and teachers, and so on. OOP models real-world entities as software objects that have some data associated with them and can perform certain functions.

Another common programming paradigm is procedural programming, which structures a program like a recipe in that it provides a set of steps, in the form of functions and code blocks, that flow sequentially in order to complete a task.

The key takeaway is that objects are at the center of object-oriented programming in Python, not only representing the data, as in procedural programming, but in the overall structure of the program as well.

## 6.3 Define a Class in Python

Primitive data structures—like numbers, strings, and lists—are designed to represent simple pieces of information, such as the cost of an apple, the name of a poem, or your favorite colors, respectively. What if you want to represent something more complex?

For example, let's say you want to track employees in an organization. You need to store some basic information about each employee, such as their name, age, position, and the year they started working.

One way to do this is to represent each employee as a list:

```
In [ ]: krik=["james krik",34,"captain",2265]
        spock=["spock",35,"science officer",2245]
        moccy=["leonard moccy","cheif medical officer",2265]
```

There are a number of issues with this approach.

First, it can make larger code files more difficult to manage. If you reference `kirk[0]` several lines away from where the kirk list is declared, will you remember that the element with index 0 is the employee's name?

Second, it can introduce errors if not every employee has the same number of elements in the list. In the mccoy list above, the age is missing, so `mccoy[1]` will return "Chief Medical Officer" instead of Dr. McCoy's age.

A great way to make this type of code more manageable and more maintainable is to use classes.

### 6.3.1 Classes vs Instances

Classes are used to create user-defined data structures. Classes define functions called methods, which identify the behaviors and actions that an object created from the class can perform with its data.

In this tutorial, you'll create a Dog class that stores some information about the characteristics and behaviors that an individual dog can have.

A class is a blueprint for how something should be defined. It doesn't actually contain any data. The Dog class specifies that a name and an age are necessary for defining a dog, but it doesn't contain the name or age of any specific dog.

While the class is the blueprint, an instance is an object that is built from a class and contains real data. An instance of the Dog class is not a blueprint anymore. It's an actual dog with a name, like Miles, who's four years old.

Put another way, a class is like a form or questionnaire. An instance is like a form that has been filled out with information. Just like many people can fill out the same form with their own unique information, many instances can be created from a single class.

### 6.3.2 How to Define a Class

All class definitions start with the class keyword, which is followed by the name of the class and a colon. Any code that is indented below the class definition is considered part of the class's body.

Here's an example of a Dog class:

```
In [ ]: class dog:
            pass
```

The body of the Dog class consists of a single statement: the pass keyword. pass is often used as a placeholder indicating where code will eventually go. It allows you to run this code without Python throwing an error.

The Dog class isn't very interesting right now, so let's spruce it up a bit by defining some properties that all Dog objects should have. There are a number of properties that we can choose from, including name, age, coat color, and breed. To keep things simple, we'll just use name and age.

The properties that all Dog objects must have are defined in a method called `.__init__()`. Every time a new Dog object is created, `.__init__()` sets the initial state of the object by assigning the values of the object's properties. That is, `.__init__()` initializes each new instance of the class.

You can give `.__init__()` any number of parameters, but the first parameter will always be a variable called self. When a new class instance is created, the instance is automatically passed to the self parameter in `.__init__()` so that new attributes can be defined on the object.

Let's update the Dog class with an `.__init__()` method that creates .name and .age attributes:

```
class dog:
    def __init__(self,name,age):
        self.name=name
        self.age=age
```

Notice that the `.__init__()` method's signature is indented four spaces. The body of the method is indented by eight spaces. This indentation is vitally important. It tells Python that the `.__init__()` method belongs to the Dog class.

In the body of `.__init__()`, there are two statements using the self variable:

- `self.name` = name creates an attribute called name and assigns to it the value of the name parameter.
- `self.age` = age creates an attribute called age and assigns to it the value of the age parameter.

Attributes created in `.__init__()` are called instance attributes. An instance attribute's value is specific to a particular instance of the class. All Dog objects have a name and an age, but the values for the name and age attributes will vary depending on the Dog instance.

On the other hand, class attributes are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of `.__init__()`.

For example, the following Dog class has a class attribute called species with the value "Canis familiaris":

```
class dog:
    species="canis familiars"
    def __init__(self,name,age):
        self.name=name
        self.age=age
```

Class attributes are defined directly beneath the first line of the class name and are indented by four spaces. They must always be assigned an initial value. When an instance of the class is created, class attributes are automatically created and assigned to their initial values.

Use class attributes to define properties that should have the same value for every class instance. Use instance attributes for properties that vary from one instance to another.

Now that we have a Dog class, let's create some dogs!

## 6.4 Instantiate an Object in Python

Open IDLE's interactive window and type the following:

```
In [ ]: class dog:
            pass
```

This creates a new Dog class with no attributes or methods.

Creating a new object from a class is called instantiating an object. You can instantiate a new Dog object by typing the name of the class, followed by opening and closing parentheses:

```
dog()

Out[534]:
<__main__.dog at 0x20e199c9cd0>
```

You now have a new Dog object at 0x106702d30. This funny-looking string of letters and numbers is a memory address that indicates where the Dog object is stored in your computer's memory. Note that the address you see on your screen will be different.

Now instantiate a second Dog object:

```
In [535]:
dog()

Out[535]:
<__main__.dog at 0x20e199c9df0>
```

The new Dog instance is located at a different memory address. That's because it's an entirely new instance and is completely unique from the first Dog object that you instantiated.

To see this another way, type the following:

```
In [536]:
a=dog()
b=dog()
a==b

Out[536]:
False
```

In this code, you create two new Dog objects and assign them to the variables a and b. When you compare a and b using the == operator, the result is False. Even though a and b are both instances of the Dog class, they represent two distinct objects in memory.

### 6.4.1 Class and Instance Attributes

Now create a new Dog class with a class attribute called .species and two instance attributes called .name and .age:

```
class dog:
    species="canis familiars"
    def __init__(self,name,age):
        self.name=name
        self.age=age
```

To instantiate objects of this Dog class, you need to provide values for the name and age. If you don't, then Python raises a TypeError:

```
class dog:
    species="canis familiars"
    def __init__(self,name,age):
        self.name=name
        self.age=age
dog()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-538-67debf56c6d9> in <module>
      4         self.name=name
      5         self.age=age
----> 6 dog()

TypeError: __init__() missing 2 required positional arguments: 'name' and 'age'
```

To pass arguments to the name and age parameters, put values into the parentheses after the class name:

```
In [540]:
class dog:
    species="canis familiars"
    def __init__(self,name,age):
        self.name=name
        self.age=age
buddy=dog("buddy",9)
miles=dog("miles",4)
```

This creates two new Dog instances—one for a nine-year-old dog named Buddy and one for a four-year-old dog named Miles.

The Dog class's `.__init__()` method has three parameters, so why are only two arguments passed to it in the example?

When you instantiate a Dog object, Python creates a new instance and passes it to the first parameter of `.__init__()`. This essentially removes the self parameter, so you only need to worry about the name and age parameters.

After you create the Dog instances, you can access their instance attributes using **dot notation**:

```
In [541]: buddy.name
Out[541]: 'buddy'

In [542]: miles.age
Out[542]: 4
```

You can access class attributes the same way:

```
In [543]: buddy.species
Out[543]: 'canis familiars'
```

One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect. All Dog instances have .species, .name, and .age attributes, so you can use those attributes with confidence knowing that they will always return a value.

Although the attributes are guaranteed to exist, their values can be changed dynamically:

```
In [544]: buddy.age=10
          buddy.age
Out[544]: 10

In [545]: miles.species='felis silves'
          miles.species
Out[545]: 'felis silves'
```

In this example, you change the .age attribute of the buddy object to 10. Then you change the .species attribute of the miles object to "Felis silvestris", which is a species of cat. That makes Miles a pretty strange dog, but it is valid Python!

The key takeaway here is that custom objects are mutable by default. An object is mutable if it can be altered dynamically. For example, lists and dictionaries are mutable, but strings and tuples are immutable.

### 6.4.2 Instance Methods

Instance methods are functions that are defined inside a class and can only be called from an instance of that class. Just like `.__init__()`, an instance method's first parameter is always self.

Open a new editor window in IDLE and type in the following Dog class:

```
In [546]: class dog:
              species="canis familiars"
              def __init__(self,name,age):
                  self.name=name
                  self.age=age
              def description(self):
                  return f"{self.name} is {self.age} years old"
              def speak (self ,sound):
                  return f"{self.name} says {sound}"
```

This Dog class has two instance methods:

- `.description()` returns a string displaying the name and age of the dog.
- `.speak()` has one parameter called sound and returns a string containing the dog's name and the sound the dog makes.

Save the modified Dog class to a file called dog.py and run the program. Then open the interactive window and type the following to see your instance methods in action:

```
In [552]: miles=dog("miles",4)

In [553]: miles.description()
Out[553]: 'miles is 4 years old'

In [549]: miles.speak("woof woof")
Out[549]: 'miles says woof woof'

In [550]: miles.speak("bow wow")
Out[550]: 'miles says bow wow'
```

In the above Dog class, `.description()` returns a string containing information about the Dog instance miles. When writing your own classes, it's a good idea to have a method that returns a string containing useful information about an instance of the class. However, `.description()` isn't the most Pythonic way of doing this.

When you create a list object, you can use `print()` to display a string that looks like the list:

```
In [554]: names=["fletcher","david","dan"]
          print(names)

          ['fletcher', 'david', 'dan']
```

Let's see what happens when you `print()` the miles object:

```
In [555]: print(miles)

          <__main__.dog object at 0x0000020E198BFA00>
```

When you print(miles), you get a cryptic looking message telling you that miles is a Dog object at the memory address 0x00aeff70. This message isn't very helpful. You can change what gets printed by defining a special instance method called `.__str__()`.

In the editor window, change the name of the Dog class's `.description()` method to `.__str__()`:

```
In [556]: class dog():
              def __str__(self):
                  return f"{self.name} is {self.age} years old"
```

Methods like `.__init__()` and `.__str__()` are called dunder methods because they begin and end with double underscores. There are many dunder methods that you can use to customize classes in Python. Although too advanced a topic for a beginning Python book, understanding dunder methods is an important part of mastering object-oriented programming in Python.

In the next section, you'll see how to take your knowledge one step further and create classes from other classes.

## 6.5 Encapsulation in Python

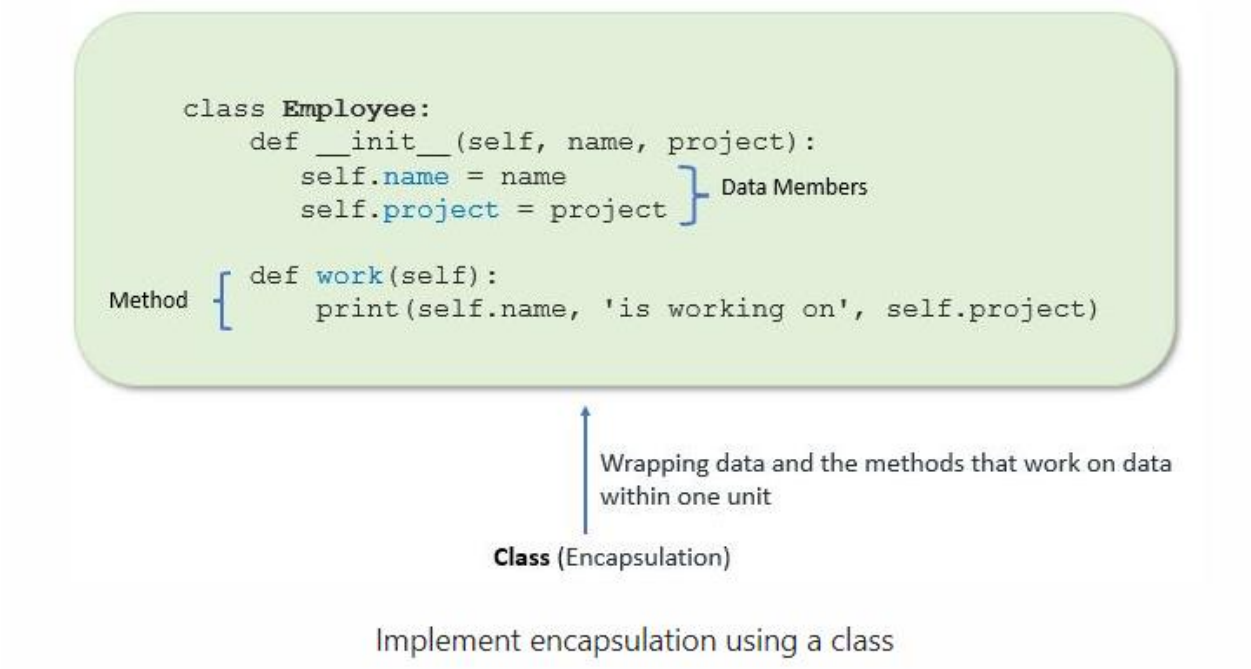
Encapsulation is one of the fundamental concepts in object-oriented programming (OOP), including abstraction, inheritance, and polymorphism. This lesson will cover what encapsulation is and how to implement it in Python.

**After reading this article, you will learn:**

- Encapsulation in Python
- Need for Encapsulation
- Data Hiding using public, protected, and private members
- Data Hiding vs. Encapsulation
- Getter and Setter Methods
- Benefits of Encapsulation

## 6.6 What is Encapsulation in Python?

Encapsulation in Python describes the concept of bundling data and methods within a single unit. So, for example, when you create a class, it means you are implementing encapsulation. A class is an example of encapsulation as it binds all the data members (instance variables) and methods into a single unit.




Implement encapsulation using a class

**Example:**

In this example, we create an Employee class by defining employee attributes such as name and salary as an instance variable and implementing behavior using `work()` and `show()` instance methods.

```
In [568]: class employee:
              def __init__(self,name,salary,project):
                      self.name=name
                      self.salary=salary
                      self.project=project
              def show(self):
                  print("name",self.name,'salary:',self.salary)
              def work(self):
                  print(self,name,'is working on',self.project)
          emp=employee('jessa',8000,'nlp')
          emp.show()
          emp.work()
```

```
name jessa salary: 8000
<__main__.employee object at 0x0000020E19B05BE0> Bob is working on nlp
```

Using encapsulation, we can hide an object's internal representation from the outside. This is called information hiding.

Also, encapsulation allows us to restrict accessing variables and methods directly and prevent accidental data modification by creating private data members and methods within a class.

Encapsulation is a way to can restrict access to methods and variables from outside of class. Whenever we are working with the class and dealing with sensitive data, providing access to all variables used within the class is not a good choice.

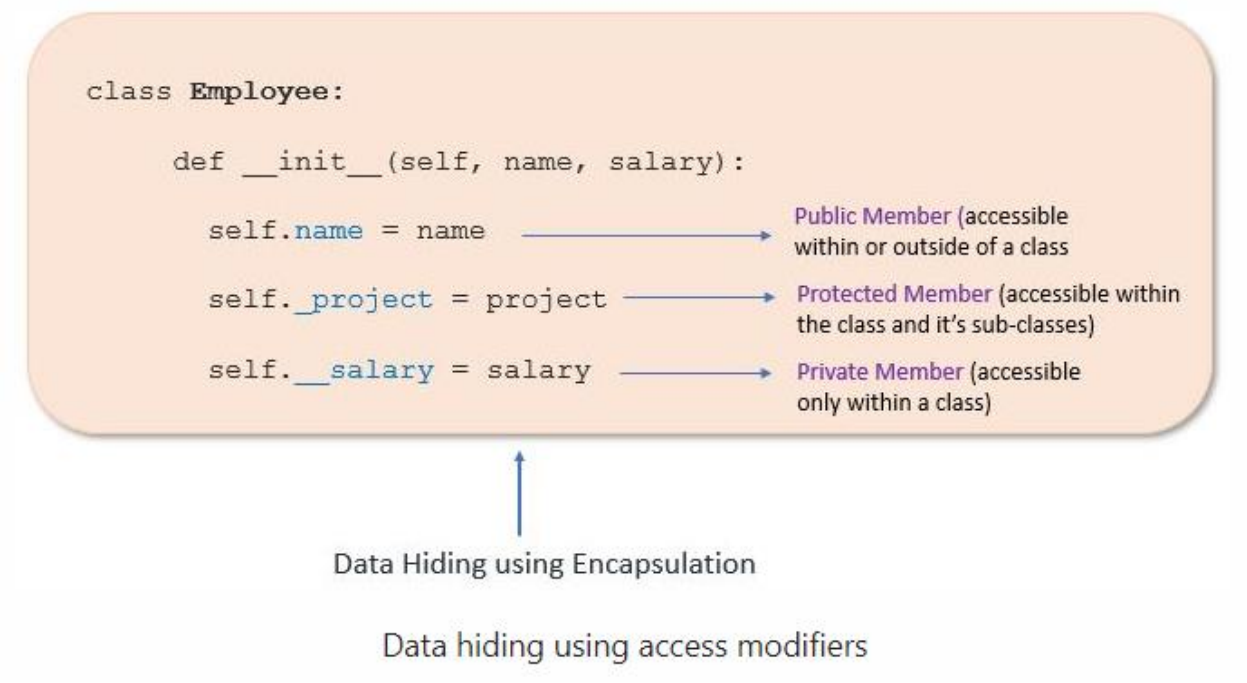
For example, Suppose you have an attribute that is not visible from the outside of an object and bundle it with methods that provide read or write access. In that case, you can hide specific information and control access to the object's internal state. Encapsulation offers a way for us to access the required variable without providing the program full-fledged access to all variables of a class. This mechanism is used to protect the data of an object from other objects.

## 6.7 Access Modifiers in Python

Encapsulation can be achieved by declaring the data members and methods of a class either as private or protected. But In Python, we don't have direct access modifiers like public, private, and protected. We can achieve this by using single **underscore** and **double underscores**.

Access modifiers limit access to the variables and methods of a class. Python provides three types of access modifiers private, public, and protected.

- **Public Member**: Accessible anywhere from outside class.
- **Private Member**: Accessible within the class
- **Protected Member**: Accessible within the class and its sub-classes




### 6.7.1 Public Member

Public data members are accessible within and outside of a class. All member variables of the class are by default public.

Example:

In [569]:
```
class employee:
    def __init__(self,name,salary,project):
        #public data members
            self.name=name
            self.salary=salary
            self.project=project
    def show(self):
        print("name",self.name,'salary:',self.salary)
    def work(self):
        print(self,name,'is working on',self.project)
emp=employee('jessa',8000,'nlp')
emp.show()
emp.work()
```

```
name jessa salary: 8000
<__main__.employee object at 0x0000020E19A97280> Bob is working on nlp
```

### 6.7.2 Private Member

We can protect variables in the class by marking them private. To define a private variable add two underscores as a prefix at the start of a variable name.

Private members are accessible only within the class, and we can't access them directly from the class objects.

Example:

```
In [570]: class employee:
              def __init__(self,name,salary,project):
                  #public data members
                      self.name=name
                  #private data member
                      self.__salary=salary
                      self.project=project
              def show(self):
                  print("name",self.name,'salary:',self.salary)
              def work(self):
                  print(self,name,'is working on',self.project)
          emp=employee('jessa',8000,'nlp')
          emp.show()
          emp.work()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-570-493625f5b7cc> in <module>
     11         print(self,name,'is working on',self.project)
     12 emp=employee('jessa',8000,'nlp')
---> 13 emp.show()
     14 emp.work()

<ipython-input-570-493625f5b7cc> in show(self)
      7             self.project=project
```

In the above example, the salary is a private variable. As you know, we can't access the private variable from the outside of that class.

We can access private members from outside of a class using the following two approaches

Create public method to access private members

Use **name mangling**

Let's see each one by one

### 6.7.3 Public method to access private members

Example: Access Private member outside of a class using an instance method

```
In [571]: class employee:
              def __init__(self,name,salary,project):
                  #public data members
                      self.name=name
                  #private data member
                      self.__salary=salary
                      self.project=project
              def show(self):
                  #acessing private member
                  print("name",self.name,'salary:',self.__salary)
              def work(self):
                  print(self,name,'is working on',self.project)
          emp=employee('jessa',8000,'nlp')
          emp.show()
          emp.work()
```

```
name jessa salary: 8000
<__main__.employee object at 0x0000020E198BF160> Bob is working on nlp
```

### 6.7.4 Name Mangling to access private members

We can directly access private and protected variables from outside of a class through name mangling. The name mangling is created on an identifier by adding two leading underscores and one trailing underscore, like this _classname__dataMember, where classname is the current class, and data member is the private variable name.

**Example:** Access private member

```
In [572]: class employee:
              def __init__(self,name,salary,project):
                  #public data members
                      self.name=name
                  #private data member
                      self.__salary=salary
                      self.project=project
              def show(self):
                  #acessing private member
                  print("name",self.name,'salary:',self.__salary)
              def work(self):
                  print(self,name,'is working on',self.project)
          emp=employee('jessa',8000,'nlp')
          emp.show()
          print('salary',emp.__employee__salary)
```

```
name jessa salary: 8000
```

### 6.7.5 Protected Member

Protected members are accessible within the class and also available to its sub-classes. To define a protected member, prefix the member name with a single underscore _.

Protected data members are used when you implement inheritance and want to allow data members access to only child classes.

Example: Proctecd member in inheritance.

```
In [578]: class company:
              def __init__(self):
                  self._project='nlp'
          class employee(company):
              def __init__(self,name):
                  self.name=name
                  company.__init__(self)
              def show(self):
                      print("employee name",self.name)
                      print("working on project",self._project)
          c=employee("jessa")
          c.show()
          print('project:',c._project)
```

```
employee name jessa
working on project nlp
project: nlp
```

## 6.8 Getters and Setters in Python

To implement proper encapsulation in Python, we need to use setters and getters. The primary purpose of using getters and setters in object-oriented programs is to ensure data encapsulation. Use the getter method to access data members and the setter methods to modify the data members.

In Python, private variables are not hidden fields like in other programming languages. The getters and setters methods are often used when:

- When we want to avoid direct access to private variables
- To add validation logic for setting a value

**Example**

```
In [580]: class student:
              def __init__ (self,name,age):
                  self.name=name
                  self.__age=age
              def get_age (self):
                  return self.__age
              def set_age(self,age):
                  self.__age=age
          stud=student('jessa',14)
          print('name',stud.name,stud.get_age())
```

```
name jessa 14
```

## 6.9 Advantages of Encapsulation

- **Security:** The main advantage of using encapsulation is the security of the data. Encapsulation protects an object from unauthorized access. It allows private and protected access levels to prevent accidental data modification.
- **Data Hiding:** The user would not be knowing what is going on behind the scene. They would only be knowing that to modify a data member, call the setter method. To read a data member, call the getter method. What these setter and getter methods are doing is hidden from them.
- **Simplicity:** It simplifies the maintenance of the application by keeping classes separated and preventing them from tightly coupling with each other.

- **Aesthetics:** Bundling data and methods within a class makes code more readable and maintainable

## 6.10 Tasks:

# 7. Lab#07 - OOP Concepts in python

---

## Objectives:

- To learn and able to use inheritance and type of inheritance
- To learn and able to use `super()` function
- To learn and able to use polymorphism (method overriding and operator overloading)

## Outcomes:

- Students should be able to use inheritance and type of inheritance
- Students should be able to use `super()` function
- Students should be able to use polymorphism (method overriding and operator overloading)

---

## 7.1 Inherit from Other Classes in Python

### 7.1.2 Types of Inheritance

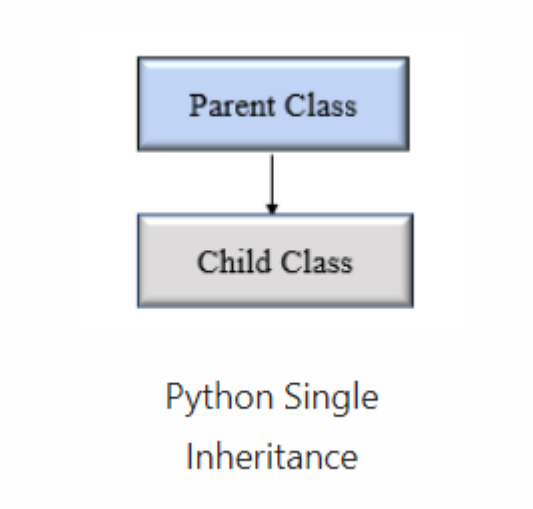
In Python, based upon the number of child and parent classes involved, there are five types of inheritance. The types of inheritance are listed below:

- Single inheritance
- Multiple Inheritance
- Multilevel inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Now let's see each in detail with an example.

### 7.1.3 Single Inheritance

In single inheritance, a child class inherits from a single-parent class. Here is one child class and one parent class.




Python Single Inheritance

**Example**

Let's create one parent class called ClassOne and one child class called ClassTwo to implement single inheritance.

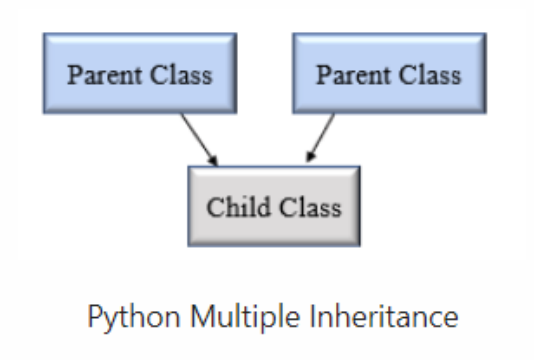
In [584]:
```
class vehicle:
    def vehicle_info(self):
        print("inside vehicle class")
class car(vehicle):
    def car_info(self):
        print("inside car class ")
c=car()
c.vehicle_info()
c.car_info
```

```
inside vehicle class
```

Out[584]:
```
<bound method car.car_info of <__main__.car object at 0x0000020E19AB6340>>
```

## 7.1.4 Multiple Inheritance

In multiple inheritance, one child class can inherit from multiple parent classes. So here is one child class and multiple parent classes.




Python Multiple Inheritance

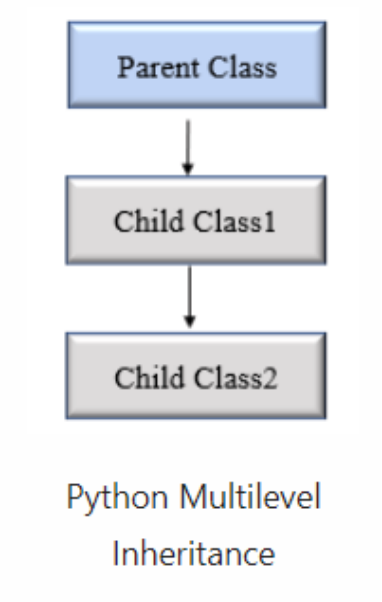
In [592]:
```
class person:
    def person_info(self,name,age):
        print("inside person class")
        print('name',name,'age',age)
class company:
    def company_info(self,company_name,location):
        print('inside company class')
        print('name',company_name,'location',location)
class employee(person,company):
     def employee_info(self,salary,skill):

            print("inside emp class")
            print ('salary',salary,'skills',skill)
emp=employee()
emp.person_info('jessa',28)
emp.company_info('google','atlanta')
emp.employee_info(12000,'ML')
```

```
inside person class
name jessa age 28
inside company class
name google location atlanta
inside emp class
salary 12000 skills ML
```

In the above example, we created two parent classes Person and Company respectively. Then we create one child called Employee which inherit from Person and Company classes.

### 7.1.5 Multilevel inheritance

In multilevel inheritance, a class inherits from a child class or derived class. Suppose three classes A, B, C. A is the superclass, B is the child class of A, C is the child class of B. In other words, we can say a chain of classes is called multilevel inheritance.

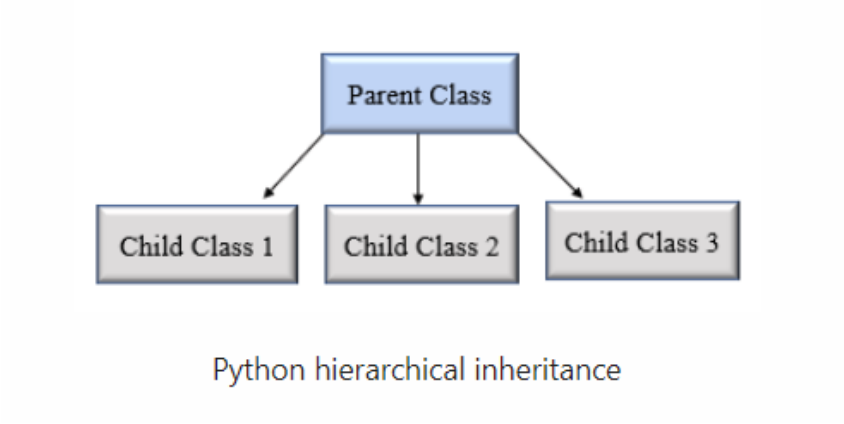



```
In [596]: class vehicle:
              def vehicle_info(self):
                  print("inside vehicle class")
          class car(vehicle):
              def car_info(self):
                  print("inside car class ")
          class sportscar(car):
              def sportscar_info(self):
                  print("inside sports car class")
          s_car=sportscar()
          s_car.car_info()
          s_car.vehicle_info()
          s_car.sportscar_info()

          inside car class
          inside vehicle class
          inside sports car class
```

In the above example, we can see there are three classes named Vehicle, Car, SportsCar. Vehicle is the superclass, Car is a child of Vehicle, SportsCar is a child of Car. So we can see the chaining of classes.

### 7.1.6 Hierarchical Inheritance

In Hierarchical inheritance, more than one child class is derived from a single parent class. In other words, we can say one parent class and multiple child classes.




Python hierarchical inheritance

**Example**

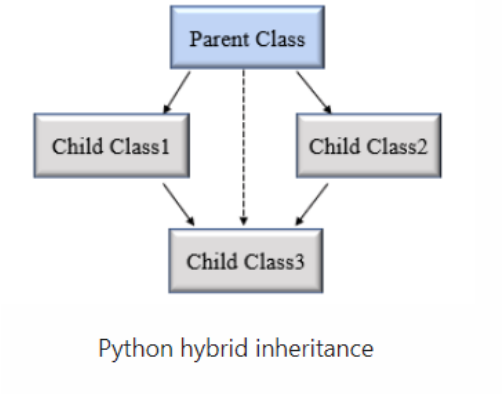
Let's create 'Vehicle' as a parent class and two child class 'Car' and 'Truck' as a parent class.

```
[40]: class vehicle:
          def vehicle_info(self):
              print('this is vehicle')
      class car(vehicle):
          def car_info(self,name):
              print("car name is ",name)
      class truck(vehicle):
          def truck_info(self,name):
              print("name of truck is",name)
      obj1=car()
      obj1.vehicle_info
      obj1.car_info
      obj2=truck()
      obj2.vehicle_info()
      obj2.truck_info('ewa')
```

```
this is vehicle
name of truck is ewa
```

### 7.1.7 Hybrid Inheritance

When inheritance is consists of multiple types or a combination of different inheritance is called hybrid inheritance.




Python hybrid inheritance

In [598]:
```
class vehicle:
    def vehicle_info(self):
        print('this is vehicle class')
class car(vehicle):
    def car_info(self):
        print("inside car class")
class truck(vehicle):
    def truck_info(self,name):
        print("inside truck class")
class sportscar(car,vehicle):
    def sportscar_info(self):
        print("inside sports car class")
s_car=sportscar()
s_car.car_info()
s_car.sportscar_info()
```

```
inside car class
inside sports car class
```

Note: In the above example, hierarchical and multiple inheritance exists. Here we created, parent class Vehicle and two child classes named Car and Truck this is hierarchical inheritance.

Another is SportsCar inherit from two parent classes named Car and Vehicle. This is multiple inheritance.

## 7.2 Python super() function

When a class inherits all properties and behavior from the parent class is called inheritance. In such a case, the inherited class is a subclass and the latter class is the parent class.

In child class, we can refer to parent class by using the `super()` function. The super function returns a temporary object of the parent class that allows us to call a parent class method inside a child class method.

**Benefits of using the super() function.**

- We are not required to remember or specify the parent class name to access its methods.
- We can use the `super()` function in both single and multiple inheritances.
- The `super()` function support code reusability as there is no need to write the entire function

In [606]:
```
class Company:
    def Company_name(self):
        return 'google'
class employee(Company):
    def info(self):
        c_name=super().Company_name()
        print("jessa works at",c_name)
emp=employee()
emp.info()
```

```
jessa works at google
```

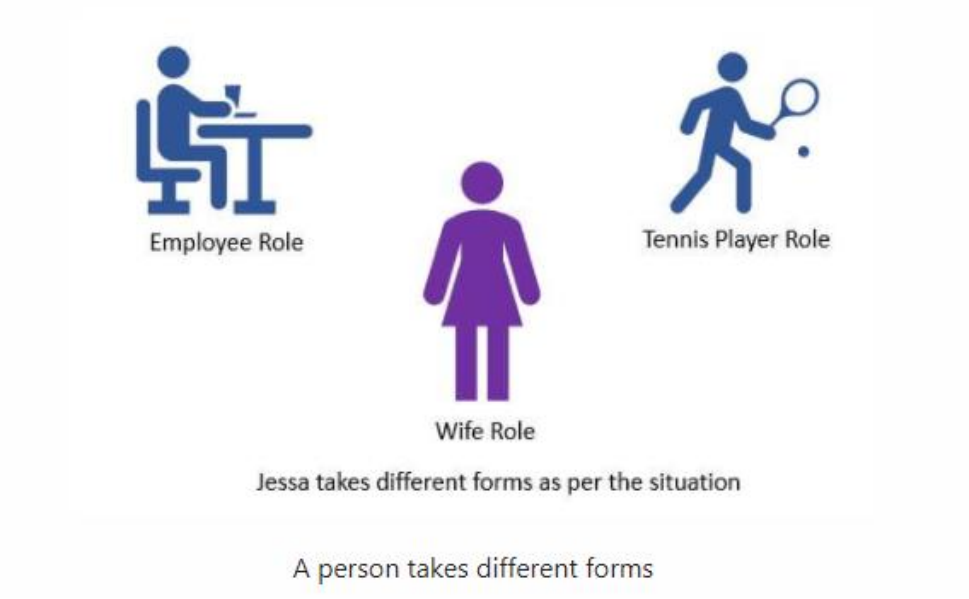
---

In the above example, we create a parent class Company and child class Employee. In Employee class, we call the parent class method by using a `super()` function.

## 7.3 Polymorphism in Python

Polymorphism in Python is the ability of an object to take many forms. In simple words, polymorphism allows us to perform the same action in many different ways.

For example, Jessa acts as an employee when she is at the office. However, when she is at home, she acts like a wife. Also, she represents herself differently in different places. Therefore, the same person takes different forms as per the situation.




In polymorphism, a method can process objects differently depending on the class type or data type. Let's see simple examples to understand it better.

## 7.4 Polymorphism in Built-in function len()

The built-in function `len()` calculates the length of an object depending upon its type. If an object is a string, it returns the count of characters, and If an object is a list, it returns the count of items in a list.

The `len()` method treats an object as per its class type.

Example:

```
In [610]: #polymorphism
          students=['emma','jess','kelly']
          school='abc school'
          print(len(students))
          print(len(school))

          3
          10
```

## 7.5 Polymorphism with Inheritance

Polymorphism is mainly used with inheritance. In inheritance, child class inherits the attributes and methods of a parent class. The existing class is called a base class or parent class, and the new class is called a subclass or child class or derived class.

Using **method overriding** polymorphism allows us to define methods in the child class that have the same name as the methods in the parent class. This **process of re-implementing the inherited method in the child class** is known as Method Overriding.

**Advantage of method overriding**

- It is effective when we want to extend the functionality by altering the inherited method. Or the method inherited from the parent class doesn't fulfill the need of a child class, so we need to re-implement the same method in the child class in a different way.
- Method overriding is useful when a parent class has multiple child classes, and one of that child class wants to redefine the method. The other child classes can use the parent class method. Due to this, we don't need to modification the parent class code

In polymorphism, **Python first checks the object's class type and executes the appropriate method** when we call the method. For example, If you create the Car object, then Python calls the `speed()` method from a Car class.

Let's see how it works with the help of an example.

### 7.5.1 Method Overriding

In this example, we have a vehicle class as a parent and a 'Car' and 'Truck' as its sub-class. But each vehicle can have a different seating capacity, speed, etc., so we can have the same instance method name in each class but with a different implementation. Using this code can be extended and easily maintained over time.

```
class Vehicle:

    def __init__(self, name, color, price):
        self.name = name
        self.color = color
        self.price = price

    def show(self):
        print('Details:', self.name, self.color, self.price)

    def max_speed(self):
        print('Vehicle max speed is 150')

    def change_gear(self):
        print('Vehicle change 6 gear')


# inherit from vehicle class
class Car(Vehicle):
    def max_speed(self):
        print('Car max speed is 240')

    def change_gear(self):
        print('Car change 7 gear')


# Car Object
car = Car('Car x1', 'Red', 20000)
car.show()
# calls methods from Car class
car.max_speed()
car.change_gear()

# Vehicle Object
vehicle = Vehicle('Truck x1', 'white', 75000)
vehicle.show()
# calls method from a Vehicle class
vehicle.max_speed()
vehicle.change_gear()
```

```
Details: Car x1 Red 20000
Car max speed is 240
Car change 7 gear

Details: Truck x1 white 75000
Vehicle max speed is 150
Vehicle change 6 gear
```

As you can see, due to polymorphism, the Python interpreter recognizes that the `max_speed()` and `change_gear()` methods are overridden for the car object. So, it uses the one defined in the child class (Car)

On the other hand, the `show()` method isn't overridden in the Car class, so it is used from the Vehicle class.

#### 7.5.1.1 Overrride Built-in Functions

In Python, we can change the default behavior of the built-in functions. For example, we can change or extend the built-in functions such as `len()`, `abs()`, or `divmod()` by redefining them in our class. Let's see the example.

Example

In this example, we will redefine the function `len()`

```
In [611]: class shopping:
              def __init__(self,basket,buyer):
                  self.basket=list(basket)
                  self.buyer=buyer
              def __len__(self):
                  print('redefine lenght')
                  count=len(self.basket)
                  return count*2
          shopp=shopping(['shoes','dress'],'jessa')
          print(len(shopp))

          redefine lenght
          4
```

## 7.6 Polymorphism in Class methods

Polymorphism with class methods is useful when we group different objects having the same method. we can add them to a list or a tuple, and we don't need to check the object type before calling their methods. Instead, Python will check object type at runtime and call the correct method. Thus, we can call the methods without being concerned about which class type each object is. We assume that these methods exist in each class.

Python allows different classes to have methods with the same name.

- Let's design a different class in the same way by adding the same methods in two or more classes.
- Next, create an object of each class
- Next, add all objects in a tuple.
- In the end, iterate the tuple using a for loop and call methods of an object without checking its class.
- Example

In the below example, `fuel_type()` and `max_speed()` are the instance methods created in both classes.

```
In [613]: class ferari:
              def  fuel_type(self):
                  print("petrol")
              def max_speed(self):
                  print("350")
          class bmw:
              def  fuel_type(self):
                  print("diesal")
              def max_speed(self):
                  print("240")
          fer=ferari()
          b=bmw()
          for car in(fer,b):
              car.fuel_type()
              car.max_speed()

          petrol
          350
          diesal
          240
```

As you can see, we have created two classes Ferrari and BMW. They have the same instance method names `fuel_type()` and `max_speed()`. However, we have not linked both the classes and have we used inheritance.

We packed two different objects into a tuple and iterate through it using a car variable. It is possible due to polymorphism because we have added the same method in both classes Python first checks the object's class type and executes the method present in its class.

### 7.6.1 Polymorphism with Function and Objects

We can create polymorphism with a function that can take any object as a parameter and execute its method without checking its class type. Using this, we can call object actions using the same function instead of repeating method calls.

```
In [618]: class ferari:
              def  fuel_type(self):
                  print("petrol")
              def max_speed(self):
                  print("350")
          class bmw:
              def  fuel_type(self):
                  print("diesal")
              def max_speed(self):
                  print("240")
          #normal function
          def car_detail(obj):
              obj.fuel_type()
              obj.max_speed()

          fer=ferari()
          b=bmw()
          car_detail(fer)

          petrol
          350
```

## 7.7 Method Overloading

The process of calling the same method with different parameters is known as method overloading. Python does not support method overloading. Python considers only the latest defined method even if you overload the method. Python will raise a TypeError if you overload the method.

Example

```
def addition(a, b):
    c = a + b
    print(c)


def addition(a, b, c):
    d = a + b + c
    print(d)


# the below line shows an error
# addition(4, 5)

# This line will call the second product method
addition(3, 7, 5)
```

To overcome the above problem, we can use different ways to achieve the method overloading. In Python, to overload the class method, we need to write the method's logic so that different code executes inside the function depending on the parameter passes.

For example, the built-in function `range()` takes three parameters and produce different result depending upon the number of parameters passed to it.

Example:

```
In [619]: #method overloading
          for i in range(5):print(i,end=',')
          print()
          for i in range(5,10):print(i,end=',')
          print()
          for i in range(5,12,2):print(i,end=',')
          print()

          0,1,2,3,4,
          5,6,7,8,9,
          5,7,9,11,
```

Let's assume we have an `area()` method to calculate the area of a square and rectangle. The method will calculate the area depending upon the number of parameters passed to it.

- If one parameter is passed, then the area of a square is calculated
- If two parameters are passed, then the area of a rectangle is calculated.

Example: User-defined polymorphic method

```
In [626]: class shape:
              def area(self,a,b=0):
                  if b>0:
                      print("area of rectangle is ",a*b)
                  else:
                      print("area of square is ",a**2)
          square=shape()
          square.area(5)

          area of square is  25
```

```
In [623]: rectangle=shape()
          rectangle.area(5,3)

          area of rectangle is  15
```

## 7.8 Operator Overloading in Python

Operator overloading means changing the default behavior of an operator depending on the operands (values) that we use. In other words, we can use the same operator for multiple purposes.

For example, the + operator will perform an arithmetic addition operation when used with numbers. Likewise, it will perform concatenation when used with strings.

The operator + is used to carry out different operations for distinct data types. This is one of the most simple occurrences of polymorphism in Python.

Example:

```
In [627]: #operator overloading
          print(100+200)
          print('jess'+'roy')

          300
          jessroy
```

### 7.8.1 Overloading + operator for custom objects

Suppose we have two objects, and we want to add these two objects with a binary + operator. However, it will throw an error if we perform addition because the compiler doesn't add two objects. See the following example for more details.

Example:

```
In [628]: class book:
              def __init__(self,pages):
                  self.pages=pages
          b1=book(400)
          b2=book(300)
          print(b1+b2)

          ---------------------------------------------------------------------------
          TypeError                                 Traceback (most recent call last)
          <ipython-input-628-4126494cee40> in <module>
                4 b1=book(400)
                5 b2=book(300)
          ----> 6 print(b1+b2)

          TypeError: unsupported operand type(s) for +: 'book' and 'book'
```

We can overload + operator to work with custom objects also. Python provides some special or magic function that is automatically invoked when associated with that particular operator.

For example, when we use the + operator, the magic method `__add__()` is automatically invoked. Internally + operator is implemented by using `__add__()` method. We have to override this method in our class if you want to add two custom objects.

In [629]:
```
class book:
    def __init__(self,pages):
        self.pages=pages
    def __add__ (self,other):
        return self.pages+other.pages
b1=book(400)
b2=book(300)
print(b1+b2)
```

```
700
```

### 7.8.2 Overloading the * Operator

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def __mul__(self, timesheet):
        print('Worked for', timesheet.days, 'days')
        # calculate salary
        return self.salary * timesheet.days


class TimeSheet:
    def __init__(self, name, days):
        self.name = name
        self.days = days


emp = Employee("Jessa", 800)
timesheet = TimeSheet("Jessa", 50)
print("salary is: ", emp * timesheet)
```

```
Wroked for 50 days
salary is:  40000
```

## 7.9 Magic Methods

In Python, there are different magic methods available to perform overloading operations. The below table shows the magic methods names to overload the mathematical operator, assignment operator, and relational operators in Python.

| Operator Name | Symbol | Magic method |
|---|---|---|
| Addition | `+` | `__add__(self, other)` |
| Subtraction | `-` | `__sub__(self, other)` |
| Multiplication | `*` | `__mul__(self, other)` |
| Division | `/` | `__div__(self, other)` |
| Floor Division | `//` | `__floordiv__(self,other)` |
| Modulus | `%` | `__mod__(self, other)` |
| Power | `**` | `__pow__(self, other)` |
| Increment | `+=` | `__iadd__(self, other)` |
| Decrement | `-=` | `__isub__(self, other)` |
| Product | `*=` | `__imul__(self, other)` |
| Division | `/+` | `__idiv__(self, other)` |
| Modulus | `%=` | `__imod__(self, other)` |
| Power | `**=` | `__ipow__(self, other)` |
| Less than | `<` | `__lt__(self, other)` |
| Greater than | `>` | `__gt__(self, other)` |
| Less than or equal to | `<=` | `__le__(self, other)` |
| Greater than or equal to | `>=` | `__ge__(self, other)` |

## 7.10 Tasks:

# 8. Lab#08 - Python Exceptions Handling

---

## Objectives:

- To learn and able to use Exception handling
- To learn and able to use try, except, else and finally
- To learn and able to use assert and raise

## Outcomes:

- Students should be able to use Exception handling
- Students should be able to use try, except, else and finally
- Students should be able to use assert and raise

---

## 8.1 Exception Handling in Python

A Python program terminates as soon as it encounters an error. In Python, an error can be a syntax error or an exception. In this Lab, you will see what an exception is and how it differs from a syntax error. After that, you will learn about raising exceptions and making assertions. Then, you'll finish with a demonstration of the try and except block.


## 8.2 Exceptions versus Syntax Errors

Syntax errors occur when the parser detects an incorrect statement. Observe the following example:

```
In [635]: print(0/0))
            File "<ipython-input-635-ded43ae9deb7>", line 1
              print(0/0))
                        ^
          SyntaxError: unmatched ')'
```

The arrow indicates where the parser ran into the **syntax error**. In this example, there was one bracket too many. Remove it and run your code again:

```
In [636]: #exception
          print(0/0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-636-67e88e78e653> in <module>
      1 #exception
----> 2 print(0/0)

ZeroDivisionError: division by zero
```

This time, you ran into an **exception error**. This type of error occurs whenever syntactically correct Python code results in an error. The last line of the message indicated what type of exception error you ran into.

Instead of showing the message exception error, Python details what type of exception error was encountered. In this case, it was a ZeroDivisionError. Python comes with various built-in exceptions as well as the possibility to create self-defined exceptions.

## 8.3 Raising an Exception

We can use raise to throw an exception if a condition occurs. The statement can be complemented with a custom exception.


If you want to throw an error when a certain condition occurs using raise, you could go about it like this:

```
In [637]: x=10
          if x>5:
              raise Exception('x should not exceed 5'.format(x))
```

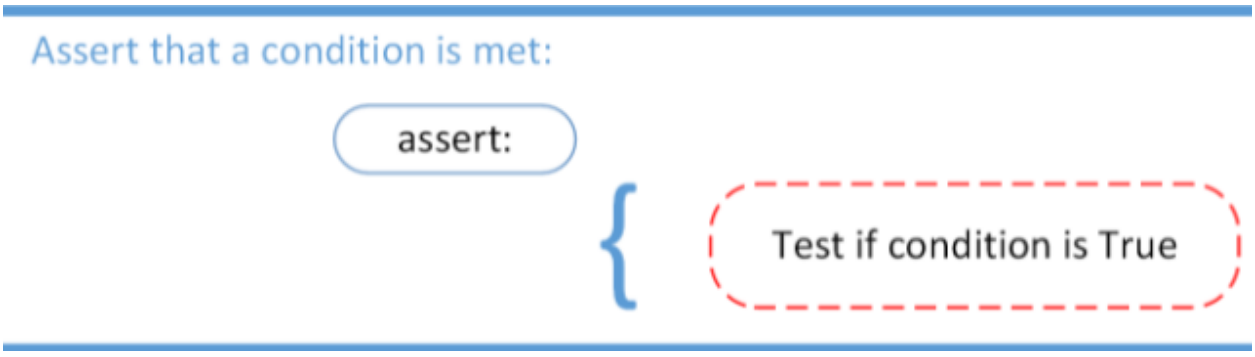
When you run this code, the output will be the following:

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-637-1a57ae5b90b2> in <module>
      1 x=10
      2 if x>5:
----> 3     raise Exception('x should not exceed 5'.format(x))

Exception: x should not exceed 5
```

The program comes to a halt and displays our exception to screen, offering clues about what went wrong.

## 8.4 The AssertionError Exception

Instead of waiting for a program to crash midway, you can also start by making an assertion in Python. We assert that a certain condition is met. If this condition turns out to be True, then that is excellent! The program can continue. If the condition turns out to be False, you can have the program throw an AssertionError exception.




Have a look at the following example, where it is asserted that the code will be executed on a Linux system:

```
In [ ]: import sys
        assert ('linux' in sys.platform),"this code run on linux only"
```

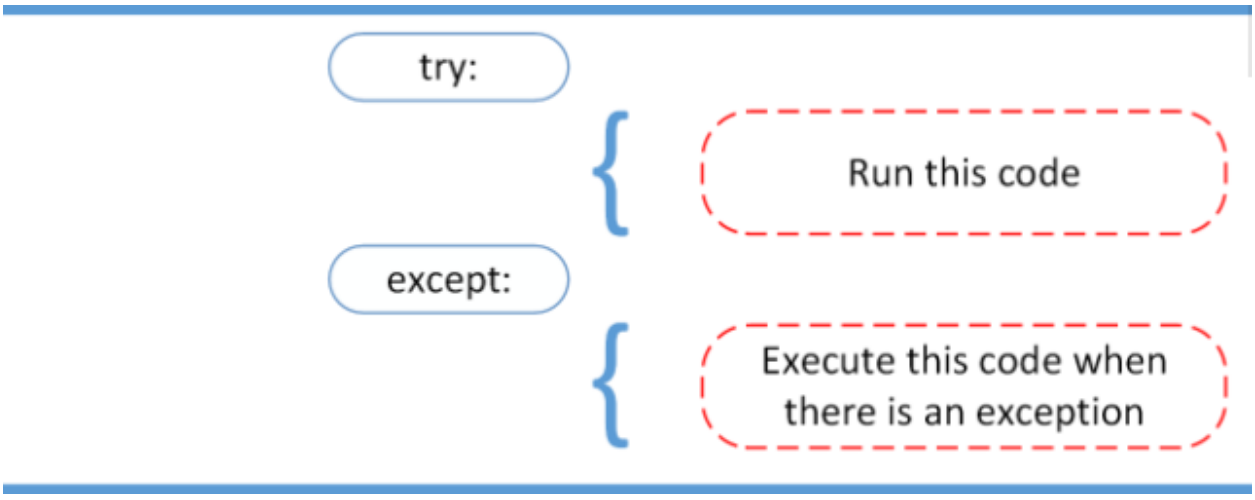
If you run this code on a Linux machine, the assertion passes. If you were to run this code on a Windows machine, the outcome of the assertion would be False and the result would be the following:

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-638-d4b29bea9319> in <module>
      1 import sys
----> 2 assert ('linux' in sys.platform),"this code run on linux only"

AssertionError: this code run on linux only
```

In this example, throwing an AssertionError exception is the last thing that the program will do. The program will come to halt and will not continue. What if that is not what you want?

## 8.5 The try and except Block: Handling Exceptions

The try and except block in Python is used to catch and handle exceptions. Python executes code following the try statement as a “normal” part of the program. The code that follows the except statement is the program’s response to any exceptions in the preceding try clause.




As you saw earlier, when syntactically correct code runs into an error, Python will throw an exception error. This exception error will crash the program if it is unhandled. The except clause determines how your program responds to exceptions.

The following function can help you understand the try and except block:

```
In [639]: def linux_interaction():
              assert('linux'in sys.platform),"function can only run on linux "
              print("doing something")
```

The `linux_interaction()` can only run on a Linux system. The assert in this function will throw an AssertionError exception if you call it on an operating system other then Linux.

You can give the function a try using the following code:

```
In [640]: try:
              linux_interaction()
          except:
              pass
```

The way you handled the error here is by handing out a pass. If you were to run this code on a Windows machine, you would get the following output:

**You got nothing**. The good thing here is that the program did not crash. But it would be nice to see if some type of exception occurred whenever you ran your code. To this end, you can change the pass into something that would generate an informative message, like so:

```
In [641]: try:
              linux_interaction()
          except:
              print("linux function was not executed")
          linux function was not executed
```

Execute this code on a Windows machine:

```
Shell

Linux function was not executed
```

When an exception occurs in a program running this function, the program will continue as well as inform you about the fact that the function call was not successful.

What you did not get to see was the type of error that was thrown as a result of the function call. In order to see exactly what went wrong, you would need to catch the error that the function threw.

The following code is an example where you capture the AssertionError and output that message to screen:

```
In [ ]: try:
            linux_interaction()
        except AssertionError as error:
            print("linux function was not executed")
```

Running this function on a Windows machine outputs the following:

```
In [642]: try:
              linux_interaction()
          except AssertionError as error:
              print("linux function was not executed")

          linux function was not executed
```

The first message is the AssertionError, informing you that the function can only be executed on a Linux machine. The second message tells you which function was not executed.

In the previous example, you called a function that you wrote yourself. When you executed the function, you caught the AssertionError exception and printed it to screen.

Here's another example where you open a file and use a built-in exception:

```
In [ ]: try:
            with open('file.log') as file:
                read_data=file.read()
        except:
            print('could not open file log')
```

If file.log does not exist, this block of code will output the following:

```
In [643]: try:
              with open('file.log') as file:
                  read_data=file.read()
          except:
              print('could not open file log')

          could not open file log
```

This is an informative message, and our program will still continue to run. In the Python docs, you can see that there are a lot of built-in exceptions that you can use here. One exception described on that page is the following:

- Exception FileNotFoundError
- Raised when a file or directory is requested but doesn't exist. Corresponds to errno ENOENT.

To catch this type of exception and print it to screen, you could use the following code:

```
In [ ]: try:
            with open('file.log') as file:
                read_data=file.read()
        except FileNotFoundError as fnf_error:
            print(fnf_error)
```

In this case, if file.log does not exist, the output will be the following:

```
In [644]: try:
              with open('file.log') as file:
                  read_data=file.read()
          except FileNotFoundError as fnf_error:
              print(fnf_error)

          [Errno 2] No such file or directory: 'file.log'
```

You can have more than one function call in your try clause and anticipate catching various exceptions. A thing to note here is that the code in the try clause will stop as soon as an exception is encountered.

Look at the following code. Here, you first call the `linux_interaction()` function and then try to open a file:

```
In [ ]: try:
            linux_interaction()
            with open('file.log') as file:
                read_data=file.read()
        except FileNotFoundError as fnf_error:
            print(fnf_error)
        except AssertionError as error:
            print(error)
            print("linux linux_interaction() function was not executed")
```

If the file does not exist, running this code on a Windows machine will output the following:

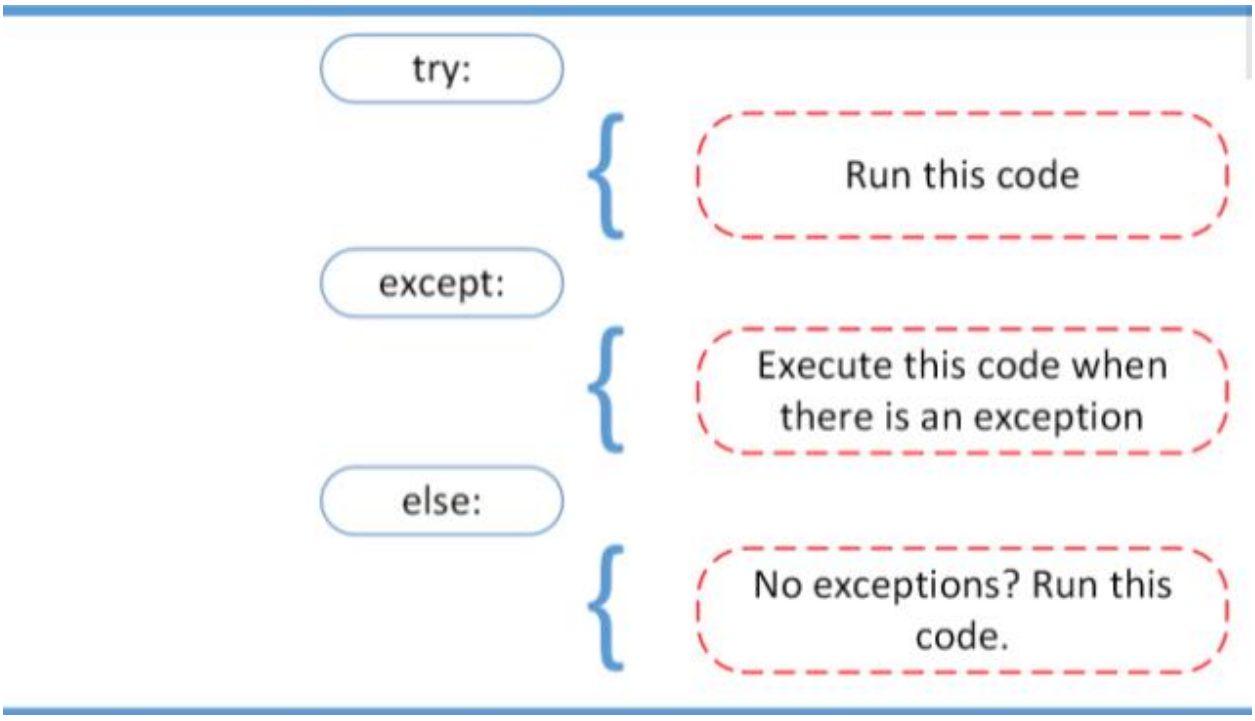
```
function can only run on linux
linux linux_interaction() function was not executed
```

Here are the key takeaways:

- A try clause is executed up until the point where the first exception is encountered.
- Inside the except clause, or the exception handler, you determine how the program responds to the exception.
- You can anticipate multiple exceptions and differentiate how the program should respond to them.
- Avoid using bare except clauses.

## 8.6 The else Clause

In Python, using the else statement, you can instruct a program to execute a certain block of code only in the absence of exceptions.




Look at the following example:

```
In [651]: try:
              linux_interaction()
          except AssertionError as error:
              print(error)
          else:
              print("executing the else clause")
```

If you were to run this code on a Linux system, the output would be the following:

```
Shell

Doing something.
Executing the else clause.
```

Because the program did not run into any exceptions, the else clause was executed.

You can also try to run code inside the else clause and catch possible exceptions there as well:

```
In [ ]: try:
            linux_interaction()
        except AssertionError as error:
            print(error)
        else:
            with open('file.log') as file:
                read_data=file.read()
        except FileNotFoundError as fnf_error:
            print(fnf_error)
```

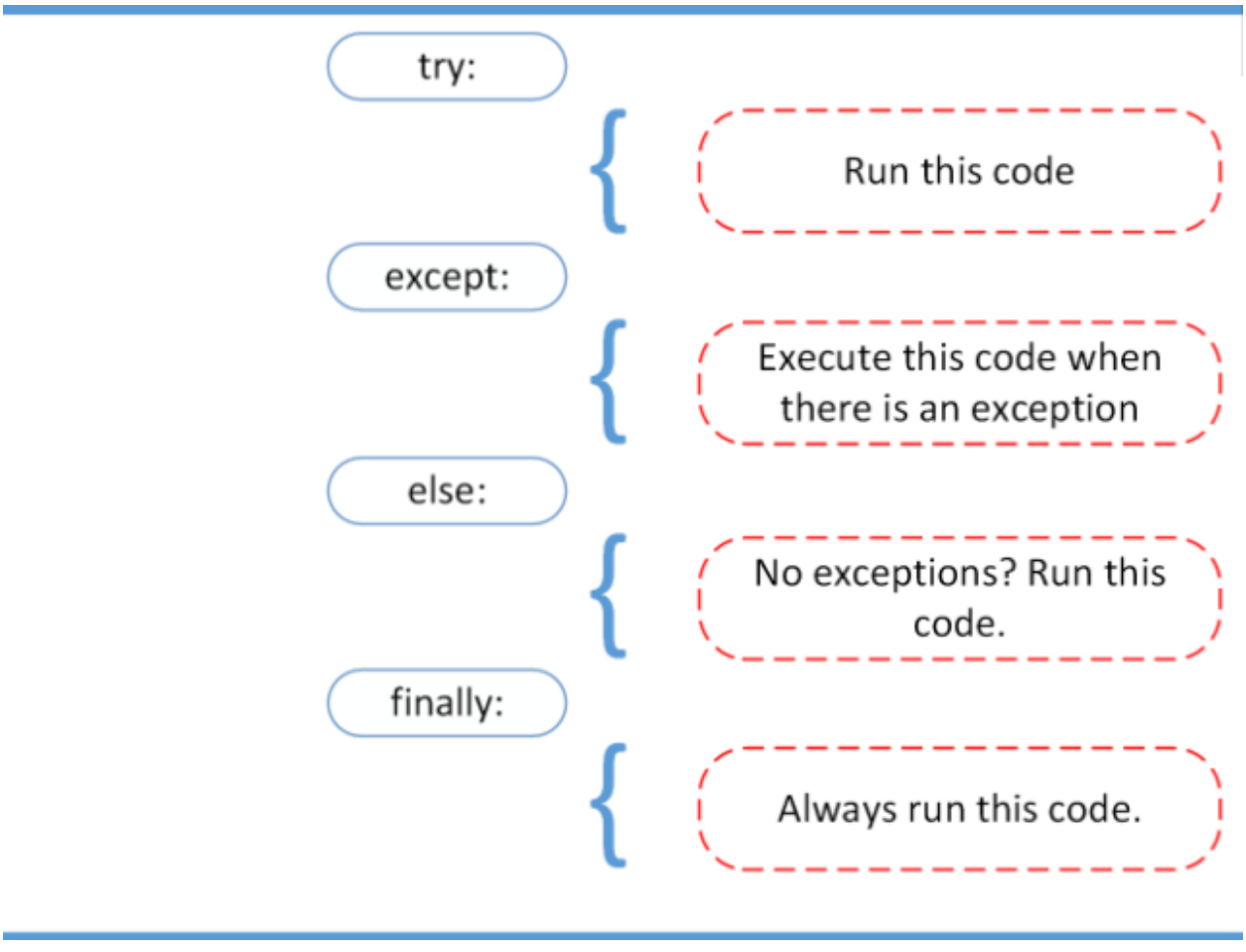
If you were to execute this code on a Linux machine, you would get the following result:

```
Shell

Doing something.
[Errno 2] No such file or directory: 'file.log'
```

From the output, you can see that the `linux_interaction()` function ran. Because no exceptions were encountered, an attempt to open file.log was made. That file did not exist, and instead of opening the file, you caught the FileNotFoundError exception.

## 8.7 Cleaning Up After Using finally

Imagine that you always had to implement some sort of action to clean up after executing your code. Python enables you to do so using the `finally` clause.




Have a look at the following example:

```
In [ ]: try:
            linux_interaction()
        except AssertionError as error:
            print(error)
        else:
            with open('file.log') as file:
                read_data=file.read()
        except FileNotFoundError as fnf_error:
            print(fnf_error)
        finally:
            print("cleaning up")
```

In the previous code, everything in the `finally` clause will be executed. It does not matter if you encounter an exception somewhere in the try or else clauses. Running the previous code on a Windows machine would output the following:

Shell

```
Function can only run on Linux systems.
Cleaning up, irrespective of any exceptions.
```

## 8.8 Built-in Exceptions

The below table shows different built-in exceptions.

Python automatically generates many exceptions and errors. Runtime exceptions, generally a result of programming errors, such as:

- Reading a file that is not present
- Trying to read data outside the available index of a list
- Dividing an integer value by zero

| Exception | Description |
|---|---|
| `AssertionError` | Raised when an `assert` statement fails. |
| `AttributeError` | Raised when attribute assignment or reference fails. |
| `EOFError` | Raised when the `input()` function hits the end-of-file condition. |
| `FloatingPointError` | Raised when a floating-point operation fails. |
| `GeneratorExit` | Raise when a generator's close() method is called. |
| `ImportError` | Raised when the imported module is not found. |
| `IndexError` | Raised when the index of a sequence is out of range. |
| `KeyError` | Raised when a key is not found in a dictionary. |
| `KeyboardInterrupt` | Raised when the user hits the interrupt key (Ctrl+C or Delete) |
| `MemoryError` | Raised when an operation runs out of memory. |
| `NameError` | Raised when a variable is not found in the local or global scope. |
| `OSError` | Raised when system operation causes system related error. |
| `ReferenceError` | Raised when a weak reference proxy is used to access a garbage collected referent. |

Python Built-in Exceptions

## 8.9 Tasks:

# 9. Lab#09 - Python Version Control

---

## Objectives:

- To learn and able to create Git repository, add new file, commit a file
- To learn and able check git log
- To learn and able to clone, pull and push
- To learn and able to merge

## Outcomes:

- Students should be able to create Git repository, add new file, commit a file
- Students should be able check git log
- Students should be able to clone, pull and push
- Students should be able to merge

---

## 9.1 Git

Git is a distributed version control system (DVCS). Let's break that down a bit and look at what it means.

### 9.1.1 Version Control

A *version control system* (VCS) is a set of tools that track the history of a set of files. This means that you can tell your VCS (Git, in our case) to save the state of your files at any point. Then, you may continue to edit the files and store that state as well. Saving the state is similar to creating a backup copy of your working directory. When using Git, we refer to this saving of state as *making a commit*.

When you make a commit in Git, you add a commit message that explains at a high level what changes you made in this commit. Git can show you the history of all of the commits and their commit messages. This provides a useful history of what work you have done and can really help pinpoint when a bug crept into the system.

In addition to showing you the log of changes you've made, Git also allows you to compare files between different commits. As I mentioned earlier, Git will also allow you to return any file (or all files) to an earlier commit with little effort.

### 9.1.2 Distributed Version Control

OK, so that's a version control system. What's the distributed part? It's probably easiest to answer that question by starting with a little history. Early version control systems worked by storing all of those commits locally on your hard drive. This collection of commits is called a repository. This solved the "I need to get back to where I was" problem but didn't scale well for a team working on the same codebase.

As larger groups started working (and networking became more common), VCSs changed to store the repository on a central server that was shared by many developers. While this solved many problems, it also created new ones, like file locking.

Following the lead of a few other products, Git broke with that model. Git does not have a central server that has the definitive version of the repository. All users have a full copy of the repository. This means that getting all of the developers back on the same page can sometimes be tricky, but it also means that developers can work offline most of the time, only connecting to other repositories when they need to share their work.

That last paragraph can seem a little confusing at first, because there are a lot of developers who use GitHub as a central repository from which everyone must pull. This is true, but Git doesn't impose this. It's just

convenient in some circumstances to have a central place to share the code. The full repository is still stored on all local repos even when you use GitHub.

## 9.2 Basic Usage

Now that we've talked about what Git is in general, let's run through an example and see it in action. We'll start by working with Git just on our local machine. Once we get the hang of that, we'll add GitHub and explain how you can interact with it.

### 9.2.1 Creating a New Repo

To work with Git, you first need to tell it who you are. You can set your username with the git config command:

Shell

```
$ git config --global user.name "your name goes here"
```

Once that is set up, you will need a repo to work in. Creating a repo is simple. Use the git init command in a directory:

Shell

```
$ mkdir example
$ cd example
$ git init
Initialized empty Git repository in /home/jima/tmp/example/.git/
```

Once you have a repo, you can ask Git about it. The Git command you'll use most frequently is git status. Try that now:

Shell

```
$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

This shows you a couple of bits of information: which branch you're on, master (we'll talk about branches later), and that you have nothing to commit. This last part means that there are no files in this directory that Git doesn't know about. That's good, as we just created the directory.

### 9.2.2 Adding a New File

Now create a file that Git doesn't know about. With your favorite editor, create the file hello.py, which has just a print statement in it.

```
# hello.py
print('hello Git!')
```

If you run git status again, you'll see a different result:

Shell

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

   hello.py

nothing added to commit but untracked files present (use "git add" to track)
```

Now Git sees the new file and tells you that it's untracked. That's just Git's way of saying that the file is not part of the repo and is not under version control. We can fix that by adding the file to Git. Use the git add command to make that happen:

Shell

```
$ git add hello.py
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

   new file:   hello.py
```

Now Git knows about hello.py and lists it under changes to be committed. Adding the file to Git moves it into the staging area (discussed below) and means we can commit it to the repo.

### 9.2.3 Committing Changes

When you commit changes, you are telling Git to make a snapshot of this state in the repo. Do that now by using the git commit command. The -m option tells Git to use the commit message that follows. If you don't use -m, Git will bring up an editor for you to create the commit message. In general, you want your commit messages to reflect what has changed in the commit:

Shell

```
$ git commit -m "creating hello.py"
[master (root-commit) 25b09b9] creating hello.py
 1 file changed, 3 insertions(+)
 create mode 100755 hello.py

$ git status
On branch master
nothing to commit, working directory clean
```

You can see that the commit command returned a bunch of information, most of which isn't that useful, but it does tell you that only 1 file changed (which makes sense as we added one file). It also tells you the *SHA* of the commit (25b09b9). We'll have an aside about SHA a bit later.

Running the git status command again shows that we have a *clean* working directory, meaning that all changes are committed to Git.

At this point, we need to stop our tutorial and have a quick chat about the staging area.

## 9.3 Aside: The Staging Area

Unlike many version control systems, Git has a *staging area* (often referred to as *the index*). The staging area is how Git keeps track of the changes you want to be in your next commit. When we ran git add above, we told Git that we wanted to move the new file hello.py to the staging area. This change was reflected in git status. The file went from the *untracked* section to the *to be committed* section of the output.

Note that the staging area reflects the exact contents of the file when you ran git add. If you modify it again, the file will appear both in the *staged* and *unstaged* portions of the status output.

At any point of working with a file in Git (assuming it's already been committed once), there can be three versions of the file you can work with:

- the version on your hard drive that you are editing
- a different version that Git has stored in your staging area
- the latest version checked in to the repo

All three of these can be different versions of the file. Moving changes to the staging area and then committing them brings all of these versions back into sync.

### 9.3.1 .gitignore

The status command is very handy, and you'll find yourself using it often. But sometimes you'll find that there are a bunch of files that show up in the untracked section and that you want Git to just not see. That's where the .gitignore file comes in.

Let's walk through an example. Create a new Python file in the same directory called myname.py.

Python

```
# myname.py
def get_name():
    return "Jim"
```

Then modify your hello.py to include myname and call its function:

Python

```
# hello.py
import myname

name = myname.get_name()
print("hello {}".format(name))
```

When you import a local module, Python will compile it to bytecode for you and leave that file on your filesystem. In Python 2, it will leave a file called myname.pyc, but we'll assume you're running Python 3. In that case it will create a __pycache__ directory and store a pyc file there. That is what's shown below:

Shell

```
$ ls
hello.py  myname.py
$ ./hello.py
hello Jim!
$ ls
hello.py  myname.py  __pycache__
```

Now if you run git status, you'll see that directory in the untracked section. Also note that your new myname.py file is untracked, while the changes you made to hello.py are in a new section called "Changes not staged for commit". This just means that those changes have not yet been added to the staging area. Let's try it out:

Shell

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

   modified:   hello.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)

   __pycache__/
   myname.py

no changes added to commit (use "git add" and/or "git commit -a")
```

Before we move on to the gitignore file, let's clean up the mess we've made a little bit. First we'll add the myname.py and hello.py files, just like we did earlier:

Shell

```
$ git add myname.py hello.py
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

   modified:   hello.py
   new file:   myname.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)

   __pycache__/
```

Let’s commit those changes and finish our clean up:

Shell

```
$ git commit -m "added myname module"
[master 946b99b] added myname module
 2 files changed, 8 insertions(+), 1 deletion(-)
 create mode 100644 myname.py
```

Now when we run status, all we see is that __pycache__ directory:

Shell

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

   __pycache__/

nothing added to commit but untracked files present (use "git add" to track)
```

To get all __pycache__ directories (and their contents) to be ignored, we’re going to add a .gitignore file to our repo. This is as simple as it sounds. Edit the file (remember the dot in front of the name!) in your favorite editor.

Python

```
# .gitignore
__pycache__
```

Now when we run git status, we no longer see the __pycache__ directory. We do, however, see the new .gitignore! Take a look:

```
Shell

$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    .gitignore

nothing added to commit but untracked files present (use "git add" to track)
```

That file is just a regular text file and can be added to you repo like any other file. Do that now:

```
Shell

$ git add .gitignore
$ git commit -m "created .gitignore"
[master 1cada8f] created .gitignore
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
```

Another common entry in the `.gitignore` file is the directory you store your virtual environments in. You can learn more about virtualenvs here, but the virtualenv directory is usually called env or venv. You can add these to your .gitignore file. If the files or directories are present, they will be ignored. If they are not, then nothing will happen.

It's also possible to have a *`global`* `.gitignore` file stored in your home directory. This is very handy if your editor likes to leave temporary or backup files in the local directory.

Here's an example of a simple Python .gitignore file:

```
Shell

# .gitignore
__pycache__
venv
env
.pytest_cache
.coverage
```

## 9.4 What NOT to Add to a Git Repo

When you first start working with any version control tool, especially Git, you might be tempted to put **everything** into the repo. This is generally a mistake. There are limitations to Git as well as security concerns that force you to limit which types of information you add to the repo.

Let's start with the basic rule of thumb about **all** version control systems.

Only put *source* files into version control, never *generated* files.

In this context, a *source* file is any file you create, usually by typing in an editor. A *generated* file is something that the computer creates, usually by processing a *source* file. For example, hello.py is a *source* file, while hello.pyc would be a *generated* file.

There are two reasons for not including generated files in the repo. The first is that doing so is a waste of time and space. The generated files can be recreated at any time and may need to be created in a different form. If someone is using Jython or IronPython while you're using the Cython interpreter, the .pyc files might be quite different. Committing one particular flavor of files can cause conflict

The second reason for not storing generated files is that these files are frequently larger than the original source files. Putting them in the repo means that everyone now needs to download and store those generated files, even if they're not using them.

This second point leads to another general rule about Git repos: commit binary files with caution and strongly avoid committing large files. This rule has a lot to do with how Git works.

Git does not store a full copy of each version of each file you commit. Rather, it uses a complicated algorithm based on the differences between subsequent versions of a file to greatly reduce the amount of storage it needs. Binary files (like JPGs or MP3 files) don't really have good diff tools, so Git will frequently just need to store the entire file each time it is committed.

When you are working with Git, and especially when you are working with GitHub, **never** put confidential information into a repo, especially one you might share publicly. This is so important that I'm going to say it again:

## 9.5 Aside: What is a SHA

When Git stores things (files, directories, commits, etc) in your repo, it stores them in a complicated way involving a *hash function*. We don't need to go into the details here, but a hash function takes a thing and produces a unique ID for that thing that is much shorter (20 bytes, in our case). This ID is called a "SHA" in Git. It is not guaranteed to be unique, but for most practical applications it is.

Git uses its hash algorithm to index **everything** in your repo. Each file has a SHA that reflects the contents of that file. Each directory, in turn, is hashed. If a file in that directory changes, then the SHA of the directory changes too.

Each commit contains the SHA of the top-level directory in your repo along with some other info. That's how a single 20 byte number describes the entire state of your repo.

You might notice that sometimes Git uses the full 20 character value to show you a SHA:

```
Shell

commit 25b09b9ccfe9110aed2d09444f1b50fa2b4c979c
```

Sometimes it shows you a shorter version:

```
Shell

[master (root-commit) 25b09b9] creating hello.py
```

Usually, it will show you the full string of characters, but you don't always have to use it. The rule for Git is that you only have to give enough characters to ensure that the SHA is unique in your repo. Generally, seven characters is more than enough.

Each time you commit changes to the repo, Git creates a new SHA that describes that state. We'll look at how SHAs are useful in the next sections.

## 9.6 Git Log

Another very frequently used Git command is git log. Git log shows you the history of the commits that you have made up to this point:

```
Shell
$ git log
commit 1cada8f59b43254f621d1984a9ffa0f4b1107a3b
Author: Jim Anderson <jima@example.com>
Date:   Sat Mar 3 13:23:07 2018 -0700

    created .gitignore

commit 946b99bfe1641102d39f95616ceaab5c3dc960f9
Author: Jim Anderson <jima@example.com>
Date:   Sat Mar 3 13:22:27 2018 -0700

    added myname module

commit 25b09b9ccfe9110aed2d09444f1b50fa2b4c979c
Author: Jim Anderson <jima@example.com>
Date:   Sat Mar 3 13:10:12 2018 -0700

    creating hello.py
```

As you can see in the listing above, all of the commit messages are shown for our repo in order. The start of each commit is marked with the word “commit” followed by the SHA of that commit. git log gives you the history of each of the SHAs.

## 9.7 Branching Basics

Let’s talk a little more about branches. Branches provide a way for you to keep separate streams of development apart. While this can be useful when you’re working alone, it’s almost essential when you’re working on team.

Imagine that I’m working in a small team and have a feature to add to the project. While I’m working on it, I don’t want to add my changes to master as it still doesn’t work correctly and might mess up my team members.

I could just wait to commit the changes until I’m completely finished, but that’s not very safe and not always practical. So, instead of working on master, I’ll create a new branch:

```
Shell
$ git checkout -b my_new_feature
Switched to a new branch 'my_new_feature'
$ git status
On branch my_new_feature
nothing to commit, working directory clean
```

We used the -b option on the checkout command to tell Git we wanted it to create a new branch. As you can see above, running git status in our branch shows us that the branch name has, indeed, changed. Let’s look at the log:

Shell

```
$ git log
commit 1cada8f59b43254f621d1984a9ffa0f4b1107a3b
Author: Jim Anderson <jima@example.com>
Date:   Thu Mar 8 20:57:42 2018 -0700

    created .gitignore

commit 946b99bfe1641102d39f95616ceaab5c3dc960f9
Author: Jim Anderson <jima@example.com>
Date:   Thu Mar 8 20:56:50 2018 -0700

    added myname module

commit 25b09b9ccfe9110aed2d09444f1b50fa2b4c979c
Author: Jim Anderson <jima@example.com>
Date:   Thu Mar 8 20:53:59 2018 -0700

    creating hello.py
```

As I hope you expected, the log looks exactly the same. When you create a new branch, the new branch will start at the location you were at. In this case, we were at the top of master, 1cada8f59b43254f621d1984a9ffa0f4b1107a3b, so that's where the new branch starts.

Now, let's work on that feature. Make a change to the hello.py file and commit it. I'll show you the commands for review, but I'll stop showing you the output of the commands for things you've already seen:

Shell

```
$ git add hello.py
$ git commit -m "added code for feature x"
```

Now if you do git log, you'll see that our new commit is present. In my case, it has a SHA 4a4f4492ded256aa7b29bf5176a17f9eda66efbb, but your repo is very likely to have a different SHA:

Shell

```
$ git log
commit 4a4f4492ded256aa7b29bf5176a17f9eda66efbb
Author: Jim Anderson <jima@example.com>
Date:   Thu Mar 8 21:03:09 2018 -0700

    added code for feature x

commit 1cada8f59b43254f621d1984a9ffa0f4b1107a3b
... the rest of the output truncated ...
```

Now switch back to the master branch and look at the log:

Shell

```
git checkout master
git log
```

Is the new commit “added code for feature x” there?

Git has a built-in way to compare the state of two branches so you don’t have to work so hard. It’s the show-branch command. Here’s what it looks like:

Shell

```
$ git show-branch my_new_feature master
* [my_new_feature] added code for feature x
 ! [master] created .gitignore
--
*  [my_new_feature] added code for feature x
*+ [master] created .gitignore
```

The chart it generates is a little confusing at first, so let’s walk though it in detail. First off, you call the command by giving it the name of two branches. In our case that was my_new_feature and master.

The first two lines of the output are the key to decoding the rest of the text. The first non-space character on each line is either * or ! followed by the name of the branch, and then the commit message for the most recent commit on that branch. The * character is used to indicate that the branch is currently checked-out while the ! is used for all other branches. The character is in the column matching commits in the table below.

The third line is a separator.

Starting on the fourth line, there are commits that are in one branch but not the other. In our current case, this is pretty easy. There’s one commit in my_new_feature that’s not in master. You can see that on the fourth line. Notice how that line starts with a * in the first column. This is to indicate which branch this commit is in.

Finally, the last line of the output shows the first common commit for the two branches.

This example is pretty easy. To make a better example, I’ve made it more interesting by adding a few more commits to my_new_feature and a few to master. That makes the output look like:

Shell

```
$ git show-branch my_new_feature master
* [my_new_feature] commit 4
 ! [master] commit 3
--
*  [my_new_feature] commit 4
*  [my_new_feature^] commit 1
*  [my_new_feature~2] added code for feature x
 + [master] commit 3
 + [master^] commit 2
*+ [my_new_feature~3] created .gitignore
```

Now you can see that there are different commits in each branch. Note that the [my_new_feature~2] text is one of the commit selection methods I mentioned earlier. If you’d rather see the SHAs, you can have it show them by adding the –sha1-name option to the command:

Shell

```
$ git show-branch --sha1-name my_new_feature master
* [my_new_feature] commit 4
 ! [master] commit 3
--
*  [6b6a607] commit 4
*  [12795d2] commit 1
*  [4a4f449] added code for feature x
 + [de7195a] commit 3
 + [580e206] commit 2
*+ [1cada8f] created .gitignore
```

Now you've got a branch with a bunch of different commits on it. What do you do when you finally finish that feature and are ready to get it to the rest of your team?

There are three main ways to get commits from one branch to another: merging, rebasing, and cherry-picking. We'll cover each of these in turn in the next sections.

### 9.7.1 Merging

Merging is the simplest of the three to understand and use. When you do a merge, Git will create a new commit that combines the top SHAs of two branches if it needs to. If all of the commits in the other branch are ahead (based on) the top of the current branch, it will just do a fast-foward merge and place those new commits on this branch.

Let's back up to the point where our show-branch output looked like this:

Shell

```
$ git show-branch --sha1-name my_new_feature master
* [my_new_feature] added code for feature x
 ! [master] created .gitignore
--
*  [4a4f449] added code for feature x
*+ [1cada8f] created .gitignore
```

Now, we want to get that commit 4a4f449 to be on master. Check out master and run the git merge command there:

Shell

```
$ git checkout master
Switched to branch 'master'

$ git merge my_new_feature
Updating 1cada8f..4a4f449
Fast-forward
 hello.py | 1 +
 1 file changed, 1 insertion(+)
```

Since we were on branch master, we did a merge of the my_new_feature branch to us. You can see that this is a fast forward merge and which files were changed. Let's look at the log now:

```
Shell

commit 4a4f4492ded256aa7b29bf5176a17f9eda66efbb
Author: Jim Anderson <jima@example.com>
Date:   Thu Mar 8 21:03:09 2018 -0700

    added code for feature x

commit 1cada8f59b43254f621d1984a9ffa0f4b1107a3b
Author: Jim Anderson <jima@example.com>
Date:   Thu Mar 8 20:57:42 2018 -0700

    created .gitignore
[rest of log truncated]
```

If we had made changes to master before we merged, Git would have created a new commit that was the combination of the changes from the two branches.

One of the things Git is fairly good at is understanding the common ancestors of different branches and automatically merging changes together. If the same section of code has been modified in both branches, Git can't figure out what to do. When this happens, it stops the merge part way through and gives you instructions for how to fix the issue. This is called a merge conflict.

## 9.8 Working with Remote Repos

All of the commands we've discussed up to this point work with only your local repo. They don't do any communication to a server or over the network. It turns out that there are only four major Git commands which actually talk to remote repos:

- clone
- fetch
- pull
- push

That's it. Everything else is done on your local machine. (OK, to be completely accurate, there *are* other commands that talk to remotes, but they don't fall into the basic category.)

Let's look at each of these commands in turn.

### 9.8.1 Clone

Git clone is the command you use when you have the address of a known repository, and you want to make a local copy. For this example, let's use a small repo I have on my GitHub account, github-playground.

If you copy that, you can then clone the repo with:

```
Shell

git clone git@github.com:jima80525/github-playground.git
```

Now you have a complete repository of that project on your local machine. This includes all of the commits and all of the branches ever made on it. (**Note:** This repo was used by some friends while they were learning Git. I copied or *forked* it from someone else.)

If you want to play with the other remote commands, you should create a new repo on GitHub and follow the same steps. You are welcome to fork the github-playground repo to your account and use that. Forking on GitHub is done by clicking the “fork” button in the UI.

### 9.8.2 Fetch

To explain the fetch command clearly, we need to take a step back and talk about how Git manages the relationship between your local repo and a remote repo. This next part is background, and while it’s not something you’ll use on a day-to-day basis, it will make the difference between fetch and pull make more sense.

When you clone a new repo, Git doesn’t just copy down a single version of the files in that project. It copies the entire repository and uses that to create a new repository on your local machine.

Git does not make local branches for you except for master. However, it does keep track of the branches that were on the server. To do that, Git creates a set of branches that all start with remotes/origin/<branch_name>.

Only rarely (almost never), will you check out these remotes/origin branches, but it’s handy to know that they are there. Remember that every branch that existed on the remote when you cloned the repo will have a branch in remotes/origin.

When you create a new branch and the name matches an existing branch on the server, Git will mark you local branch as a *tracking branch* that is associated with a remote branch. We’ll see how that is useful when we get to pull.

Now that you know about the remotes/origin branches, understanding git fetch will be pretty easy. All fetch does is update all of the remotes/origin branches. It will modify only the branches stored in remotes/origin and not any of your local branches.

### 9.8.3 Pull

Git pull is simply the combination of two other commands. First, it does a git fetch to update the remotes/origin branches. Then, if the branch you are on is tracking a remote branch, then it does a git merge of the corresponding remote/origin branch to your branch.

For example, say you were on the my_new_feature branch, and your coworker had just added some code to it on the server. If you do a git pull, Git will update ALL of the remotes/origin branches and then do a git merge remotes/origin/my_new_feature, which will get the new commit onto the branch you’re on!

There are, of course, some limitations here. Git won’t let you even try to do a git pull if you have modified files on your local system. That can create too much of a mess.

If you have commits on your local branch, and the remote also has new commits (ie “the branches have diverged”), then the git merge portion of the pull will create a merge commit, just as we discussed above.

Those of you who have been reading closely will see that you can also have Git do a rebase instead of a merge by doing git pull -r.

### 9.8.4 Push

As you have probably guessed, git push is just the opposite of git pull. Well, almost the opposite. Push sends the info about the branch you are pushing and asks the remote if it would like to update its version of that branch to match yours.

Generally, this amounts to you pushing your new changes up to the server. There are a lot of details and complexity here involving exactly what a fast-forward commit is.

## 9.9 Putting It All Together: Simple Git Workflow

t this point, we've reviewed several basic Git commands and how you might use them. I'll wrap up with a quick description of a possible workflow in Git. This workflow assumes you are working on your local repo and have a remote repo to which you will push changes. It can be GitHub, but it will work the same with other remote repos. It assumes you've already cloned the repo.

1. git status – Make sure your current area is clean.
2. git pull – Get the latest version from the remote. This saves merging issues later.
3. Edit your files and make your changes.
4. git status – Find all files that are changed. Make sure to watch untracked files too!
5. git add [files] – Add the changed files to the staging area.
6. git commit -m "message" – Make your new commit.
7. git push origin [branch-name] – Push your changes up to the remote.

This is one of the more basic flows through the system.

## 9.10 Tasks:

# 10. Lab#10 - NumPy

---

## Objectives:

- To learn and able to use basic functions of NumPy Library
- To learn and able to use create 1d, 2d and 3d arrays
- To learn and able to use stacking and broadcasting

## Outcomes:

- Students should be able to use basic functions of NumPy Library
- Students should be able to use create 1d, 2d and 3d arrays
- Students should be able to use stacking and broadcasting

---

## 10.1 What are NumPy Arrays?

NumPy is a Python package that stands for 'Numerical Python'. It is the core library for scientific computing, which contains a powerful n-dimensional array object.

## 10.2 Where is NumPy used?

Python NumPy arrays provide tools for integrating C, C++, etc. It is also useful in linear algebra, random number capability etc. NumPy array can also be used as an efficient multi-dimensional container for generic data. Now, let me tell you what exactly is a Python NumPy array.

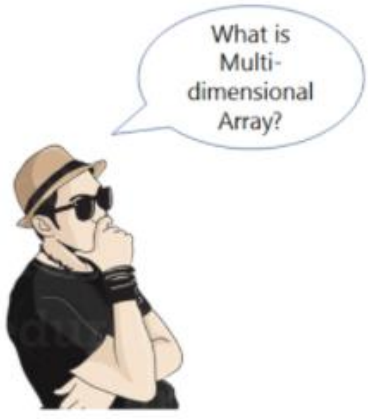
## 10.3 Python NumPy Array

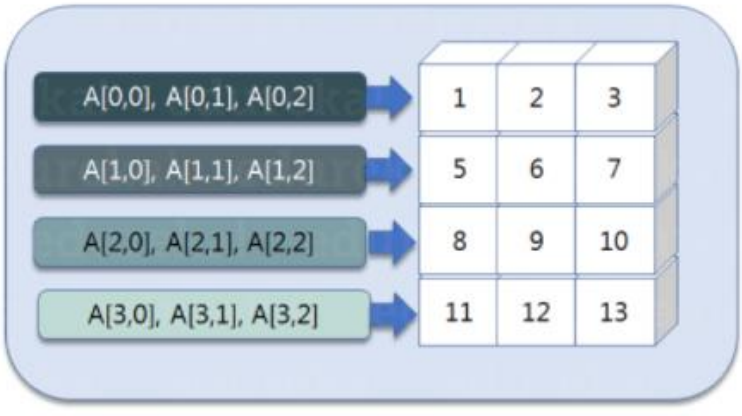
NumPy array is a powerful N-dimensional array object which is in the form of rows and columns. We can initialize NumPy arrays from nested Python lists and access it elements. In order to perform these NumPy operations, the next question which will come in your mind is:

### 10.3.1 How do I install NumPy?

To install Python NumPy, go to your command prompt and type "pip install numpy". Once the installation is completed, go to your IDE (For example: PyCharm) and simply import it by typing: "import numpy as np"

Moving ahead in python numpy tutorial, let us understand what exactly is a multi-dimensional numPy array.



Here, I have different elements that are stored in their respective memory locations. It is said to be two dimensional because it has rows as well as columns. In the above image, we have 3 columns and 4 rows available.

## 10.4 Single-dimensional Array and Multi-dimensional Array:

```
[41]: import numpy as np
      a=np.array([1,2,3])
      print(a)

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

```
[1 2 3]
[[1 2 3]
 [4 5 6]]
```

Many of you must be wondering that why do we use python NumPy if we already have Python list? So, let us understand with some examples in this python NumPy Lab.

## 10.5 Python NumPy Array v/s List

### 10.5.1 Why NumPy is used in Python?

We use python NumPy array instead of a list because of the below three reasons:

1. Less Memory
2. Fast
3. Convenient

The very first reason to choose python NumPy array is that it occupies less memory as compared to list. Then, it is pretty fast in terms of execution and at the same time, it is very convenient to work with NumPy. So these are the major advantages that Python NumPy array has over list.

```
In [658]: import numpy as np
          import time
          import sys
          s=range(1000)
          print(sys.getsizeof(5)*len(s))
          d=np.arange(1000)
          print(d.size*d.itemsize)
```

```
28000
4000
```

The above output shows that the memory allocated by list (denoted by S) is 14000 whereas the memory allocated by the NumPy array is just 4000. From this, you can conclude that there is a major difference between the two and this makes Python NumPy array as the preferred choice over list.

Next, let's talk how python NumPy array is faster and more convenient when compared to list.

```
In [659]: import time
          import sys
          size=1000000
          l1=range(size)
          l2=range(size)
          a1=np.arange(size)
          a2=np.arange(size)
          start=time.time()
          result=[(x,y) for x,y in zip(l1,l2)]
          print((time.time()-start)*1000)
          start=time.time()
          result=a1+a2
          print((time.time()-start)*1000)

          1785.7141494750977
          78.88603210449219
```

In the above code, we have defined two lists and two numpy arrays. Then, we have compared the time taken in order to find the sum of lists and sum of numpy arrays both. If you see the output of the above program, there is a significant change in the two values. List took 380ms whereas the numpy array took almost 49ms. Hence, numpy array is faster than list. Now, if you noticed we had run a 'for' loop for a list which returns the concatenation of both the lists whereas for numpy arrays, we have just added the two array by simply printing A1+A2. That's why working with numpy is much easier and convenient when compared to the lists.

Therefore, the above examples proves the point as to why you should go for python numpy array rather than a list!

## 10.6 Python NumPy Operations

### 10.6.1 ndim:

You can find the dimension of the array, whether it is a two-dimensional array or a single dimensional array. So, let us see this practically how we can find the dimensions. In the below code, with the help of 'ndim' function, I can find whether the array is of single dimension or multi dimension.

```
In [662]: import numpy as np
          a=np.array([(1,2,3),(4,5,6)])
          print(a.ndim)

          2
```

Since the output is 2, it is a two-dimensional array (multi dimension).

### 10.6.2 itemsize:

You can calculate the byte size of each element. In the below code, I have defined a single dimensional array and with the help of 'itemsize' function, we can find the size of each element.

```
In [663]: a=np.array([(1,2,3)])
          print(a.itemsize)

          4
```

So every element occupies 4 byte in the above numpy array.

### 10.6.3 dtype:

You can find the data type of the elements that are stored in an array. So, if you want to know the data type of a particular element, you can use '`dtype`' function which will print the datatype along with the size. In the below code, I have defined an array where I have used the same function.

```
In [664]: a=np.array([(1,2,3)])
          print(a.dtype)

          int32
```

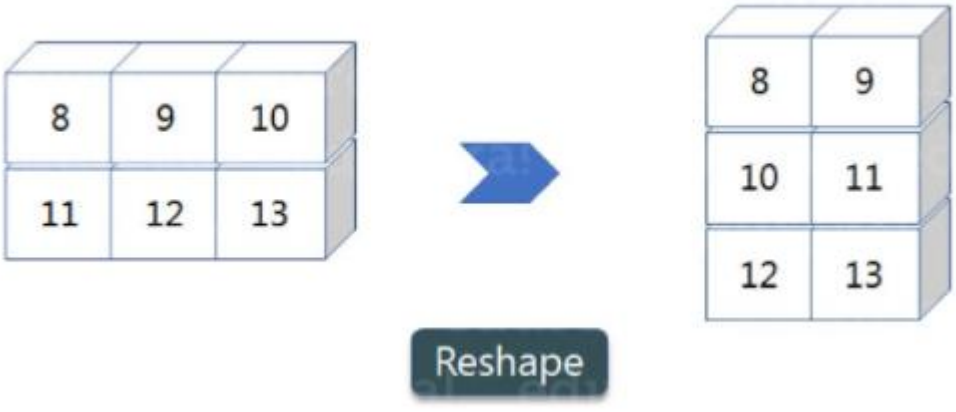
As you can see, the data type of the array is integer 32 bits. Similarly, you can find the size and shape of the array using '`size`' and '`shape`' function respectively.

```
In [665]: a=np.array([(1,2,3,4,5,6)])
          print(a.size)
          print(a.shape)

          6
          (1, 6)
```

Next, let us move forward and see what are the other operations that you can perform with python numpy module. We can also perform reshape as well as slicing operation using python numpy operation. But, what exactly is reshape and slicing? So let me explain this one by one in this python numpy lab.

### 10.6.4 reshape:

Reshape is when you change the number of rows and columns which gives a new view to an object. Now, let us take an example to reshape the below array:




As you can see in the above image, we have 3 columns and 2 rows which has converted into 2 columns and 3 rows. Let me show you practically how it's done.

```
In [666]: import numpy as np
          a=np.array([(1,2,3),(4,5,6)])
          print(a)
          a=a.reshape(3,2)
          print(a)

          [[1 2 3]
           [4 5 6]]
          [[1 2]
           [3 4]
           [5 6]]
```

### 10.6.5 slicing:

As you can see the 'reshape' function has showed its magic. Now, let's take another operation i.e Slicing. Slicing is basically extracting particular set of elements from an array. This slicing operation is pretty much similar to the one which is there in the list as well. Consider the following example:


Before getting into the above example, let's see a simple one. We have an array and we need a particular element (say 3) out of a given array. Let's consider the below example:

```
In [667]: a=np.array([(1,2,3),(4,5,6)])
          print(a[0,2])

          3
```

Here, the `array(1,2,3,4)` is your index 0 and (3,4,5,6) is index 1 of the python numpy array. Therefore, we have printed the second element from the zeroth index.

Taking one step forward, let's say we need the 2nd element from the zeroth and first index of the array. Let's see how you can perform this operation:

```
In [668]: a=np.array([(1,2,3),(4,5,6)])
          print(a[0:2])

          [[1 2 3]
           [4 5 6]]
```

Here colon represents all the rows, including zero. Now to get the 2nd element, we'll call index 2 from both of the rows which gives us the value 3 and 5 respectively.

Next, just to remove the confusion, let's say we have one more row and we don't want to get its 2nd element printed just as the image above. What we can do in such case?

Consider the below code:

```
In [670]: a=np.array([(1,2),(4,5),(12,13)])
          print(a[0:2,1])

          [2 5]
```

As you can see in the above code, only 9 and 11 gets printed. Now when I have written 0:2, this does not include the second index of the third row of an array. Therefore, only 9 and 11 gets printed else you will get all the elements i.e [9 11 13].

### 10.6.6 linspace

This is another operation in python numpy which returns evenly spaced numbers over a specified interval. Consider the below example:

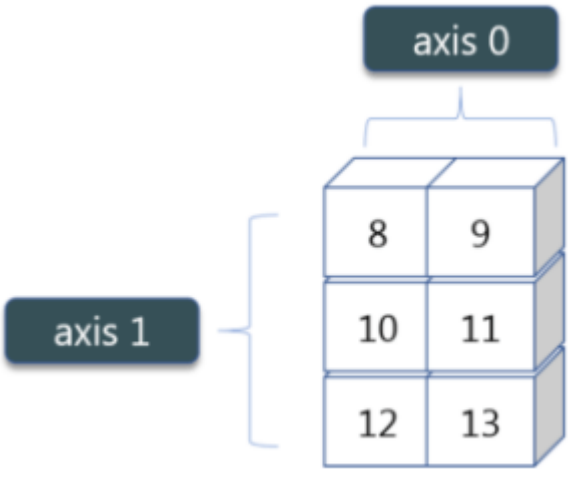
```
In [671]: a=np.linspace(1,3,10)
          print(a)

          [1.         1.22222222 1.44444444 1.66666667 1.88888889 2.11111111
           2.33333333 2.55555556 2.77777778 3.        ]
```

As you can see in the result, it has printed 10 values between 1 to 3.

### 10.6.7 max/ min

Next, we have some more operations in numpy such as to find the minimum, maximum as well the sum of the numpy array. Let's go ahead in python numpy tutorial and execute it practically.

```
In [672]: a=np.array([1,2,3])
          print(a.min())
          print(a.max())
          print(a.sum())

          1
          3
          6
```




As you can see in the figure, we have a numpy array 2*3. Here the rows are called as axis 1 and the columns are called as axis 0. Now you must be wondering what is the use of these axis?

Suppose you want to calculate the sum of all the columns, then you can make use of axis.

```
[42]: import numpy as np
      a=np.array([(1,2,3),(3,4,5)])
      print(a.sum(axis=0))

      [4 6 8]
```

Therefore, the sum of all the columns are added where 1+3=4, 2+4=6 and 3+5=8. Similarly, if you replace the axis by 1, then it will print [6 12] where all the rows get added.

### 10.6.8 Square Root & Standard Deviation

There are various mathematical functions that can be performed using python numpy. You can find the square root, standard deviation of the array. So, let's implement these operations:

```
In [673]: a=np.array([(1,2,3),(3,4,5)])
          print(np.sqrt(a))
          print(np.std(a))

          [[1.         1.41421356 1.73205081]
           [1.73205081 2.         2.23606798]]
          1.2909944487358056
```

As you can see the output above, the square root of all the elements are printed. Also, the standard deviation is printed for the above array i.e how much each element varies from the mean value of the python numpy array.

### 10.6.9 Addition Operation

You can perform more operations on numpy array i.e addition, subtraction, multiplication and division of the two matrices. Let me go ahead in python numpy tutorial, and show it to you practically:

```
In [674]: a=np.array([(1,2,3),(3,4,5)])
          b=np.array([(1,2,3),(3,4,5)])
          print(a+b)

          [[ 2  4  6]
           [ 6  8 10]]
```

This is extremely simple! Right? Similarly, we can perform other operations such as subtraction, multiplication and division. Consider the below example:

```
In [675]: a=np.array([(1,2,3),(3,4,5)])
          b=np.array([(1,2,3),(3,4,5)])
          print(a-b)
          print(a*b)
          print(a/b)

          [[0 0 0]
           [0 0 0]]
          [[ 1  4  9]
           [ 9 16 25]]
          [[1. 1. 1.]
           [1. 1. 1.]]
```

### 10.6.10 Vertical & Horizontal Stacking

Next, if you want to concatenate two arrays and not just add them, you can perform it using two ways – *vertical stacking* and *horizontal stacking*

```
In [676]: x=np.array([(1,2,3),(3,4,5)])
          y=np.array([(1,2,3),(3,4,5)])
          print(np.vstack((x,y)))
          print(np.hstack((x,y)))
```

```
[[1 2 3]
 [3 4 5]
 [1 2 3]
 [3 4 5]]
[[1 2 3 1 2 3]
 [3 4 5 3 4 5]]
```

### 10.6.11 Ravel

There is one more operation where you can convert one numpy array into a single column i.e *ravel*. Let me show how it is implemented practically:

```
In [677]: x=np.array([(1,2,3),(3,4,5)])
          print(x.ravel())
```

```
[1 2 3 3 4 5]
```

## 10.7 Tasks:

# 11. Lab#11 - Pandas

## Objectives:

- To learn and able to create series and data frame
- To learn and able to read the data from different format
- To learn and able to use indexing, selection and sorting
- To learn and able to handle missing data and duplicate data
- To learn and able to merge the data frame

## Outcomes:

- Students should be able to create series and data frame
- Students should be able to read the data from different format
- Students should be able to use indexing, selection and sorting
- Students should be able to handle missing data and duplicate data
- Students should be able to merge the data frame
-

## 11.1 What are Pandas and uses?

pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, **real-world** data analysis in Python. Additionally, it has the broader goal of becoming **the most powerful and flexible open source data analysis/manipulation tool available in any language**. It is already well on its way toward this goal.

pandas is well suited for many different kinds of data:

- Tabular data with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet
- Ordered and unordered (not necessarily fixed-frequency) time series data.
- Arbitrary matrix data (homogeneously typed or heterogeneous) with row and column labels
- Any other form of observational / statistical data sets. The data need not be labeled at all to be placed into a pandas data structure

The two primary data structures of pandas, **Series** (1-dimensional) and **DataFrame** (2-dimensional), handle the vast majority of typical use cases in finance, statistics, social science, and many areas of engineering. For R users, **DataFrame** provides everything that R's data.frame provides and much more. pandas is built on top of NumPy and is intended to integrate well within a scientific computing environment with many other 3rd party libraries.

Here are just a few of the things that pandas does well:

- Easy handling of **missing data** (represented as NaN) in floating point as well as non-floating point data
- Size mutability: columns can be **inserted and deleted** from DataFrame and higher dimensional objects
- Automatic and explicit **data alignment**: objects can be explicitly aligned to a set of labels, or the user can simply ignore the labels and let Series, DataFrame, etc. automatically align the data for you in computations
- Powerful, flexible **group by** functionality to perform split-apply-combine operations on data sets, for both aggregating and transforming data

- Make it **easy to convert** ragged, differently-indexed data in other Python and NumPy data structures into DataFrame objects
- Intelligent label-based **slicing**, **fancy indexing**, and **subsetting** of large data sets
- Intuitive **merging** and **joining** data sets
- Flexible **reshaping** and pivoting of data sets
- **Hierarchical** labeling of axes (possible to have multiple labels per tick)
- Robust IO tools for loading data from **flat files** (CSV and delimited), Excel files, databases, and saving / loading data from the ultrafast **HDF5 format**
- **Time series**-specific functionality: date range generation and frequency conversion, moving window statistics, date shifting, and lagging.

Many of these principles are here to address the shortcomings frequently experienced using other languages / scientific research environments. For data scientists, working with data is typically divided into multiple stages: managing and cleaning data, analyzing / modeling it, then organizing the results of the analysis into a form suitable for plotting or tabular display. pandas is the ideal tool for all of these tasks.

Some other notes

- pandas is **fast**. Many of the low-level algorithmic bits have been extensively tweaked in Cython code. However, as with anything else generalization usually sacrifices performance. So if you focus on one feature for your application you may be able to create a faster specialized tool.
- pandas is a dependency of statsmodels, making it an important part of the statistical computing ecosystem in Python.
- pandas has been used extensively in production in financial applications.

**NOTE:** Use the pandas documentation for this lab https://pandas.pydata.org/docs/

## 11.2 Pandas Functions

### 11.2.1 What is Pandas?

Pandas is one of the most important libraries of Python.

Pandas has data structures for easy data analysis. The most used of these are Series and DataFrame data structures. Series data structure is one dimensional, that is, it consists of a column. DataFrame data structure is two-dimensional, i.e. it consists of rows and columns.

To install Pandas you can use "pip install pandas"

```
In [2]: import pandas as pd # Let's import pandas with pd import numpy as np
        pd.__version__ # To print the installed vesion pandas

Out[2]: '1.2.4'
```

## 11.2.2 Series Data Structure

### 11.2.2.1 Creating pandas series

```
In [3]: obj=pd.Series([1,"John",3.5,"Hey"])
        obj

Out[3]: 0       1
        1    John
        2     3.5
        3     Hey
        dtype: object
```

### 11.2.2.2 Indexing pandas series

```
In [4]: obj[0]

Out[4]: 1

In [5]: obj.values

Out[5]: array([1, 'John', 3.5, 'Hey'], dtype=object)

In [6]: obj2=pd.Series([1,"John",3.5,"Hey"],index=["a","b","c","d"])

        obj2

Out[6]: a       1
        b    John
        c     3.5
        d     Hey
        dtype: object

In [7]: obj2["b"]

Out[7]: 'John'

In [8]: obj2.index

Out[8]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

### 11.2.2.3 Creating pandas series from dictionary object

```
[14]: score={"Jane":90, "Bill" :80,"Elon": 85,"Tom":75,"Tim":95}
      names=pd.Series(score) # Convert to Series names
      names
```

```
[14]: Jane    90
      Bill    80
      Elon    85
      Tom     75
      Tim     95
      dtype: int64
```

```
[13]: names["Tim"]
```

```
[13]: 95
```

### 11.2.2.4 Operations on pandas series

```
In [15]: names[names>=85]
```

```
Out[15]: Jane    90
         Elon    85
         Tim     95
         dtype: int64
```

```
In [17]: names["Tom"]=60
         names
```

```
Out[17]: Jane    90
         Bill    80
         Elon    85
         Tom     60
         Tim     95
         dtype: int64
```

```
In [18]: names[names<=80]=83
         names
```

```
Out[18]: Jane    90
         Bill    83
         Elon    85
         Tom     83
         Tim     95
         dtype: int64
```

```
In [19]: "Tom" in names
```

```
Out[19]: True
```

```
In [20]: "Can" in names
```

```
Out[20]: False
```

```
In [21]: names/10
Out[21]: Jane    9.0
         Bill    8.3
         Elon    8.5
         Tom     8.3
         Tim     9.5
         dtype: float64

In [22]: names**2
Out[22]: Jane    8100
         Bill    6889
         Elon    7225
         Tom     6889
         Tim     9025
         dtype: int64

In [23]: names.isnull()

Out[23]: Jane    False
         Bill    False
         Elon    False
         Tom     False
         Tim     False
         dtype: bool
```

## 11.2.2.5 Working with Series Data Structure (Real Data)

```
In [25]: games=pd.read_csv("vgsalesGlobale.csv")
```

```
In [26]: games.head()
```

Out[26]:

| | Rank | Name | Platform | Year | Genre | Publisher | NA_Sales | EU_Sales | JP_Sales | Other_Sales | Global_Sales |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | Wii Sports | Wii | 2006.0 | Sports | Nintendo | 41.49 | 29.02 | 3.77 | 8.46 | 82.74 |
| 1 | 2 | Super Mario Bros. | NES | 1985.0 | Platform | Nintendo | 29.08 | 3.58 | 6.81 | 0.77 | 40.24 |
| 2 | 3 | Mario Kart Wii | Wii | 2008.0 | Racing | Nintendo | 15.85 | 12.88 | 3.79 | 3.31 | 35.82 |
| 3 | 4 | Wii Sports Resort | Wii | 2009.0 | Sports | Nintendo | 15.75 | 11.01 | 3.28 | 2.96 | 33.00 |
| 4 | 5 | Pokemon Red/Pokemon Blue | GB | 1996.0 | Role-Playing | Nintendo | 11.27 | 8.89 | 10.22 | 1.00 | 31.37 |

```
In [27]: games.dtypes
Out[27]: Rank            int64
         Name           object
         Platform       object
         Year          float64
         Genre          object
         Publisher      object
         NA_Sales      float64
         EU_Sales      float64
         JP_Sales      float64
         Other_Sales   float64
         Global_Sales  float64
         dtype: object
```

In [28]:
```
games.describe()
```

Out[28]:

| | Rank | Year | NA_Sales | EU_Sales | JP_Sales | Other_Sales | Global_Sales |
|---|---|---|---|---|---|---|---|
| count | 16598.000000 | 16327.000000 | 16598.000000 | 16598.000000 | 16598.000000 | 16598.000000 | 16598.000000 |
| mean | 8300.605254 | 2006.406443 | 0.264667 | 0.146652 | 0.077782 | 0.048063 | 0.537441 |
| std | 4791.853933 | 5.828981 | 0.816683 | 0.505351 | 0.309291 | 0.188588 | 1.555028 |
| min | 1.000000 | 1980.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.010000 |
| 25% | 4151.250000 | 2003.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.060000 |
| 50% | 8300.500000 | 2007.000000 | 0.080000 | 0.020000 | 0.000000 | 0.010000 | 0.170000 |
| 75% | 12449.750000 | 2010.000000 | 0.240000 | 0.110000 | 0.040000 | 0.040000 | 0.470000 |
| max | 16600.000000 | 2020.000000 | 41.490000 | 29.020000 | 10.220000 | 10.570000 | 82.740000 |

In [29]:
```
games.Genre.describe()
```

Out[29]:
```
count      16598
unique        12
top       Action
freq        3316
Name: Genre, dtype: object
```

In [30]:
```
games.Genre.value_counts()
```

Out[30]:
```
Action          3316
Sports          2346
Misc            1739
Role-Playing    1488
Shooter         1310
Adventure       1286
Racing          1249
Platform         886
Simulation       867
Fighting         848
Strategy         681
Puzzle           582
```

In [31]:
```
games.Genre.value_counts(normalize=True)
```

Out[31]:
```
Action          0.199783
Sports          0.141342
Misc            0.104772
Role-Playing    0.089649
Shooter         0.078925
Adventure       0.077479
Racing          0.075250
Platform        0.053380
Simulation      0.052235
Fighting        0.051090
Strategy        0.041029
Puzzle          0.035064
Name: Genre, dtype: float64
```

```
[32]: type(games.Genre.value_counts())

[32]: pandas.core.series.Series

[33]: games.Genre.value_counts().head()

[33]: Action          3316
      Sports          2346
      Misc            1739
      Role-Playing    1488
      Shooter         1310
      Name: Genre, dtype: int64

[34]: games.Genre.unique()

[34]: array(['Sports', 'Platform', 'Racing', 'Role-Playing', 'Puzzle', 'Misc',
             'Shooter', 'Simulation', 'Action', 'Fighting', 'Adventure',
             'Strategy'], dtype=object)
```

```
In [35]: games.Genre.nunique()

Out[35]: 12
```

### 11.2.3 DataFrame

#### 11.2.3.1 DataFrame Creation and basic operations?

```
[36]: # Creating dataframe from dictionary

      data={"name":["Bill","Tom","Tim","John","Alex","Vanessa","Kate"],

      "score":[90,80,85,75,95,60,65],

      "sport":["Wrestling","Football","Skiing","Swimming","Tennis",

      "Karete","Surfing"],

      "sex":["M","M","M","M","F","F","F"]}

      df=pd.DataFrame(data)
```

```
In [37]: #Viewing dataframe
         df
```

Out[37]:

| | name | score | sport | sex |
|---|---|---|---|---|
| 0 | Bill | 90 | Wrestling | M |
| 1 | Tom | 80 | Football | M |
| 2 | Tim | 85 | Skiing | M |
| 3 | John | 75 | Swimming | M |
| 4 | Alex | 95 | Tennis | F |
| 5 | Vanessa | 60 | Karete | F |
| 6 | Kate | 65 | Surfing | F |

In [38]:
```
# Create dataframe and set columns separately

df=pd.DataFrame(data,columns=["name","sport","sex","score"])

df
```

Out[38]:

| | name | sport | sex | score |
|---|---|---|---|---|
| 0 | Bill | Wrestling | M | 90 |
| 1 | Tom | Football | M | 80 |
| 2 | Tim | Skiing | M | 85 |
| 3 | John | Swimming | M | 75 |
| 4 | Alex | Tennis | F | 95 |
| 5 | Vanessa | Karete | F | 60 |
| 6 | Kate | Surfing | F | 65 |

---

In [39]:
```
df.head()
```

Out[39]:

| | name | sport | sex | score |
|---|---|---|---|---|
| 0 | Bill | Wrestling | M | 90 |
| 1 | Tom | Football | M | 80 |
| 2 | Tim | Skiing | M | 85 |
| 3 | John | Swimming | M | 75 |
| 4 | Alex | Tennis | F | 95 |

---

In [40]:
```
#⟶Print last n rows of the dataframe (default = 5)
df.tail()
```

Out[40]:

| | name | sport | sex | score |
|---|---|---|---|---|
| 2 | Tim | Skiing | M | 85 |
| 3 | John | Swimming | M | 75 |
| 4 | Alex | Tennis | F | 95 |
| 5 | Vanessa | Karete | F | 60 |
| 6 | Kate | Surfing | F | 65 |

```
In [42]: # Setting index labels while creating dataframe
df=pd.DataFrame(data,columns=["name", "sport", "gender", "score", "age"], index=["one","two","three","four","five","six","seven"]

df
```

Out[42]:

| | name | sport | gender | score | age |
|---|---|---|---|---|---|
| one | Bill | Wrestling | NaN | 90 | NaN |
| two | Tom | Football | NaN | 80 | NaN |
| three | Tim | Skiing | NaN | 85 | NaN |
| four | John | Swimming | NaN | 75 | NaN |
| five | Alex | Tennis | NaN | 95 | NaN |
| six | Vanessa | Karete | NaN | 60 | NaN |
| seven | Kate | Surfing | NaN | 65 | NaN |

```
In [44]: my_columns=["name","sport"]
df[my_columns]
```

Out[44]:

| | name | sport |
|---|---|---|
| one | Bill | Wrestling |
| two | Tom | Football |
| three | Tim | Skiing |
| four | John | Swimming |
| five | Alex | Tennis |
| six | Vanessa | Karete |
| seven | Kate | Surfing |

---

## 11.2.3.2 Operations on dataframe

```
In [45]: #⟶Getting dataframe rows using .loc

#⟶Note: it uses index labels

df.loc[["one"]]
```

Out[45]:

| | name | sport | gender | score | age |
|---|---|---|---|---|---|
| one | Bill | Wrestling | NaN | 90 | NaN |

I

```
In [46]: df.loc[["one","two"]]
```

Out[46]:

| | name | sport | gender | score | age |
|---|---|---|---|---|---|
| one | Bill | Wrestling | NaN | 90 | NaN |
| two | Tom | Football | NaN | 80 | NaN |

### 11.2.3.2.1 DataFrame Indexing

In [59]:
```
import numpy as np
data=pd.DataFrame(np.arange(16).reshape(4,4),index=["London","Paris","Berlin","Istanbul"],
columns=["one","two","three","four"])
data
```

Out[59]:

| | one | two | three | four |
|---|---|---|---|---|
| London | 0 | 1 | 2 | 3 |
| Paris | 4 | 5 | 6 | 7 |
| Berlin | 8 | 9 | 10 | 11 |
| Istanbul | 12 | 13 | 14 | 15 |

In [60]:
```
data["two"]
```

Out[60]:
```
London      1
Paris       5
Berlin      9
Istanbul    13
Name: two, dtype: int32
```

In [62]:
```
data[["one","two"]]
```

Out[62]:

| | one | two |
|---|---|---|
| London | 0 | 1 |
| Paris | 4 | 5 |
| Berlin | 8 | 9 |
| Istanbul | 12 | 13 |

In [64]:
```
data[:3]
```

Out[64]:

| | one | two | three | four |
|---|---|---|---|---|
| London | 0 | 1 | 2 | 3 |
| Paris | 4 | 5 | 6 | 7 |
| Berlin | 8 | 9 | 10 | 11 |

In [65]:
```
data[data["four"]>5]
```

Out[65]:

| | one | two | three | four |
|---|---|---|---|---|
| Paris | 4 | 5 | 6 | 7 |
| Berlin | 8 | 9 | 10 | 11 |
| Istanbul | 12 | 13 | 14 | 15 |

```
In [67]: data[data<5]=0
         data
```

Out[67]:

| | one | two | three | four |
|---|---|---|---|---|
| London | 0 | 0 | 0 | 0 |
| Paris | 0 | 5 | 6 | 7 |
| Berlin | 8 | 9 | 10 | 11 |
| Istanbul | 12 | 13 | 14 | 15 |

11.2.3.2.2 Selecting with iloc and loc

```
In [68]: data.iloc[1]
```

```
Out[68]: one      0
         two      5
         three    6
         four     7
         Name: Paris, dtype: int32
```

```
In [69]: data.iloc[[1,3],[1,2,3]]
```

Out[69]:

| | two | three | four |
|---|---|---|---|
| Paris | 5 | 6 | 7 |
| Istanbul | 13 | 14 | 15 |

```
In [70]: data.iloc[1,[1,2,3]]
```

```
Out[70]: two      5
         three    6
         four     7
         Name: Paris, dtype: int32
```

```
In [71]: data.loc["Paris",["one","two"]]
```

```
Out[71]: one    0
         two    5
         Name: Paris, dtype: int32
```

```
In [72]: data.loc[:"Paris","four"]
```

```
Out[72]: London    0
         Paris     7
         Name: four, dtype: int32
```

### 11.2.3.3 Important Methods in Pandas

In [73]:
```
df=pd.DataFrame(np.arange(9).reshape(3,3),

index=["a","c","d"],

columns=["Tim","Tom","Kate"])

df
```

Out[73]:

| | Tim | Tom | Kate |
|---|---|---|---|
| a | 0 | 1 | 2 |
| c | 3 | 4 | 5 |
| d | 6 | 7 | 8 |

---

In [74]:
```
df2=df.reindex(["d","c","b","a"])
df2
```

Out[74]:

| | Tim | Tom | Kate |
|---|---|---|---|
| d | 6.0 | 7.0 | 8.0 |
| c | 3.0 | 4.0 | 5.0 |
| b | NaN | NaN | NaN |
| a | 0.0 | 1.0 | 2.0 |

[77]:
```
data=pd.DataFrame(np.arange(16).reshape(4,4), index=["Kate","Tim",

"Tom","Alex"],

columns=list("ABCD"))

data
```

[77]:

| | A | B | C | D |
|---|---|---|---|---|
| Kate | 0 | 1 | 2 | 3 |
| Tim | 4 | 5 | 6 | 7 |
| Tom | 8 | 9 | 10 | 11 |
| Alex | 12 | 13 | 14 | 15 |

In [78]:
```
data.drop("A",axis=1)
```

Out[78]:

| | B | C | D |
|---|---|---|---|
| Kate | 1 | 2 | 3 |
| Tim | 5 | 6 | 7 |
| Tom | 9 | 10 | 11 |
| Alex | 13 | 14 | 15 |

11.2.3.3.1 Applying a Function on DataFrame

In [79]:
```
df=pd.DataFrame(

np.random.randn(4,3),

columns=list("ABC"),

index=["Kim","Susan","Tim","Tom"])

df
```

Out[79]:

| | A | B | C |
|---|---|---|---|
| Kim | -0.554483 | 0.172139 | -1.109825 |
| Susan | 1.646337 | 0.791896 | -0.843393 |
| Tim | -1.358107 | -0.669194 | -0.208259 |
| Tom | 0.610991 | 0.960443 | -0.942074 |

In [80]:
```
np.abs(df)
```

Out[80]:

| | A | B | C |
|---|---|---|---|
| Kim | 0.554483 | 0.172139 | 1.109825 |
| Susan | 1.646337 | 0.791896 | 0.843393 |
| Tim | 1.358107 | 0.669194 | 0.208259 |
| Tom | 0.610991 | 0.960443 | 0.942074 |

| | | | |
|---|---|---|---|
| Tom | 0.610991 | 0.960443 | 0.942074 |

In [81]:
```
f=lambda x:x.max()-x.min()
```

In [82]:
```
df.apply(f)
```

Out[82]:
```
A    3.004444
B    1.629637
C    0.901566
dtype: float64
```

In [83]:
```
df.apply(f,axis=1)
```

Out[83]:
```
Kim      1.281963
Susan    2.489730
Tim      1.149848
Tom      1.902517
dtype: float64
```

### 11.2.3.3.2 Sorting and Ranking

```
In [84]: df=pd.DataFrame(
         np.arange(12).reshape(3,4),
         index=["two","one","three"],
         columns=["d","a","b","c"])
         df
```

Out[84]:

| | d | a | b | c |
|---|---|---|---|---|
| two | 0 | 1 | 2 | 3 |
| one | 4 | 5 | 6 | 7 |
| three | 8 | 9 | 10 | 11 |

```
In [87]: df.sort_index(axis=1)
```

Out[87]:

| | a | b | c | d |
|---|---|---|---|---|
| two | 1 | 2 | 3 | 0 |
| one | 5 | 6 | 7 | 4 |
| three | 9 | 10 | 11 | 8 |

```
In [88]: df.sort_index(axis=1, ascending=False)
```

Out[88]:

| | d | c | b | a |
|---|---|---|---|---|
| two | 0 | 3 | 2 | 1 |
| one | 4 | 7 | 6 | 5 |
| three | 8 | 11 | 10 | 9 |

### 11.2.3.3.3 Data Reading and Writing

The Text file is shown below

```
0,Tom,80,M
1,Tim,85,M
2,Kim,70,M
3,Kate,90,F
4,Alex,75,F
```

[5]:
```
df2=pd.read_table("DataSets/data.txt")
df2
```

[5]:

| | 0,Tom,80,M |
|---|---|
| 0 | 1,Tim,85,M |
| 1 | 2,Kim,70,M |
| 2 | 3,Kate,90,F |
| 3 | 4,Alex,75,F |

[6]:
```
df2=pd.read_table("DataSets/data.txt", sep=",")
df2
```

[6]:

| | 0 | Tom | 80 | M |
|---|---|---|---|---|
| 0 | 1 | Tim | 85 | M |
| 1 | 2 | Kim | 70 | M |
| 2 | 3 | Kate | 90 | F |
| 3 | 4 | Alex | 75 | F |

[8]:
```
df=pd.read_table("DataSets/data.txt",sep=",",header=None)
df
```

[8]:

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | Tom | 80 | M |
| 1 | 1 | Tim | 85 | M |
| 2 | 2 | Kim | 70 | M |
| 3 | 3 | Kate | 90 | F |
| 4 | 4 | Alex | 75 | F |

[9]:
```
df=pd.read_table("DataSets/data.txt",
                 sep=",",
                 header=None,
                 names=["name","score","sex"])
df
```

[9]:

| | name | score | sex |
|---|---|---|---|
| 0 | Tom | 80 | M |
| 1 | Tim | 85 | M |
| 2 | Kim | 70 | M |
| 3 | Kate | 90 | F |
| 4 | Alex | 75 | F |

11.2.3.3.4 Writing Data

```
[10]: df=pd.read_csv("DataSets/data.txt",sep=",")
      df
```

[10]:

| | 0 | Tom | 80 | M |
|---|---|---|---|---|
| 0 | 1 | Tim | 85 | M |
| 1 | 2 | Kim | 70 | M |
| 2 | 3 | Kate | 90 | F |
| 3 | 4 | Alex | 75 | F |

```
[11]: df.to_csv("DataSets/new_data.csv",index=False)
```

## 11.2.4 Missing Data

```
[14]: df=pd.DataFrame([[1,2,3],[4,NA,5],
                       [NA,NA,NA]])
      df
```

[14]:

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1.0 | 2.0 | 3.0 |
| 1 | 4.0 | NaN | 5.0 |
| 2 | NaN | NaN | NaN |

```
[15]: df.fillna('')
```

[15]:

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1.0 | 2.0 | 3.0 |
| 1 | 4.0 | | 5.0 |
| 2 | | | |

```
[17]: df.drop_duplicates()
```

[17]:

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1.0 | 2.0 | 3.0 |
| 1 | 4.0 | NaN | 5.0 |
| 2 | NaN | NaN | NaN |

```
[18]: df.dropna(how="all")
```

[18]:

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1.0 | 2.0 | 3.0 |
| 1 | 4.0 | NaN | 5.0 |

[20]:
```
df[1]=NA
df
```

[20]:

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1.0 | NaN | 3.0 |
| 1 | 4.0 | NaN | 5.0 |
| 2 | NaN | NaN | NaN |

[21]:
```
df.dropna(axis=1,how="all")
```

[21]:

| | 0 | 2 |
|---|---|---|
| 0 | 1.0 | 3.0 |
| 1 | 4.0 | 5.0 |
| 2 | NaN | NaN |

[24]:
```
#Drop a row if non-nan are not atleast equal to thresh

df.dropna(thresh=1)
```

[24]:

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1.0 | NaN | 3.0 |
| 1 | 4.0 | NaN | 5.0 |

[25]:
```
df.fillna(0)
```

[25]:

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1.0 | 0.0 | 3.0 |
| 1 | 4.0 | 0.0 | 5.0 |
| 2 | 0.0 | 0.0 | 0.0 |

[27]:
```
df.fillna({0:15,1:25,2:35})
```

[27]:

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1.0 | 25.0 | 3.0 |
| 1 | 4.0 | 25.0 | 5.0 |
| 2 | 15.0 | 25.0 | 35.0 |

[28]:
```
df.fillna(0,inplace=True)
df
```

[28]:

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1.0 | 0.0 | 3.0 |
| 1 | 4.0 | 0.0 | 5.0 |
| 2 | 0.0 | 0.0 | 0.0 |

## 11.2.5 Combining and Merging Datasets

### 11.2.5.1 Joining DataFrame

[35]:
```
d1=pd.DataFrame(
    {"key":["a","b","c","c","d","e"],
     "num1":range(6)})
d2=pd.DataFrame(
    {"key":["b","c","e","f"],
     "num2":range(4)})
```

[36]:
```
d1
```

[36]:

| | key | num1 |
|---|---|---|
| **0** | a | 0 |
| **1** | b | 1 |
| **2** | c | 2 |
| **3** | c | 3 |
| **4** | d | 4 |
| **5** | e | 5 |

[37]:
```
d2
```

[37]:

| | key | num2 |
|---|---|---|
| **0** | b | 0 |
| **1** | c | 1 |
| **2** | e | 2 |
| **3** | f | 3 |

[38]:
```
pd.merge(d1, d2)
```

[38]:

| | key | num1 | num2 |
|---|---|---|---|
| **0** | b | 1 | 0 |
| **1** | c | 2 | 1 |
| **2** | c | 3 | 1 |
| **3** | e | 5 | 2 |

[39]:
```
pd.merge(d1, d2, on='key')
```

[39]:

| | key | num1 | num2 |
|---|---|---|---|
| **0** | b | 1 | 0 |
| **1** | c | 2 | 1 |
| **2** | c | 3 | 1 |
| **3** | e | 5 | 2 |

## 11.2.6 Practice

Let's practice using a real data set. You can download data set from **https://www.kaggle.com/melodyxyz/global**.

```
In [97]: data=pd.read_csv("vgsalesGlobale.csv")
```

```
In [98]: data.head()
```

Out[98]:

| | Rank | Name | Platform | Year | Genre | Publisher | NA_Sales | EU_Sales | JP_Sales | Other_Sales | Global_Sales |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | Wii Sports | Wii | 2006.0 | Sports | Nintendo | 41.49 | 29.02 | 3.77 | 8.46 | 82.74 |
| 1 | 2 | Super Mario Bros. | NES | 1985.0 | Platform | Nintendo | 29.08 | 3.58 | 6.81 | 0.77 | 40.24 |
| 2 | 3 | Mario Kart Wii | Wii | 2008.0 | Racing | Nintendo | 15.85 | 12.88 | 3.79 | 3.31 | 35.82 |
| 3 | 4 | Wii Sports Resort | Wii | 2009.0 | Sports | Nintendo | 15.75 | 11.01 | 3.28 | 2.96 | 33.00 |
| 4 | 5 | Pokemon Red/Pokemon Blue | GB | 1996.0 | Role-Playing | Nintendo | 11.27 | 8.89 | 10.22 | 1.00 | 31.37 |

```
In [ ]:
```

```
In [99]: data["Name"].sort_values(ascending=False)
```

```
Out[99]: 9135                ¡Shin Chan Flipa en colores!
         470                   wwe Smackdown vs. Raw 2006
         15523               uDraw Studio: Instant Artist
         7835                uDraw Studio: Instant Artist
         627                               uDraw Studio
                                    ...
         8304               .hack//G.U. Vol.3//Redemption
         8602     .hack//G.U. Vol.2//Reminisce (jp sales)
         7107                .hack//G.U. Vol.2//Reminisce
         8357                  .hack//G.U. Vol.1//Rebirth
         4754                                '98 Koshien
         Name: Name, Length: 16598, dtype: object
```

```
In [100]: data.sort_values("Name")
```

Out[100]:

| | Rank | Name | Platform | Year | Genre | Publisher | NA_Sales | EU_Sales | JP_Sales | Other_Sales | Global_Sales |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 4754 | 4756 | '98 Koshien | PS | 1998.0 | Sports | Magical Company | 0.15 | 0.10 | 0.12 | 0.03 | 0.41 |
| 8357 | 8359 | .hack//G.U. Vol.1//Rebirth | PS2 | 2006.0 | Role-Playing | Namco Bandai Games | 0.00 | 0.00 | 0.17 | 0.00 | 0.17 |
| 7107 | 7109 | .hack//G.U. Vol.2//Reminisce | PS2 | 2006.0 | Role-Playing | Namco Bandai Games | 0.11 | 0.09 | 0.00 | 0.03 | 0.23 |
| 8602 | 8604 | .hack//G.U. Vol.2//Reminisce (jp sales) | PS2 | 2006.0 | Role-Playing | Namco Bandai Games | 0.00 | 0.00 | 0.16 | 0.00 | 0.16 |
| 8304 | 8306 | .hack//G.U. Vol.3//Redemption | PS2 | 2007.0 | Role-Playing | Namco Bandai Games | 0.00 | 0.00 | 0.17 | 0.00 | 0.17 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 627 | 628 | uDraw Studio | Wii | 2010.0 | Misc | THQ | 1.67 | 0.58 | 0.00 | 0.20 | 2.46 |
| 7835 | 7837 | uDraw Studio: Instant Artist | Wii | 2011.0 | Misc | THQ | 0.08 | 0.09 | 0.00 | 0.02 | 0.19 |
| 15523 | 15526 | uDraw Studio: Instant Artist | X360 | 2011.0 | Misc | THQ | 0.01 | 0.01 | 0.00 | 0.00 | 0.02 |
| 470 | 471 | wwe Smackdown vs. Raw 2006 | PS2 | NaN | Fighting | NaN | 1.57 | 1.02 | 0.00 | 0.41 | 3.00 |
| 9135 | 9137 | ¡Shin Chan Flipa en colores! | DS | 2007.0 | Platform | 505 Games | 0.00 | 0.00 | 0.14 | 0.00 | 0.14 |

16598 rows × 11 columns

## 11.3 Tasks:

# 12. Lab#12 - Matplotlib

---

## Objectives:

- To learn and able to create the bar and line chart
- To learn and able to create the pie, histogram and scatter plot

## Outcomes:

- Students should be able to create the bar and line chart
- Students should be able to create the pie, histogram and scatter plot

---

## 12.1 What Is Python Matplotlib?

`matplotlib.pyplot` is a plotting library used for 2D graphics in python programming language. It can be used in python scripts, shell, web application servers and other graphical user interface toolkits.

## 12.2 What is Matplotlib used for?

Matplotlib is a Python Library used for plotting, this python library provides and objected-oriented APIs for integrating plots into applications.

## 12.3 Matplotlib Function

The objective of this lab is to plot different types of plots using matplotlib. This Kernel is a work in process and if you like my work please do vote.

**importing matplotlib module**

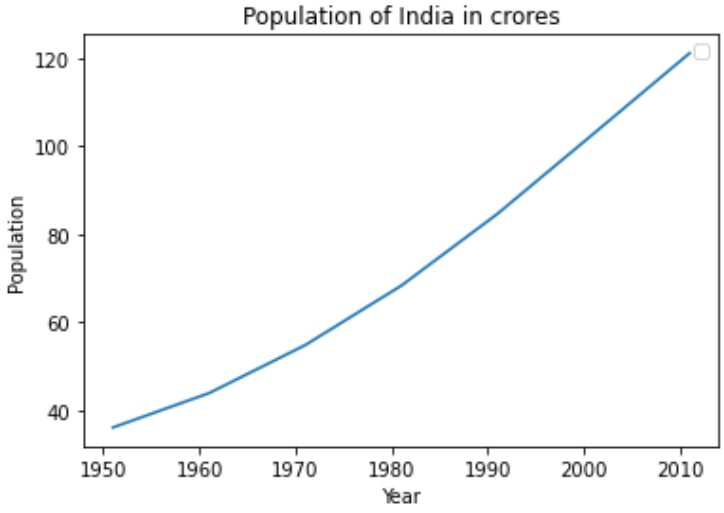
```
import matplotlib.pyplot as plt
import numpy as np
```

### 12.3.1 Making a simple X,Y plot

X, Y plot is generally used when there is a change in value of a variable with respect to time or some other variable.

```
In [706]: year=[1951,1961,1971,1981,1991,2001,2011]
          population=[36.1,43.9,54.8,68.3,84.6,102.8,121.1]
          plt.plot(year,population)  # for plotting
          plt.xlabel('Year')         # x-axis label
          plt.ylabel('Population')   # y-axis label
          plt.title('Population of India in crores')  # Diagram Title
          plt.legend()   # for legend
          plt.show()   # for show()
```

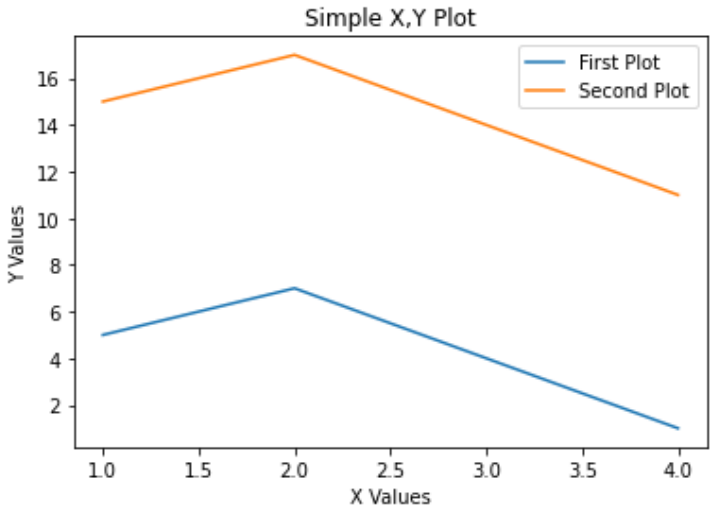
```
No handles with labels found to put in legend.
```




## 12.3.2 Adding Legends, Titles and labels to plot

A plot is nothing without proper legend ,titles and labels. Code below shows the way in which labels, titles and labels are applied to a plot.

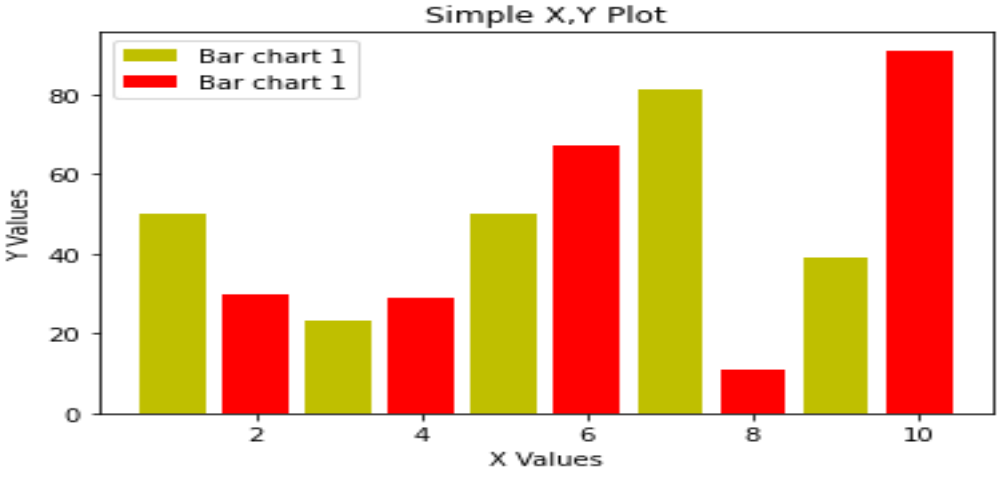
```
In [709]: # Defining the first data set
          x=[1,2,4]
          y=[5,7,1]

          #Defining the second data set
          x1=[1,2,4]
          y1=[15,17,11]

          #Plotting the graphs
          plt.plot(x,y,label='First Plot')
          plt.plot(x1,y1,label='Second Plot')

          #Defining the X,Y labels and title
          plt.xlabel('X Values')
          plt.ylabel('Y Values')
          plt.title('Simple X,Y Plot')
          plt.legend()
          plt.ioff()
```




## 12.3.3 Plotting Bar Chart

A bar chart or bar graph is a chart or graph that presents categorical data with rectangular bars with heights or lengths proportional to the values that they represent. The bars can be plotted vertically or horizontally. A vertical bar chart is sometimes called a column chart.

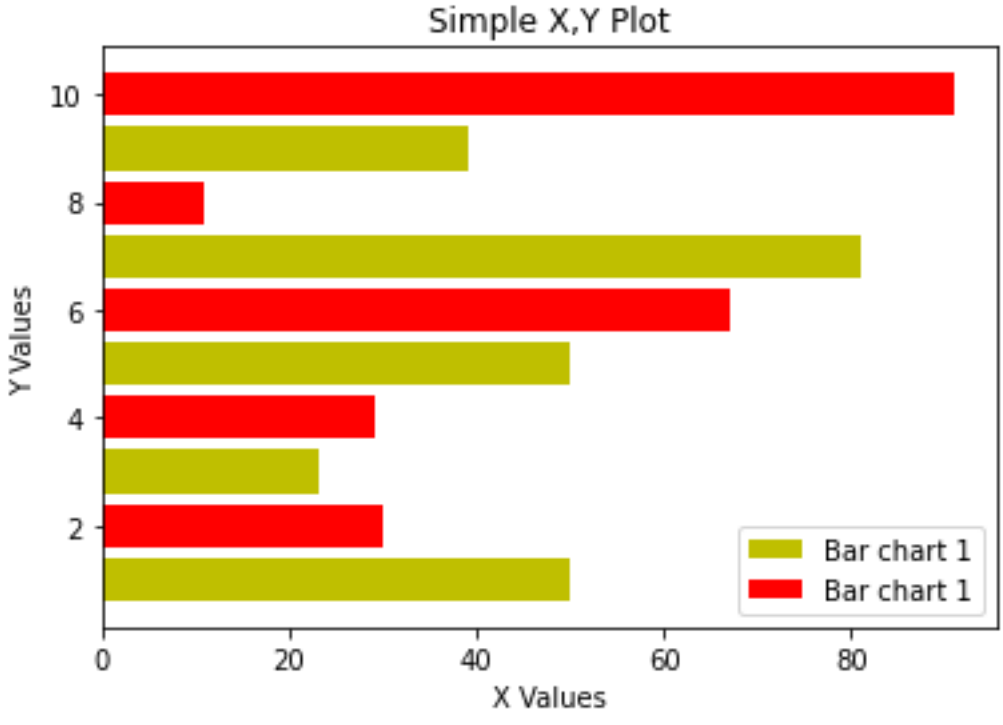
In [710]:
```
# Defining the first data set
x1=[1,3,5,7,9]
y1=[50,23,50,81,39]

# Defining the second data set
x2=[2,4,6,8,10]
y2=[30,29,67,11,91]

#Plotting the bar charts with labels and colors
plt.bar(x1,y1,label='Bar chart 1',color='y')
plt.bar(x2,y2,label='Bar chart 1',color='r')

#Defining the x,y labels,title and legend
plt.xlabel('X Values')
plt.ylabel('Y Values')
plt.title('Simple X,Y Plot')
plt.legend()
plt.ioff()
```



## 12.3.4 Horizontal bar plot

A horizontal bar chart is a graph in the form of rectangular bars. The bar chart title indicates which data is represented. The vertical axis represents the categories being compared, while the horizontal axis represents a value. This type of chart provides a visual representation of categorical data.

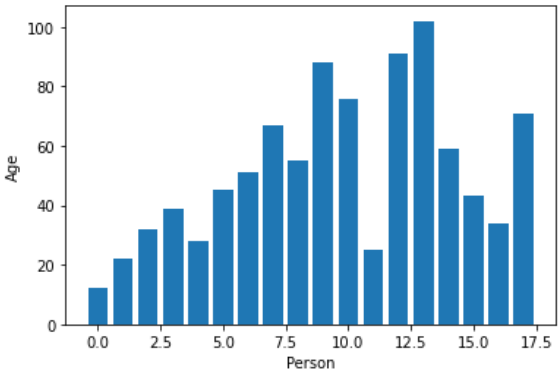
```
In [711]: # Defining the first data set
          x1=[1,3,5,7,9]
          y1=[50,23,50,81,39]

          # Defining the second data set
          x2=[2,4,6,8,10]
          y2=[30,29,67,11,91]

          #Plotting the bar charts with labels and colors
          plt.barh(x1,y1,label='Bar chart 1',color='y')
          plt.barh(x2,y2,label='Bar chart 1',color='r')

          #Defining the x,y labels,title and legend
          plt.xlabel('X Values')
          plt.ylabel('Y Values')
          plt.title('Simple X,Y Plot')
          plt.legend()
          plt.ioff()
```



### 12.3.5 Plotting a bar plot of population data

Here we are plotting the age of people using a bar chart

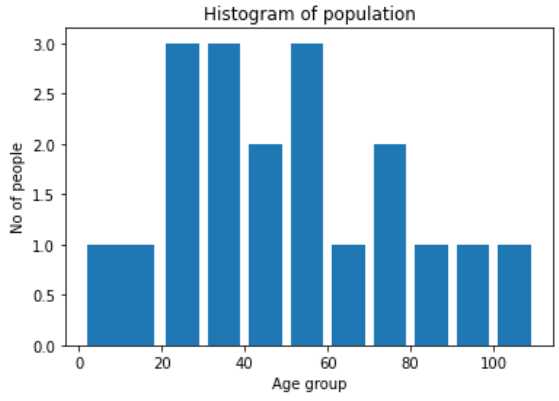
```
In [713]: # Plotting Age distribution using bar chart
          population=[12,22,32,39,28,45,51,67,55,88,76,25,91,102,59,43,34,71]
          per=[x for x in range(len(population))]
          plt.bar(per,population)
          plt.xlabel('Person')
          plt.ylabel('Age')
          plt.show()
          plt.ioff()
```




### 12.3.6 Plotting histogram with same population data

Population data cannot be well represented by bar chart.To represent age data well we have to convert age data into different range like 1-10,11-20 called bins.This is done using histogram

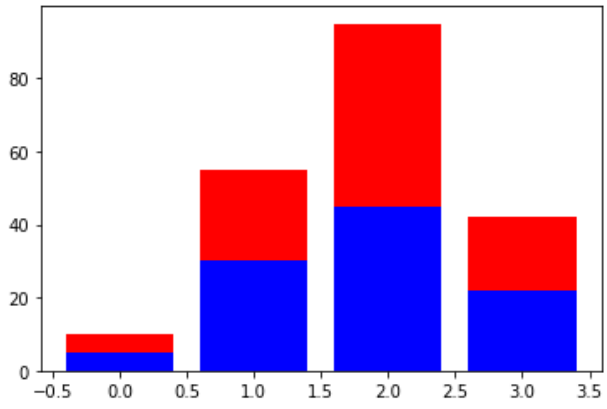
```
In [714]: bins=[0.10,20,30,40,50,60,70,80,90,100,110]
          plt.hist(population,bins,histtype='bar',rwidth=0.8)
          plt.xlabel('Age group')
          plt.ylabel('No of people')
          plt.title('Histogram of population')
          plt.show()
          plt.ioff()
```



### 12.3.7 Stacked bar plot

A stacked bar chart, also known as a stacked bar graph, is a graph that is used to break down and compare parts of a whole. Each bar in the chart represents a whole, and segments in the bar represent different parts or categories of that whole

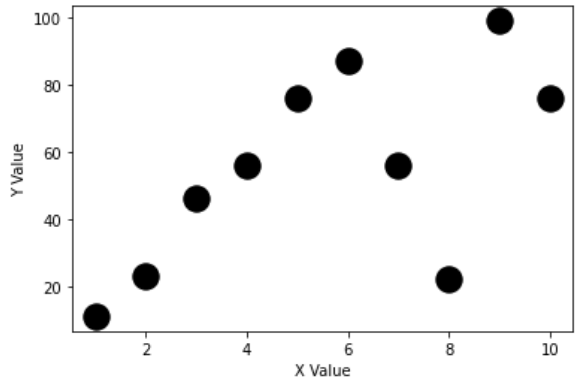
In [715]:
```
import matplotlib.pyplot as plt

Men = [5., 30., 45., 22.]
Women = [5., 25., 50., 20.]

X = range(4)

plt.bar(X, Men, color = 'b')
plt.bar(X, Women, color = 'r',bottom=Men)
plt.show()
plt.ioff()
```




### 12.3.8 Scatter plot

Scatter plots are used when we want to plot relation between two variables.

In [716]:
```
#Data set for scatter plot

x=[1,2,3,4,5,6,7,8,9,10]
y=[11,23,46,56,76,87,56,22,99,76]
plt.scatter(x,y,label='Scatter',color='k',marker='o',s=300) #k represents black color
plt.xlabel('X Value')
plt.ylabel('Y Value')
plt.show()
plt.ioff()
```



## 12.3.9 Plotting pie chart

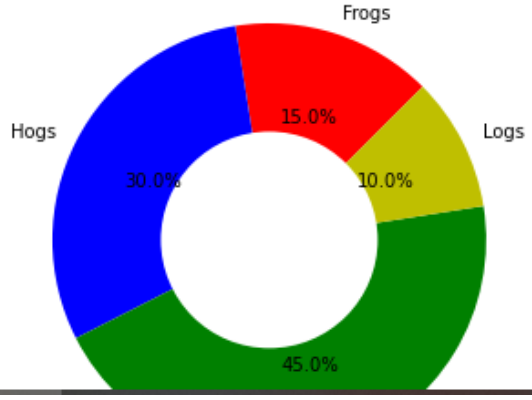
Pie chart is widely used to show the percentage contribution of different entities.The circle represents 100 % and the pie's represent the percentage of each individual quantities.If we slice a pie chart it looks more like a slice from a PIzza.

```
In [717]: hours=[8,2,8,2,4]
          activity=['Sleep','Eat','Work','Play','Others']
          col=['c','b','m','r','y']
          plt.pie(hours,labels=activity,colors=col,shadow=True,startangle=0,explode=(0,0,0,0.2,0),autopct='%1.1f%%'
          plt.title('How time is in a Day')
          plt.ioff()
```


## 12.3.10 Nested pie or doughnut plots

If you are a foodie then better prefer a doughnut plot over a pie plot

```
In [718]: import matplotlib.pyplot as plt
          # Pie chart
          labels = ['Frogs', 'Hogs', 'Dogs', 'Logs']
          sizes = [15, 30, 45, 10]
          #colors
          colors = ['r','b','g','y']

          fig1, ax1 = plt.subplots()
          ax1.pie(sizes, colors = colors, labels=labels, autopct='%1.1f%%', startangle
          #draw circle
          centre_circle = plt.Circle((0,0),.5,fc='white')
          fig = plt.gcf()
          fig.gca().add_artist(centre_circle)
          # Equal aspect ratio ensures that pie is drawn as a circle
          ax1.axis('equal')
          plt.tight_layout()
          plt.show()
          plt.ioff()
```



## 12.4 Tasks:

# 13. Reference

https://docs.python.org/3/tutorial/

https://www.tutorialspoint.com/python/index.htm

https://www.guru99.com/python-tutorials.html

https://www.w3resource.com/python/python-tutorial.php

https://www.w3resource.com/python-exercises/

https://numpy.org/doc/

https://pandas.pydata.org/docs/

https://matplotlib.org/