

# Python

## 1. What is Python?

Python is a popular programming language. It is a widely used general-purpose, high-level programming language. It was created by Guido van Rossum and released in 1991.

### It is used for:

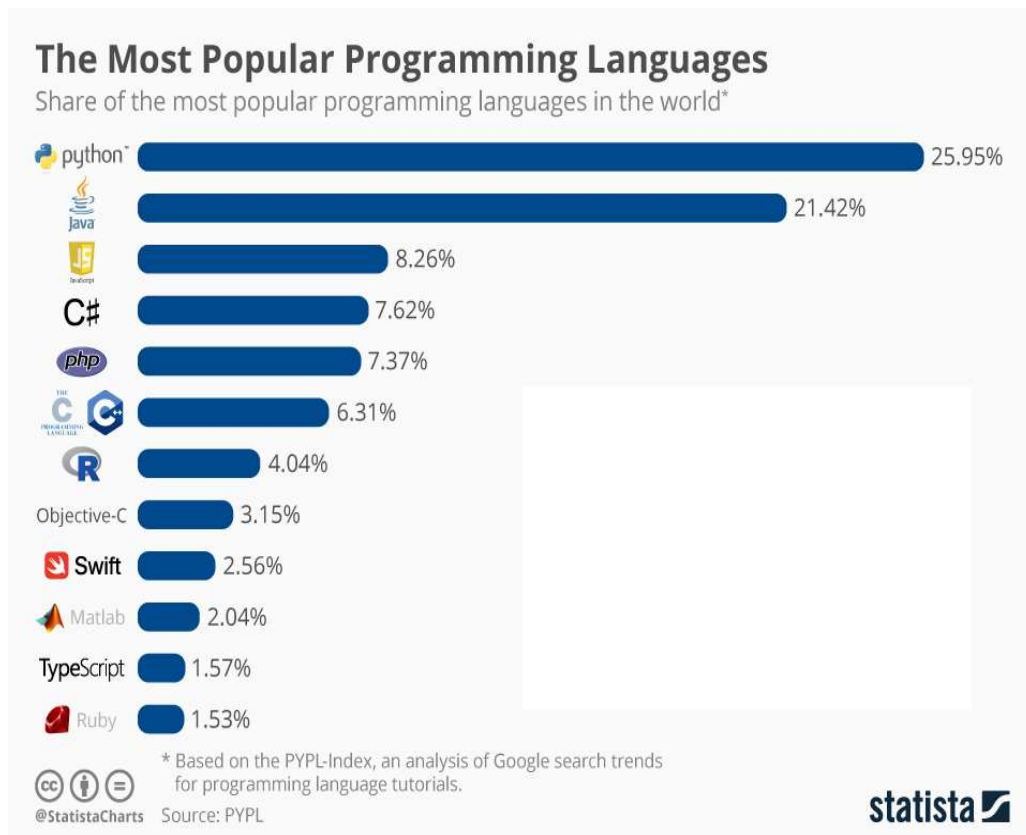
- web development (server-side),
- software development,
- mathematics,
- system scripting.

### Advantages :

- Presence of third-party modules
- Extensive support libraries(NumPy for numerical calculations, Pandas for data analytics etc)
- Open source and community development
- Versatile, Easy to read, learn and write
- User-friendly data structures
- High-level language
- Dynamically typed language(No need to mention data type based on the value assigned, it takes data type)
- Object-oriented language
- Portable and Interactive
- Ideal for prototypes – provide more functionality with less coding
- Highly Efficient
- (IoT)Internet of Things Opportunities
- Interpreted Language
- Portable across Operating systems

### Applications :

- GUI based desktop applications
- Graphic design, image processing applications, Games, and Scientific/ computational Applications
- Web frameworks and applications
- Enterprise and Business applications
- Operating Systems
- Education
- Database Access
- Language Development
- Prototyping
- Software Development



## 2. Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way

### Example 1:

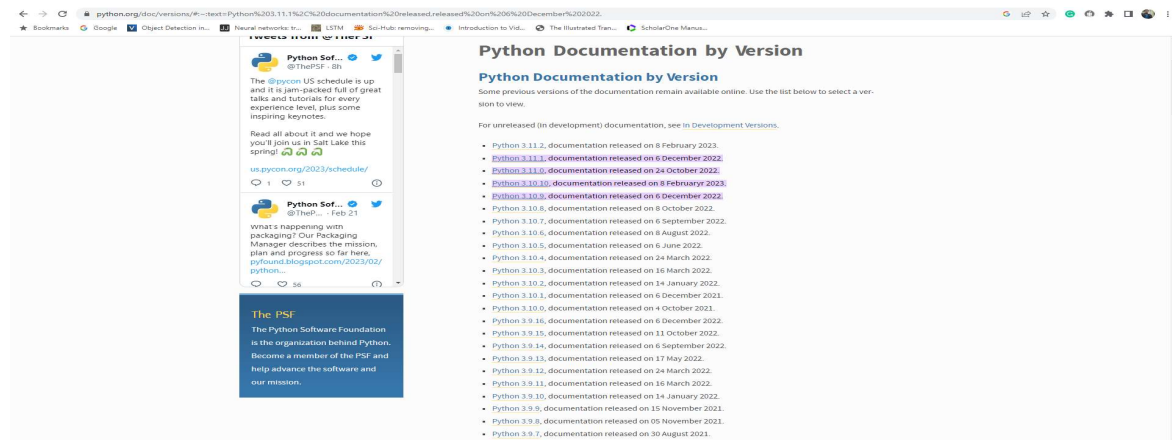
```
print("Hello, World!")
```

Output:

Hello, World!

## 3. Installation of Python

### Step 1: Select Version of Python to Install



### Step 2: Download Python Executable Installer

### Step 3: Run Executable Installer

### Step 4: Verify Python Was Installed On Windows/Linux

## 4. Python IDE:

- An IDE enables programmers to combine the different aspects of writing a computer program.
- IDEs increase programmer productivity by introducing features like editing source code, building executables, and debugging.

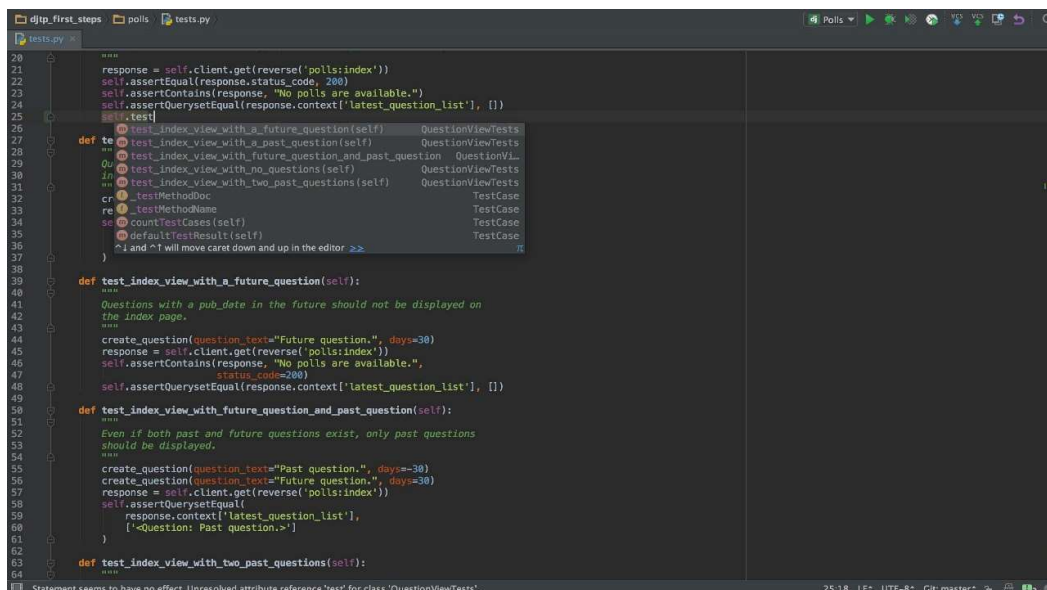
### **IDLE**

- IDLE (Integrated Development and Learning Environment) is a default editor that accompanies Python
- This IDE is suitable for beginner level developers
- The IDLE tool can be used on Mac OS, Windows, and Linux
- Price: Free

```
Python 3.6.4 Shell
Python 3.6.4 (v3.6.4:d48eeced5, Dec 18 2017, 21:07:28)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> WARNING: The version of Tcl/Tk (8.5.9) in use may be unstable.
Visit http://www.python.org/download/mac/tcltk/ for current information.
import pynput
Traceback (most recent call last):
  File "<pyshe11#0>", line 1, in <module>
    import pynput
ModuleNotFoundError: No module named 'pynput'
>>> import sys
>>> sys.version
'3.6.4 (v3.6.4:d48eeced5, Dec 18 2017, 21:07:28) \n[GCC 4.2.1 (Apple Inc. build
5666) (dot 3)]'
>>> sys.path
['', '/Users/jwgsolitude/Documents', '/Library/Frameworks/Python.framework/Versi
ons/3.6/lib/python36.zip', '/Library/Frameworks/Python.framework/Versions/3.6/Li
b/python3.6', '/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/l
ib-dynload', '/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/si
te-packages']
>>> |
```

## 2. PyCharm

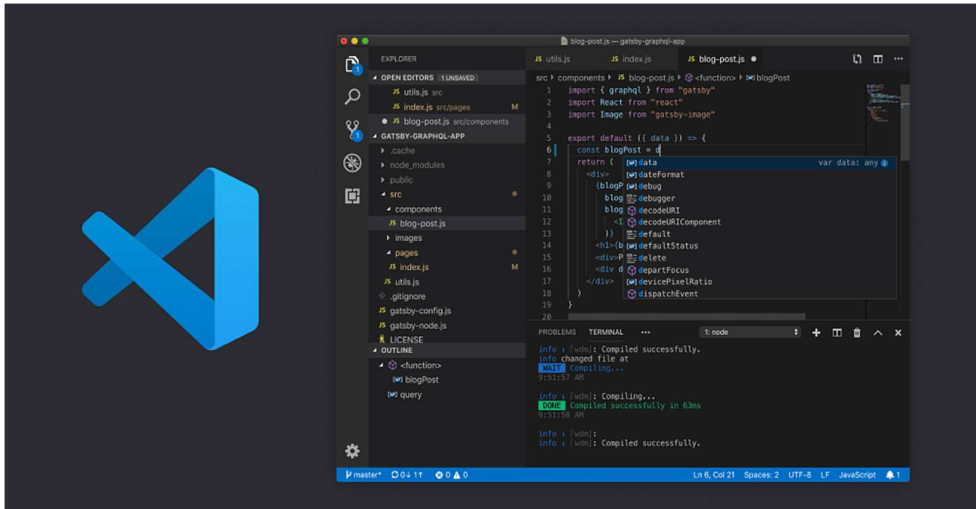
- PyCharm is a widely used Python IDE created by JetBrains
- This IDE is suitable for professional developers and facilitates the development of large Python projects
- Price: Free



```
tests.py
20
21 response = self.client.get(reverse('polls:index'))
22 self.assertEqual(response.status_code, 200)
23 self.assertContains(response, "No polls are available.")
24 self.assertQuerysetEqual(response.context['latest_question_list'], [])
25 self.test
26
27 def test_index_view_with_a_future_question(self):
28     """
29     test_index_view_with_future_question_and_past_question
30     test_index_view_with_no_questions(self)
31     test_index_view_with_two_past_questions(self)
32     """
33     cr = testMethodDoc
34     re = testMethodName
35     se = countTestCases(self)
36     defaultTestResult(self)
37     ^i and ^f will move caret down and up in the editor >>
38
39 def test_index_view_with_a_future_question(self):
40     """
41     Questions with a pub_date in the future should not be displayed on
42     the index page.
43     """
44     create_question(question_text="Future question.", day=30)
45     response = self.client.get(reverse('polls:index'))
46     self.assertContains(response, "No polls are available.",
47                         status_code=200)
48     self.assertQuerysetEqual(response.context['latest_question_list'], [])
49
50 def test_index_view_with_future_question_and_past_question(self):
51     """
52     Even if both past and future questions exist, only past questions
53     should be displayed.
54     """
55     create_question(question_text="Past question.", day=-30)
56     create_question(question_text="Future question.", day=30)
57     response = self.client.get(reverse('polls:index'))
58     self.assertQuerysetEqual(
59         response.context['latest_question_list'],
60         ['<Question: Past question.>']
61     )
62
63 def test_index_view_with_two_past_questions(self):
64     """
```

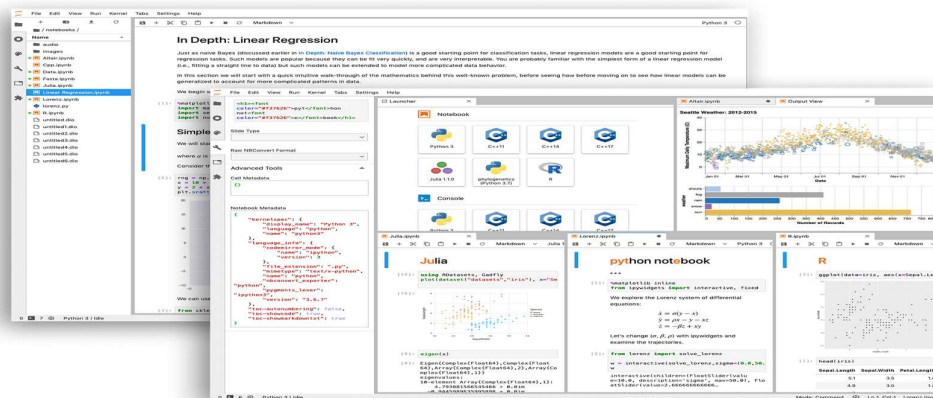
## 3. Visual Studio Code

- Visual Studio Code is an open-source (and free) IDE created by Microsoft. It finds great use for Python development
- VS Code is lightweight and comes with powerful features that only some of the paid IDEs offer
- Price: Free



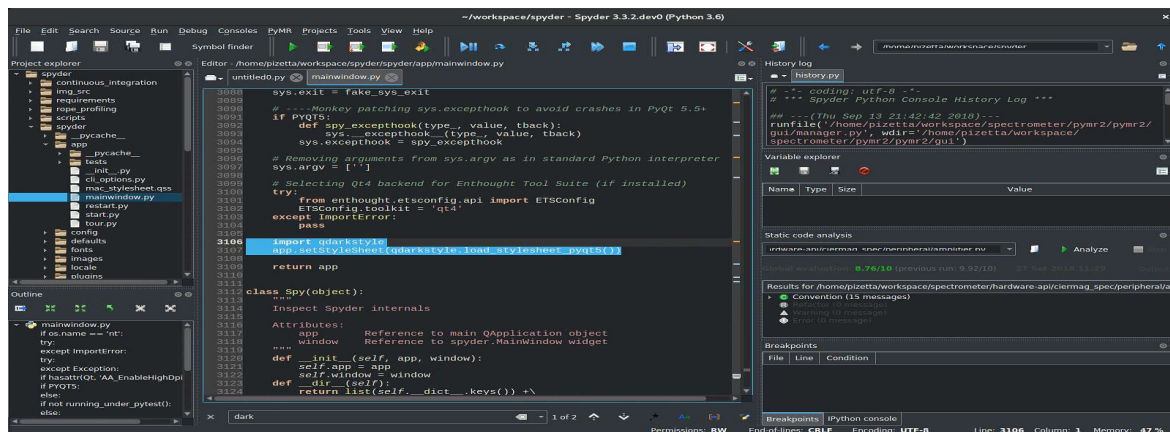
#### 4. Jupyter

- Jupyter is widely used in the field of data science
- It is easy to use, interactive and allows live code sharing and visualization
- Price: Free



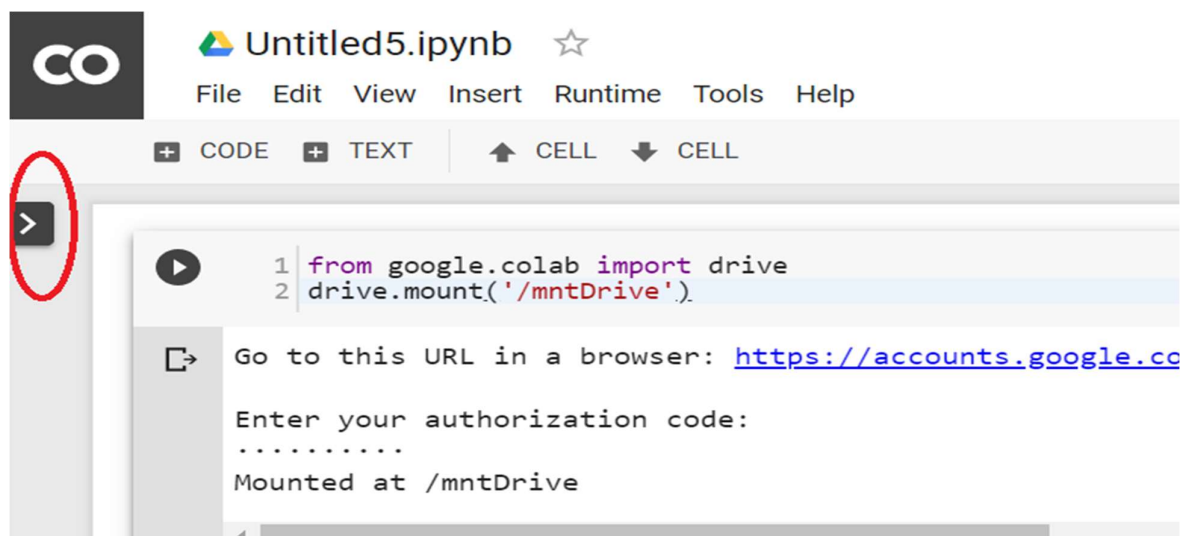
#### 5. Spyder

- Spyder is an open-source IDE most commonly used for scientific development
- Spyder comes with Anaconda distribution, which is popular for data science and machine learning
- Price: Free



## 6. Google Colab:

- Google Colab is a free Jupyter notebook that allows to run Python in the browser without the need for complex configuration.
- It comes with Python installed and has all the main Python libraries installed.
- It also comes integrated with free GPUs.



## 5. The Python Command Line

To test a short amount of code in python sometimes it is quickest and easiest not to write the code in a file. This is made possible because Python can be run as a command line itself.

Type the following on the Windows, Mac or Linux command line:

```
C:\Users\Your Name>python
```

Or, if the "python" command did not work, you can try "py":

```
C:\Users\Your Name>py
```

From there you can write any python, including our hello world example from earlier in the tutorial:

```
C:\Users\Your Name>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
```

Which will write "Hello, World!" in the command line:

```
C:\Users\Your Name>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
Hello, World!
```

Whenever you are done in the python command line, you can simply type the following to quit the python command line interface:

```
exit()
```

## 6. Python Indentation

- Indentation refers to the spaces at the beginning of a code line.
- Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
- Python uses indentation to indicate a block of code.

**Example:**

**No Error**

```
if 5 > 2:
    print("Five is greater than two!")
```

**Error**

```
if 5 > 2:
print("Five is greater than two!")
```

The number of spaces is up to you as a programmer, but it has to be at least one.

```
if 5 > 2:
    print("Five is greater than two!")
```

```
if 5 > 2:  
    print("Five is greater than two!")
```

You have to use the same number of spaces in the same block of code, otherwise Python will give you an error:

```
if 5 > 2:  
    print("Five is greater than two!")  
    print("Five is greater than two!")
```

## 7. Python Variables

In Python, variables are created when you assign a value to it.

Rules for creating variables in Python

- A variable name must start with a letter or the underscore character.
- A variable name cannot start with a number.
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and \_).
- Variable names are case-sensitive (name, Name and NAME are three different variables).
- The reserved words(keywords) cannot be used naming the variable.

### Example:

Variables in Python:

```
x = 5  
y = "Hello, World!"
```

## 8. Comments

- Python has commenting capability for the purpose of in-code documentation.
- Comments start with a #, and Python will render the rest of the line as a comment:

### Examples:

```
#This is a comment.  
print("Hello, World!")
```

Multiple line comment starts from ''' end with '''

```
'''This is comment
```

```
print("hello")'''
```



## 9. Type Casting

Type Casting is the method to convert the variable data type into a certain data type in order to the operation required to be performed by users.

### Example:

```
x = str(3)  # x will be '3'
y = int(3)  # y will be 3
z = float(3) # z will be 3.0
```

You can get the data type of a variable with the `type()` function.

```
x = 5
y = "John"
print(type(x)) # int
print(type(y)) # str
```

### a. Type Casting int to float:

```
# Python program to demonstrate
# type Casting
# int variable
a = 5
# typecast to float
n = float(a)
print(n)
print(type(n))
```

### b. Type Casting float to int:

```
# Python program to demonstrate
# type Casting
# int variable
a = 5.9
# typecast to int
n = int(a)
print(n)
print(type(n))
```

**c. Type casting int to string:**

```
# Python program to demonstrate  
# type Casting  
# int variable  
a = 5  
# typecast to str  
n = str(a)  
print(n)  
print(type(n))
```

**d. Type Casting string to int:**

```
# Python program to demonstrate  
# type Casting  
# string variable  
a = "5"  
# typecast to int  
n = int(a)  
print(n)  
print(type(n))
```

## 10. Python Data Types

### Built-in Data Types

- In programming, data type is an important concept.
- Variables can store data of different types, and different types can do different things.
- Python has the following data types built-in by default, in these categories:

Text Type:                    **str**

Numeric Types:            **int, float, complex**

Sequence Types:      `list, tuple, range`

Mapping Type:        `dict`

Set Types:            `set, frozenset`

Boolean Type:        `bool`

Binary Types:        `bytes, bytearray, memoryview`

### **a. Int**

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

```
x = 1
y = 35656222554887711
z = -3255522
```

```
print(type(x))
print(type(y))
print(type(z))
```

### **b. Float**

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

```
x = 1.10
y = 1.0
z = -35.59
```

```
print(type(x))
print(type(y))
print(type(z))
```

Float can also be scientific numbers with an "e" to indicate the power of 10.

```
x = 35e3
y = 12E4
z = -87.7e100
```

```
print(type(x))
```

```
print(type(y))
print(type(z))
```

### c. Complex

Complex numbers are written with a "j" as the imaginary part:

```
x = 3+5j
y = 5j
z = -5j
```

```
print(type(x))
print(type(y))
print(type(z))
```

### d. Type Conversion

You can convert from one type to another with the `int()`, `float()`, and `complex()` methods:

```
x = 1 # int
y = 2.8 # float
z = 1j # complex
```

```
#convert from int to float:
a = float(x)
```

```
#convert from float to int:
b = int(y)
```

```
#convert from int to complex:
c = complex(x)
```

```
print(a)
print(b)
print(c)
```

```
print(type(a))
print(type(b))
print(type(c))
```

### e. Random Number

Python does not have a `random()` function to make a random number, but Python has a built-in module called `random` that can be used to make random numbers:

```
import random

print(random.randrange(1, 10))
```

## f. Strings

Strings in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

```
print("Hello")    // Hello
print('Hello')    // Hello
```

## Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

```
a = "Hello"
print(a)
```

## Multiline Strings

You can assign a multiline string to a variable by using three quotes:

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

or

```
a = "Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."
print(a)
```

## String Exercise

1. 

```
a = "Hello, World!"
print(a[1])
```
2. 

```
for x in "banana":
    print(x)
```

3. `a = "Hello, World!"`  
`print(len(a))`
4. `txt = "The best things in life are free!"`  
`print("free" in txt)`
5. `txt = "The best things in life are free!"`  
`if "free" in txt:`  
 `print("Yes, 'free' is present.")`
6. `txt = "The best things in life are free!"`  
`print("expensive" not in txt)`
7. `txt = "The best things in life are free!"`  
`if "expensive" not in txt:`  
 `print("No, 'expensive' is NOT present.")`

## Slicing String

You can return a range of characters by using the slice syntax.

Specify the start index and the end index, separated by a colon, to return a part of the string.

1. `b = "Hello, World!"`  
`print(b[2:5])`
2. `b = "Hello, World!"`  
`print(b[:5])`
3. `b = "Hello, World!"`  
`print(b[2:])`
4. `b = "Hello, World!"`  
`print(b[-5:-2])`

## Modify Strings

1. `a = "Hello, World!"`  
`print(a.upper())`
2. `a = "Hello, World!"`  
`print(a.lower())`

3. `a = " Hello, World! "`  
`print(a.strip()) # returns "Hello, World!"`
4. `a = "Hello, World!"`  
`print(a.replace("H", "J"))`
5. `a = "Hello, World!"`  
`print(a.split(",")) # returns ['Hello', ' World!']`

## String Concatenation

1. `a = "Hello"`  
`b = "World"`  
`c = a + b`  
`print(c)`
2. `a = "Hello"`  
`b = "World"`  
`c = a + " " + b`  
`print(c)`

## String Format

Following methods put error.

```
age = 36
txt = "My name is John, I am " + age
print(txt)
```

correct methods

1. `age = 36`  
`txt = "My name is John, and I am {"`  
`print(txt.format(age))`
2. `quantity = 3`  
`itemno = 567`  
`price = 49.95`  
`myorder = "I want {} pieces of item {} for {} dollars."`  
`print(myorder.format(quantity, itemno, price))`
3. `quantity = 3`  
`itemno = 567`  
`price = 49.95`  
`myorder = "I want to pay {2} dollars for {0} pieces of item {1}."`  
`print(myorder.format(quantity, itemno, price))`

## Escape Characters

To insert characters that are illegal in a string, use an escape character.

An escape character is a backslash `\` followed by the character you want to insert.

Error:

```
txt = "We are the so-called "Vikings" from the north."
```

Correct Way:

```
txt = "We are the so-called \"Vikings\" from the north."
```

\'	Single Quote
\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\ooo	Octal value
\xhh	Hex value



## String Methods

Method	Description
<a href="#"><u>capitalize()</u></a>	Converts the first character to upper case
<a href="#"><u>casefold()</u></a>	Converts string into lower case
<a href="#"><u>center()</u></a>	Returns a centered string
<a href="#"><u>count()</u></a>	Returns the number of times a specified value occurs in a string
<a href="#"><u>encode()</u></a>	Returns an encoded version of the string
<a href="#"><u>endswith()</u></a>	Returns true if the string ends with the specified value
<a href="#"><u>expandtabs()</u></a>	Sets the tab size of the string
<a href="#"><u>find()</u></a>	Searches the string for a specified value and returns the position of where it was found
<a href="#"><u>format()</u></a>	Formats specified values in a string
<code>format_map()</code>	Formats specified values in a string

<a href="#"><u>index()</u></a>	Searches the string for a specified value and returns the position of where it was found
<a href="#"><u>isalnum()</u></a>	Returns True if all characters in the string are alphanumeric
<a href="#"><u>isalpha()</u></a>	Returns True if all characters in the string are in the alphabet
<a href="#"><u>isdecimal()</u></a>	Returns True if all characters in the string are decimals
<a href="#"><u>isdigit()</u></a>	Returns True if all characters in the string are digits
<a href="#"><u>isidentifier()</u></a>	Returns True if the string is an identifier
<a href="#"><u>islower()</u></a>	Returns True if all characters in the string are lower case
<a href="#"><u>isnumeric()</u></a>	Returns True if all characters in the string are numeric
<a href="#"><u>isprintable()</u></a>	Returns True if all characters in the string are printable
<a href="#"><u>isspace()</u></a>	Returns True if all characters in the string are whitespaces
<a href="#"><u>istitle()</u></a>	Returns True if the string follows the rules of a title
<a href="#"><u>isupper()</u></a>	Returns True if all characters in the string are upper case

<a href="#"><u>join()</u></a>	Joins the elements of an iterable to the end of the string
<a href="#"><u>ljust()</u></a>	Returns a left justified version of the string
<a href="#"><u>lower()</u></a>	Converts a string into lower case
<a href="#"><u>lstrip()</u></a>	Returns a left trim version of the string
<a href="#"><u>maketrans()</u></a>	Returns a translation table to be used in translations
<a href="#"><u>partition()</u></a>	Returns a tuple where the string is parted into three parts
<a href="#"><u>replace()</u></a>	Returns a string where a specified value is replaced with a specified value
<a href="#"><u>rfind()</u></a>	Searches the string for a specified value and returns the last position of where it was found
<a href="#"><u>rindex()</u></a>	Searches the string for a specified value and returns the last position of where it was found
<a href="#"><u>rjust()</u></a>	Returns a right justified version of the string
<a href="#"><u>rpartition()</u></a>	Returns a tuple where the string is parted into three parts

<a href="#"><u>rsplit()</u></a>	Splits the string at the specified separator, and returns a list
<a href="#"><u>rstrip()</u></a>	Returns a right trim version of the string
<a href="#"><u>split()</u></a>	Splits the string at the specified separator, and returns a list
<a href="#"><u>splitlines()</u></a>	Splits the string at line breaks and returns a list
<a href="#"><u>startswith()</u></a>	Returns true if the string starts with the specified value
<a href="#"><u>strip()</u></a>	Returns a trimmed version of the string
<a href="#"><u>swapcase()</u></a>	Swaps cases, lower case becomes upper case and vice versa
<a href="#"><u>title()</u></a>	Converts the first character of each word to upper case
<a href="#"><u>translate()</u></a>	Returns a translated string
<a href="#"><u>upper()</u></a>	Converts a string into upper case
<a href="#"><u>zfill()</u></a>	Fills the string with a specified number of 0 values at the beginning

## 11. Operators

Operators are used to perform operations on variables and values.

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

### Arithmetic operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

+	Addition	$x + y$
-	Subtraction	$x - y$
*	Multiplication	$x * y$
/	Division	$x / y$
%	Modulus	$x \% y$
**	Exponentiation	$x ** y$
//	Floor division	$x // y$

## Assignment Operators:

Assignment operators are used to assign values to variables.

=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
=	x  = 3	x = x   3
^=	x ^= 3	x = x ^ 3

>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

### Comparison Operators:

Comparison operators are used to compare two values.

==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

### Logical Operators:

Logical operators are used to combine conditional statements:

and	Returns True if both statements are true	<code>x &lt; 5 and x &lt; 10</code>
or	Returns True if one of the statements is true	<code>x &lt; 5 or x &lt; 4</code>
not	Reverse the result, returns False if the result is true	<code>not(x &lt; 5 and x &lt; 10)</code>

### 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:

is	Returns True if both variables are the same object	<code>x is y</code>
is not	Returns True if both variables are not the same object	<code>x is not y</code>

### Membership Operators

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

in	Returns True if a sequence with the specified value is present in the object	<code>x in y</code>
not in	Returns True if a sequence with the specified value is not present in the object	<code>x not in y</code>



## Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

&	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

## 12. Python Collections

There are four collection data types in the Python programming language:

### List

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- Lists are used to store multiple items in a single variable.
- Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.
- List items are indexed, the first item has index [0], the second item has index [1] etc.
- Lists are created using square brackets:

**Example1:**

```
thislist = ["apple", "banana", "cherry"]  
  
print(thislist)
```

**Output:**

```
['apple', 'banana', 'cherry']
```

**Example2:**

```
thislist = ["apple", "banana", "cherry", "apple", "cherry"]  
print(thislist)
```

**Output:**

```
['apple', 'banana', 'cherry', 'apple', 'cherry']
```

**Tuple:**

- Tuple is a collection which is ordered and unchangeable. Allows duplicate members.
- Tuples are used to store multiple items in a single variable.
- Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.
- A tuple is a collection which is ordered and **unchangeable**.
- Tuples are written with round brackets.

**Example1:**

```
thistuple = ("apple", "banana", "cherry")  
print(thistuple)
```

Output:

```
('apple', 'banana', 'cherry')
```

**Example2:**

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")  
print(thistuple)
```

Output:

```
('apple', 'banana', 'cherry', 'apple', 'cherry')
```

## Set:

- Set is a collection which is unordered and unindexed. No duplicate members.
- Sets are used to store multiple items in a single variable.
- Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.
- A set is a collection which is *unordered*, *unchangeable\**, and *unindexed*.
- Sets are written with curly brackets.

### Example 1:

```
thisset = {"apple", "banana", "cherry"}  
print(thisset)
```

### Output:

```
{'banana', 'apple', 'cherry'}
```

### Example 2:

```
thisset = {"apple", "banana", "cherry", "apple"}  
print(thisset)
```

### Output:

```
{'banana', 'apple', 'cherry'}
```

## Dictionary:

- Dictionaries are used to store data values in key:value pairs.
- A dictionary is a collection which is ordered\*, changeable and do not allow duplicates.
- Dictionaries are written with curly brackets, and have keys and values.
- Dictionary is a collection which is ordered\* and changeable. No duplicate members.

### Example 1:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(thisdict)
```

### Output:

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

### Example 2:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964,  
    "year": 2020  
}  
print(thisdict)
```

### Output:

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2020}
```

LIST	SET	TUPLE	DICTIONARY
Mutable	Immutable	Immutable	Mutable
Ordered	Unordered	Ordered	Unordered
Changeable	Unchangeable	Unchangeable	Changeable
Allow Duplicate	No Duplicate	Allow Duplicate	No Duplicate
Square Bracket	Curly Bracket	Round Bracket	Curly Bracket

## Exercises

### Lists

```
mylist = ["apple", "banana", "cherry"]
```

Lists are used to store multiple items in a single variable.

```
thislist = ["apple", "banana", "cherry"]  
print(thislist)
```

It allows duplicate also

```
thislist = ["apple", "banana", "cherry", "apple", "cherry"]  
print(thislist)
```

### Length

```
thislist = ["apple", "banana", "cherry"]  
print(len(thislist))
```

### Data Types

1. list1 = ["apple", "banana", "cherry"]  
list2 = [1, 5, 7, 9, 3]  
list3 = [True, False, False]
2. list1 = ["abc", 34, True, 40, "male"]

### type()

```
mylist = ["apple", "banana", "cherry"]  
print(type(mylist))
```

### list()

```
thislist = list(("apple", "banana", "cherry")) # note the double round-brackets  
print(thislist)
```

### Exercise:

1. thislist = ["apple", "banana", "cherry"]  
print(thislist[1])
2. thislist = ["apple", "banana", "cherry"]  
print(thislist[-1])
3. thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]  
print(thislist[2:5])
4. thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]  
print(thislist[:4])
5. thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]  
print(thislist[2:])
6. thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]  
print(thislist[-4:-1])

```
7. thislist = ["apple", "banana", "cherry"]
   if "apple" in thislist:
       print("Yes, 'apple' is in the fruits list")
```

### Change List Items

```
1. thislist = ["apple", "banana", "cherry"]
   thislist[1] = "blackcurrant"
   print(thislist)
2. thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
   thislist[1:3] = ["blackcurrant", "watermelon"]
   print(thislist)
3. thislist = ["apple", "banana", "cherry"]
   thislist[1:2] = ["blackcurrant", "watermelon"]
   print(thislist)
4. thislist = ["apple", "banana", "cherry"]
   thislist[1:3] = ["watermelon"]
   print(thislist)
5. thislist = ["apple", "banana", "cherry"]
   thislist.insert(2, "watermelon")
   print(thislist)
```

### Add List Items

```
1. thislist = ["apple", "banana", "cherry"]
   thislist.append("orange")
   print(thislist)
2. thislist = ["apple", "banana", "cherry"]
   thislist.insert(1, "orange")
   print(thislist)
3. thislist = ["apple", "banana", "cherry"]
   tropical = ["mango", "pineapple", "papaya"]
   thislist.extend(tropical)
   print(thislist)
4. thislist = ["apple", "banana", "cherry"]
   thistuple = ("kiwi", "orange")
   thislist.extend(thistuple)
   print(thislist)
```

### Remove List Items

```
1. thislist = ["apple", "banana", "cherry"]
   thislist.remove("banana")
   print(thislist)
2. thislist = ["apple", "banana", "cherry"]
   thislist.pop(1)
   print(thislist)
```

3. `thislist = ["apple", "banana", "cherry"]`  
`thislist.pop()`  
`print(thislist)`
4. `thislist = ["apple", "banana", "cherry"]`  
`del thislist[0]`  
`print(thislist)`
5. `thislist = ["apple", "banana", "cherry"]`  
`del thislist`
6. `thislist = ["apple", "banana", "cherry"]`  
`thislist.clear()`  
`print(thislist)`

### Loop Lists

1. `thislist = ["apple", "banana", "cherry"]`  
`for x in thislist:`  
`print(x)`
2. `thislist = ["apple", "banana", "cherry"]`  
`for i in range(len(thislist)):`  
`print(thislist[i])`
3. `thislist = ["apple", "banana", "cherry"]`  
`i = 0`  
`while i < len(thislist):`  
`print(thislist[i])`  
`i = i + 1`
4. `thislist = ["apple", "banana", "cherry"]`  
`[print(x) for x in thislist]`

### Sort Lists

1. `thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]`  
`thislist.sort()`  
`print(thislist)`
2. `thislist = [100, 50, 65, 82, 23]`  
`thislist.sort()`  
`print(thislist)`
3. `thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]`  
`thislist.sort(reverse = True)`  
`print(thislist)`
4. `thislist = [100, 50, 65, 82, 23]`  
`thislist.sort(reverse = True)`  
`print(thislist)`
5. `def myfunc(n):`  
`return abs(n - 50)`  
  
`thislist = [100, 50, 65, 82, 23]`

```

        thislist.sort(key = myfunc)
    print(thislist)
6.  thislist = ["banana", "Orange", "Kiwi", "cherry"]
    thislist.sort()
    print(thislist)
7.  thislist = ["banana", "Orange", "Kiwi", "cherry"]
    thislist.sort(key = str.lower)
    print(thislist)
8.  thislist = ["banana", "Orange", "Kiwi", "cherry"]
    thislist.reverse()
    print(thislist)

```

## Copy Lists

You cannot copy a list simply by typing `list2 = list1`, because: `list2` will only be a *reference* to `list1`, and changes made in `list1` will automatically also be made in `list2`.

```

1.  thislist = ["apple", "banana", "cherry"]
    mylist = thislist.copy()
    print(mylist)
2.  thislist = ["apple", "banana", "cherry"]
    mylist = list(thislist)
    print(mylist)

```

## Join Lists

```

1.  list1 = ["a", "b", "c"]
    list2 = [1, 2, 3]

    list3 = list1 + list2
    print(list3)
2.  list1 = ["a", "b", "c"]
    list2 = [1, 2, 3]

    for x in list2:
        list1.append(x)

    print(list1)
3.  list1 = ["a", "b", "c"]
    list2 = [1, 2, 3]

    list1.extend(list2)
    print(list1)

```



<a href="#"><u>append()</u></a>	Adds an element at the end of the list
<a href="#"><u>clear()</u></a>	Removes all the elements from the list
<a href="#"><u>copy()</u></a>	Returns a copy of the list
<a href="#"><u>count()</u></a>	Returns the number of elements with the specified value
<a href="#"><u>extend()</u></a>	Add the elements of a list (or any iterable), to the end of the current list
<a href="#"><u>index()</u></a>	Returns the index of the first element with the specified value
<a href="#"><u>insert()</u></a>	Adds an element at the specified position
<a href="#"><u>pop()</u></a>	Removes the element at the specified position
<a href="#"><u>remove()</u></a>	Removes the item with the specified value
<a href="#"><u>reverse()</u></a>	Reverses the order of the list
<a href="#"><u>sort()</u></a>	Sorts the list

# Tuples

```
mytuple = ("apple", "banana", "cherry")
```

Tuples are used to store multiple items in a single variable.

Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.

A tuple is a collection which is ordered and **unchangeable**.

Tuples are written with round brackets.

```
1. thistuple = ("apple", "banana", "cherry")
   print(thistuple)
```

It also allows duplicate

```
1. thistuple = ("apple", "banana", "cherry")
   print(len(thistuple))
```

```
2. thistuple = ("apple", "banana", "cherry")
   print(len(thistuple))
```

```
3. thistuple = ("apple",)
   print(type(thistuple))
```

```
#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

```
4. tuple1 = ("apple", "banana", "cherry")
   tuple2 = (1, 5, 7, 9, 3)
   tuple3 = (True, False, False)
```

```
5. tuple1 = ("abc", 34, True, 40, "male")
```

```
6. mytuple = ("apple", "banana", "cherry")
   print(type(mytuple))
```

```
7. thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets
   print(thistuple)
```

```
8. thistuple = ("apple", "banana", "cherry")
   print(thistuple[1])
```

9. `thistuple = ("apple", "banana", "cherry")`  
`print(thistuple[-1])`
10. `thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")`  
`print(thistuple[2:5])`
11. `thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")`  
`print(thistuple[:4])`
12. `thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")`  
`print(thistuple[2:])`

## Update Tuples

1. `x = ("apple", "banana", "cherry")`  
`y = list(x)`  
`y[1] = "kiwi"`  
`x = tuple(y)`  
  
`print(x)`
2. `thistuple = ("apple", "banana", "cherry")`  
`y = list(thistuple)`  
`y.append("orange")`  
`thistuple = tuple(y)`
3. `thistuple = ("apple", "banana", "cherry")`  
`y = ("orange",)`  
`thistuple += y`  
  
`print(thistuple)`
4. `thistuple = ("apple", "banana", "cherry")`  
`y = list(thistuple)`  
`y.remove("apple")`  
`thistuple = tuple(y)`
5. `thistuple = ("apple", "banana", "cherry")`  
`del thistuple`  
`print(thistuple)` *#this will raise an error because the tuple no longer exists*

## Unpack Tuples

```
fruits = ("apple", "banana", "cherry")
```

## Unpack Tuple

1. `fruits = ("apple", "banana", "cherry")`

```
(green, yellow, red) = fruits
```

```
print(green)
print(yellow)
print(red)
```

## Using Astrick \*

2. `fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")`

```
(green, yellow, *red) = fruits
```

```
print(green)
print(yellow)
print(red)
```

3. `fruits = ("apple", "mango", "papaya", "pineapple", "cherry")`

```
(green, *tropic, red) = fruits
```

```
print(green)
print(tropic)
print(red)
```

## Loop Tuples

1. `thistuple = ("apple", "banana", "cherry")`  
`for x in thistuple:`  
 `print(x)`
2. `thistuple = ("apple", "banana", "cherry")`  
`for i in range(len(thistuple)):`  
 `print(thistuple[i])`
3. `thistuple = ("apple", "banana", "cherry")`  
`i = 0`  
`while i < len(thistuple):`  
 `print(thistuple[i])`  
 `i = i + 1`

## Join Tuples

1. `tuple1 = ("a", "b", "c")`  
`tuple2 = (1, 2, 3)`

```
tuple3 = tuple1 + tuple2
print(tuple3)
2. fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2

print(mytuple)
```

## Tuple Methods

### [count\(\)](#)

Returns the number of times a specified value occurs in a tuple

### [index\(\)](#)

Searches the tuple for a specified value and returns the position of where it was found

```
1. thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
```

```
x = thistuple.count(5)
```

```
print(x)
```

```
2. thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
```

```
x = thistuple.index(8)
```

```
print(x)
```

## Sets

```
myset = {"apple", "banana", "cherry"}
```

Sets are used to store multiple items in a single variable.

Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are [List](#), [Tuple](#), and [Dictionary](#), all with different qualities and usage.

A set is a collection which is both *unordered* and *unindexed*.

Sets are written with curly brackets.

```
thisset = {"apple", "banana", "cherry"}
```

```
print(thisset)
```

Here duplicate value will be ignored:

```
thisset = {"apple", "banana", "cherry", "apple"}
```

```
print(thisset)
```

1. `thisset = {"apple", "banana", "cherry"}`  
  
`print(len(thisset))`
2. `set1 = {"apple", "banana", "cherry"}`  
`set2 = {1, 5, 7, 9, 3}`  
`set3 = {True, False, False}`
3. `set1 = {"abc", 34, True, 40, "male"}`
4. `myset = {"apple", "banana", "cherry"}`  
`print(type(myset))`
5. `thisset = set(("apple", "banana", "cherry"))` # note the double round-brackets  
`print(thisset)`
6. `thisset = {"apple", "banana", "cherry"}`  
  
`for x in thisset:`  
 `print(x)`
7. `thisset = {"apple", "banana", "cherry"}`  
  
`print("banana" in thisset)`

## Change Items

Once a set is created, you cannot change its items, but you can add new items.

## Add Set Items

1. `thisset = {"apple", "banana", "cherry"}`  
  
`thisset.add("orange")`  
  
`print(thisset)`
2. `thisset = {"apple", "banana", "cherry"}`  
`tropical = {"pineapple", "mango", "papaya"}`  
  
`thisset.update(tropical)`  
  
`print(thisset)`
3. `thisset = {"apple", "banana", "cherry"}`  
`mylist = ["kiwi", "orange"]`  
  
`thisset.update(mylist)`

```

print(thisset)
4. thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")

print(thisset)
5. thisset = {"apple", "banana", "cherry"}

thisset.discard("banana")

print(thisset)
6. thisset = {"apple", "banana", "cherry"}

x = thisset.pop()

print(x)

print(thisset)
7. thisset = {"apple", "banana", "cherry"}

thisset.clear()

print(thisset)
8. thisset = {"apple", "banana", "cherry"}

del thisset

print(thisset)

```

### Loop Sets

```

thisset = {"apple", "banana", "cherry"}

for x in thisset:
    print(x)

```

### Join Sets

```

1. set1 = {"a", "b", "c"}
   set2 = {1, 2, 3}

   set3 = set1.union(set2)
   print(set3)

2. set1 = {"a", "b", "c"}
   set2 = {1, 2, 3}

```

```
set1.update(set2)
print(set1)
```

```
3. x = {"apple", "banana", "cherry"}
   y = {"google", "microsoft", "apple"}
```

```
x.intersection_update(y)

print(x)
```

```
4. x = {"apple", "banana", "cherry"}
   y = {"google", "microsoft", "apple"}
```

```
z = x.intersection(y)

print(z)
```

```
5. x = {"apple", "banana", "cherry"}
   y = {"google", "microsoft", "apple"}
```

```
x.symmetric_difference_update(y)

print(x)
```

```
6. x = {"apple", "banana", "cherry"}
   y = {"google", "microsoft", "apple"}
```

```
z = x.symmetric_difference(y)

print(z)
```

## Set Methods

[add\(\)](#)

Adds an element to the set

[clear\(\)](#)

Removes all the elements from the set

[copy\(\)](#)

Returns a copy of the set



<a href="#"><u>difference()</u></a>	Returns a set containing the difference between two or more sets
<a href="#"><u>difference_update()</u></a>	Removes the items in this set that are also included in another, specified set
<a href="#"><u>discard()</u></a>	Remove the specified item
<a href="#"><u>intersection()</u></a>	Returns a set, that is the intersection of two other sets
<a href="#"><u>intersection_update()</u></a>	Removes the items in this set that are not present in other, specified set(s)
<a href="#"><u>isdisjoint()</u></a>	Returns whether two sets have a intersection or not
<a href="#"><u>issubset()</u></a>	Returns whether another set contains this set or not
<a href="#"><u>issuperset()</u></a>	Returns whether this set contains another set or not
<a href="#"><u>pop()</u></a>	Removes an element from the set
<a href="#"><u>remove()</u></a>	Removes the specified element
<a href="#"><u>symmetric_difference()</u></a>	Returns a set with the symmetric differences of two sets

[symmetric\\_difference\\_update\(\)](#) inserts the symmetric differences from this set and another

[union\(\)](#) Return a set containing the union of sets

[update\(\)](#) Update the set with the union of this set and others

## Dictionaries

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}
```

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered\*, changeable and does not allow duplicates.

As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

Dictionaries are written with curly brackets, and have keys and values:

1. 

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(thisdict)
```
2. 

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(thisdict["brand"])
```

Here duplicates are not allowed:

1. 

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964,  
    "year": 2020  
}  
print(thisdict)
```
2. 

```
print(len(thisdict))
```
3. 

```
thisdict = {  
    "brand": "Ford",  
    "electric": False,  
    "year": 1964,  
    "colors": ["red", "white", "blue"]  
}
```
4. 

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(type(thisdict))
```

### Access Dictionary Items

1. 

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
x = thisdict["model"]
```
2. 

```
x = thisdict.get("model")
```
3. 

```
x = thisdict.keys()
```
4. 

```
car = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
x = car.keys()  
print(x) #before the change  
car["color"] = "white"  
  
print(x) #after the change
```

5. `x = thisdict.values()`

```
6. car = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}
```

`x = car.values()`

`print(x)` #before the change

`car["year"] = 2020`

`print(x)` #after the change

```
7. car = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}
```

`x = car.values()`

`print(x)` #before the change

`car["color"] = "red"`

`print(x)` #after the change

8. `x = thisdict.items()`

```
car = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}
```

`x = car.items()`

`print(x)` #before the change

`car["year"] = 2020`

`print(x)` #after the change

```

9. car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}

x = car.items()

print(x) #before the change

car["color"] = "red"

print(x) #after the change
10. thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
if "model" in thisdict:
    print("Yes, 'model' is one of the keys in the thisdict dictionary")

```

### Change Dictionary Items

```

1. thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
thisdict["year"] = 2018
2. thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
thisdict.update({"year": 2020})

```

### Add Dictionary Items

```

1. thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
thisdict["color"] = "red"
print(thisdict)

```

```
2. thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.update({"color": "red"})
```

## Remove Dictionary Items

1. 

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.pop("model")  
print(thisdict)
```
2. 

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.popitem()  
print(thisdict)
```
3. 

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
del thisdict["model"]  
print(thisdict)
```
4. 

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
del thisdict  
print(thisdict) #this will cause an error because "thisdict" no longer exists.
```
5. 

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}
```

```
thisdict.clear()
print(thisdict)
```

## Loop Dictionaries

1. 

```
for x in thisdict:
    print(x)
```
2. 

```
for x in thisdict:
    print(thisdict[x])
```
3. 

```
for x in thisdict.values():
    print(x)
```
4. 

```
for x in thisdict.keys():
    print(x)
```
5. 

```
for x, y in thisdict.items():
    print(x, y)
```

## Copy Dictionaries

1. 

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```
2. 

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
mydict = dict(thisdict)
print(mydict)
```

## Nested Dictionaries

1. 

```
myfamily = {
    "child1" : {
        "name" : "Emil",
        "year" : 2004
    },
    "child2" : {
        "name" : "Tobias",
        "year" : 2007
    },
    "child3" : {
        "name" : "Linus",
        "year" : 2011
    }
}
```

```

    }
}
2. child1 = {
    "name" : "Emil",
    "year" : 2004
}
child2 = {
    "name" : "Tobias",
    "year" : 2007
}
child3 = {
    "name" : "Linus",
    "year" : 2011
}

myfamily = {
    "child1" : child1,
    "child2" : child2,
    "child3" : child3
}

```

## Dictionary Methods

<a href="#"><u>clear()</u></a>	Removes all the elements from the dictionary
<a href="#"><u>copy()</u></a>	Returns a copy of the dictionary
<a href="#"><u>fromkeys()</u></a>	Returns a dictionary with the specified keys and value
<a href="#"><u>get()</u></a>	Returns the value of the specified key
<a href="#"><u>items()</u></a>	Returns a list containing a tuple for each key value pair
<a href="#"><u>keys()</u></a>	Returns a list containing the dictionary's keys



<a href="#"><code>pop()</code></a>	Removes the element with the specified key
<a href="#"><code>popitem()</code></a>	Removes the last inserted key-value pair
<a href="#"><code>setdefault()</code></a>	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
<a href="#"><code>update()</code></a>	Updates the dictionary with the specified key-value pairs
<a href="#"><code>values()</code></a>	Returns a list of all the values in the dictionary

## 13. Python Conditions and If statements/ Decision making

Python supports the usual logical conditions from mathematics:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

These conditions can be used in several ways, most commonly in "if statements" and loops.

An "if statement" is written by using the `if` keyword.

### Example:

```
a = 33
b = 200
if b > a:
    print("b is greater than a")
```

Output:

b is greater than a

### Example:

```
a = 200
b = 33
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
else:
    print("a is greater than b")
```

### Example:

```
x = 41
if x > 10:
    print("Above ten,")
    if x > 20:
        print("and also above 20!")
    else:
        print("but not above 20.")
```

### LOOP:

Python has two primitive loop commands:

- while loops
- for loops

#### a. While Loop

With the while loop we can execute a set of statements as long as a condition is true.

#### Example

```
i = 1
while i < 6:
    print(i)
    i += 1
```

output

1  
2  
3  
4  
5

## Exercise

1. 

```
i = 1
while i < 6:
    print(i)
    i += 1
```
2. 

```
i = 1
while i < 6:
    print(i)
    if i == 3:
        break
    i += 1
```
3. 

```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```
4. 

```
i = 1
while i < 6:
    print(i)
    i += 1
else:
    print("i is no longer less than 6")
```

## b. For Loops:

A **for** loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the **for** keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the **for** loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Example:

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

Output: