



Amarjeet Mohanty

-Department of CSE

What is Python?

Python is a very popular general-purpose interpreted, interactive, object-oriented, and high-level programming language. Python is dynamically-typed and garbage-collected programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL).

Python is very high in demand and all the major companies are looking for great Python Programmers to develop websites, software components, and applications or to work with Data Science, AI, and ML technologies.

Too name a few companies, that use Python extensively are:

- Google
- Intel
- NASA
- PayPal
- Facebook
- IBM
- Amazon
- Netflix
- Pinterest
- Uber
- Many more...

Key features of Python

[Python](#) has many reasons for being popular and in demand. A few of the reasons are mentioned below.

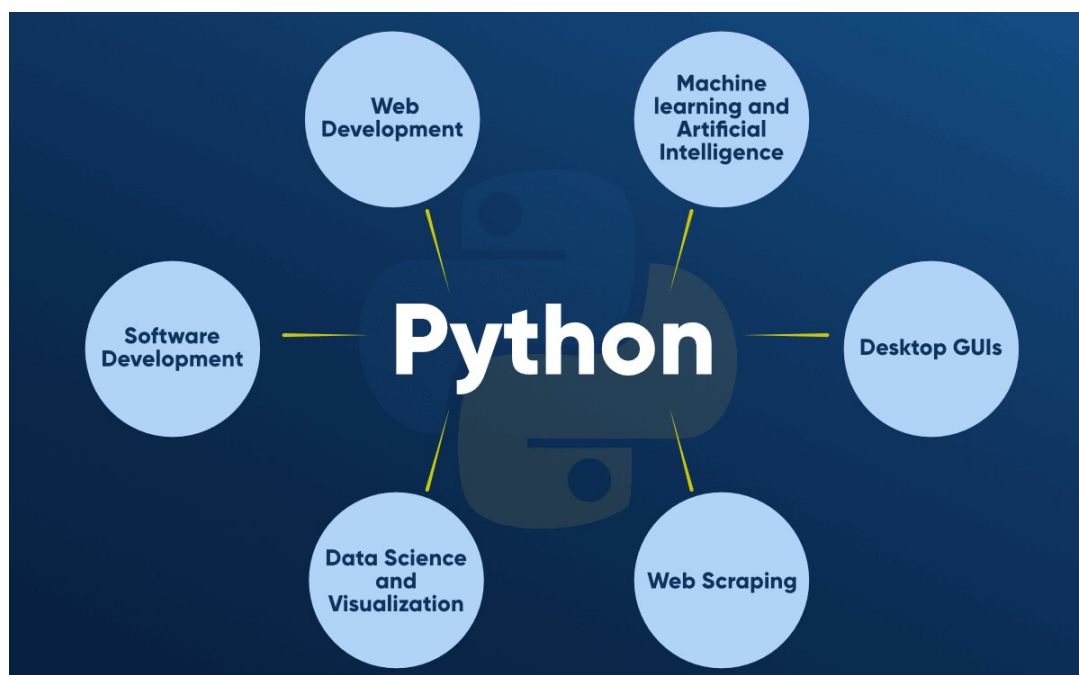
- Emphasis on **code readability, shorter codes, ease of writing**.
- Programmers can express logical concepts in **fewer lines of code** in comparison to languages such as C++ or Java.

- Python **supports multiple programming paradigms**, like object-oriented, imperative and functional programming or procedural.
- It provides **extensive support libraries** (Django for web development, Pandas for data analytics etc)
- **Dynamically typed language**(Data type is based on value assigned)
- Philosophy is “Simplicity is the best”.

Applications of Python

The latest release of Python is 3.x. As mentioned before, Python is one of the most widely used language over the web. I'm going to list few of them here:

- **Easy-to-learn** – Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** – Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** – Python's source code is fairly easy-to-maintain.
- **A broad standard library** – Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** – Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable** – Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable** – You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases** – Python provides interfaces to all major commercial databases.
- **GUI Programming** – Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable** – Python provides a better structure and support for large programs than shell scripting.



Python - Environment Setup

Local Environment Setup

Open a terminal window and type "python" to find out if it is already installed and which version is installed. If Python is already installed then you will get a message something like as follows:

```
$ python
Python 3.6.8 (default, Sep 10 2021, 09:13:53)
[GCC 8.5.0 20210514 (Red Hat 8.5.0-3)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Getting Python

The most up-to-date and current source code, binaries, documentation, news, etc., is available on the official website of Python <https://www.python.org/>

Installing Python

Python distribution is available for a wide variety of platforms. You need to download only the binary code (MSI file) applicable for your platform and install Python.

Setting path at Windows

To add the Python directory to the path for a particular session in Windows –

At the command prompt – type path %path%;C:\Python and press Enter.

Note – C:\Python is the path of the Python directory

Running Python

There are three different ways to start Python –

Interactive Interpreter

You can start Python from Unix, DOS, or any other system that provides you a command-line interpreter or shell window.

Enter **python** the command line.

Start coding right away in the interactive interpreter.

```
$python # Unix/Linux
or
python% # Unix/Linux
or
C:> python # Windows/DOS
```

Script from the Command-line

A Python script can be executed at command line by invoking the interpreter on your application, as in the following –

```
$python script.py # Unix/Linux
```

or

```
python% script.py # Unix/Linux
```

or

```
C: >python script.py # Windows/DOS
```

Python - Basic Syntax

The Python syntax defines a set of rules that are used to create Python statements while writing a Python Program. The Python Programming Language Syntax has many similarities to Perl, C, and Java Programming Languages. However, there are some definite differences between the languages.

First Python Program

Let us execute a Python "Hello, World!" Programs in different modes of programming.

Python - Interactive Mode Programming

We can invoke a Python interpreter from command line by typing **python** at the command prompt as following –

```
$ python
Python 3.6.8 (default, Sep 10 2021, 09:13:53)
[GCC 8.5.0 20210514 (Red Hat 8.5.0-3)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Here >>> denotes a Python Command Prompt where you can type your commands. Let's type the following text at the Python prompt and press the Enter –

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

If you are running older version of Python, like Python 2.4.x, then you would need to use print statement without parenthesis as in **print "Hello, World!"**. However in Python version 3.x, this produces the following result –

```
Hello, World!
```

Python - Script Mode Programming

We can invoke the Python interpreter with a script parameter which begins the execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script which is simple text file. Python files have extension **.py**. Type the following source code in a **test.py** file –

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

We assume that you have Python interpreter path set in PATH variable. Now, let's try to run this program as follows –

```
$ python test.py
```

This produces the following result –

```
Hello, World!
```

Let us try another way to execute a Python script. Here is the modified test.py file –

```
#!/usr/bin/python  
  
print ("Hello, World!")
```

We assume that you have Python interpreter available in /usr/bin directory. Now, try to run this program as follows –

```
$ chmod +x test.py      # This is to make file executable  
$ ./test.py
```

This produces the following result –

```
Hello, World!
```

Python Syntax

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

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

Python will give you an error if you skip the indentation:

Example

Syntax Error:

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

Python Comments

Comments can be used to explain Python code.

Comments can be used to make the code more readable.

Comments can be used to prevent execution when testing code.

Creating a Comment

Comments starts with a #, and Python will ignore them:

Example

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

Comments can be placed at the end of a line, and Python will ignore the rest of the line:

Example

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

A comment does not have to be text that explains the code, it can also be used to prevent Python from executing code:

Example

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

Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it:

Example

```
"""  
This is a comment  
written in  
more than just one line  
"""  
print("Hello, World!")
```

As long as the string is not assigned to a variable, Python will read the code, but then ignore it, and you have made a multiline comment.

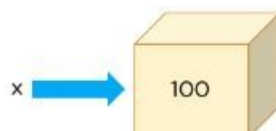
Python Variables

A variable is a fundamental concept in any programming language. It is a reserved memory location that stores and manipulates data.

Variables are entities of a program that holds a value. Here is an example of a variable:

```
x=100
```

In the below diagram, the box holds a value of **100** and is named as **x**. Therefore, the variable is x, and the data it holds is the value.



The **data type** for a variable is the type of data it holds.

In the above example, x is holding 100, which is a number, and the data type of x is a number.

In Python, there are three types of numbers: **Integer**, **Float**, and **Complex**.

Integers are numbers without decimal points. Floats are numbers with decimal points. Complex numbers have real parts and imaginary parts.

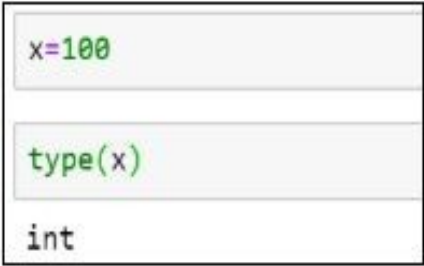
Another data type that is very different from a number is called a **string**, which is a collection of characters.

Let's see a variable with an integer data type:

```
x=100
```

To check the data type of x, use the **type()** function:

```
type(x)
```



```
x=100
type(x)
int
```

Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

Example

```
x = 5
y = "John"
print(x)
print(y)
```

Variables do not need to be declared with any particular *type*, and can even change type after they have been set.

Example

```
X = 4      # x is of type int
x = "Sally" # x is now of type str
print(x)
```

It will print Sally

Casting

If you want to specify the data type of a variable, this can be done with casting.

Example

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

Get the Type

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

Example

```
x = 5  
y = "John"  
print(type(x))  
print(type(y))
```

Single or Double Quotes?

String variables can be declared either by using single or double quotes:

Example

```
x = "John"  
# is the same as  
x = 'John'
```

Case-Sensitive

Variable names are case-sensitive.

Example

This will create two variables:

```
a = 4  
A = "Sally"  
#A will not overwrite a
```

Python-Variable Names

Variable Names

A variable can have a short name (like `x` and `y`) or a more descriptive name (`age`, `carname`, `total_volume`). Rules for Python variables:

- 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 (age, Age and AGE are three different variables)

Example

Legal variable names:

```
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"
```

Remember that variable names are case-sensitive

Many Values to Multiple Variables

Python allows you to assign values to multiple variables in one line:

Example

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

Note: Make sure the number of variables matches the number of values, or else you will get an error.

One Value to Multiple Variables

And you can assign the *same* value to multiple variables in one line:

Example

```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

Unpack a Collection

If you have a collection of values in a list, tuple etc. Python allows you to extract the values into variables. This is called *unpacking*.

Example

Unpack a list:

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
print(x)
print(y)
print(z)
```

Python - Output Variables

Output Variables

The Python `print()` function is often used to output variables.

Example

```
x = "Python is awesome"
print(x)
```

In the `print()` function, you can output multiple variables, separated by a comma:

Example

```
x = "Python"
y = "is"
z = "awesome"
print(x, y, z)
```

You can also use the `+` operator to output multiple variables:

Example

```
x = "Python "
y = "is "
z = "awesome"
print(x + y + z)
```

For numbers, the `+` character works as a mathematical operator:

Example

```
x = 5
y = 10
print(x + y)
```

In the `print()` function, when you try to combine a string and a number with the `+` operator, Python will give you an error:

Example

```
x = 5
y = "John"
print(x + y)                **ERROR
```

The best way to output multiple variables in the `print()` function is to separate them with commas, which even support different data types:

Example

```
x = 5
y = "John"
print(x, y)
```

Python - Global Variables

Global Variables

Variables that are created outside of a function (as in all of the examples above) are known as global variables.

Global variables can be used by everyone, both inside of functions and outside.

Example

Create a variable outside of a function, and use it inside the function

```
x = "awesome"

def myfunc():
    print("Python is " + x)

myfunc()
```

op: Python is awesome

If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

Example

Create a variable inside a function, with the same name as the global variable

```
x = "awesome"

def myfunc():
    x = "fantastic"
    print("Python is " + x)

myfunc()

print("Python is " + x)
```

op: Python is fantastic
Python is awesome

The global Keyword

Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.

To create a global variable inside a function, you can use the `global` keyword.

Example

If you use the `global` keyword, the variable belongs to the global scope:

```
def myfunc():  
    global x  
    x = "fantastic"  
  
myfunc()  
  
print("Python is " + x)
```

op: Python is fantastic

Also, use the `global` keyword if you want to change a global variable inside a function.

Example

To change the value of a global variable inside a function, refer to the variable by using the `global` keyword:

```
x = "awesome"  
  
def myfunc():  
    global x  
    x = "fantastic"  
  
myfunc()  
  
print("Python is " + x)
```

op: Python is fantastic

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
None Type: NoneType

Getting the Data Type

You can get the data type of any object by using the `type()` function:

Example

Print the data type of the variable x:

```
x = 5  
print(type(x))
```

Setting the Data Type

In Python, the data type is set when you assign a value to a variable:

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name" : "John", "age" : 36}	dict
x = {"apple", "banana", "cherry"}	set
x = frozenset({"apple", "banana", "cherry"})	frozenset
x = True	bool
x = b"Hello"	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview
x = None	NoneType

Setting the Specific Data Type

If you want to specify the data type, you can use the following constructor functions:

Example	Data Type
<code>x = str("Hello World")</code>	<code>str</code>
<code>x = int(20)</code>	<code>int</code>
<code>x = float(20.5)</code>	<code>float</code>
<code>x = complex(1j)</code>	<code>complex</code>
<code>x = list(("apple", "banana", "cherry"))</code>	<code>list</code>
<code>x = tuple(("apple", "banana", "cherry"))</code>	<code>tuple</code>
<code>x = range(6)</code>	<code>range</code>
<code>x = dict(name="John", age=36)</code>	<code>dict</code>
<code>x = set(("apple", "banana", "cherry"))</code>	<code>set</code>
<code>x = frozenset(("apple", "banana", "cherry"))</code>	<code>frozenset</code>
<code>x = bool(5)</code>	<code>bool</code>
<code>x = bytes(5)</code>	<code>bytes</code>
<code>x = bytearray(5)</code>	<code>bytearray</code>
<code>x = memoryview(bytes(5))</code>	<code>memoryview</code>

Python Numbers

Python Numbers

There are three numeric types in Python:

- `int`
- `float`
- `complex`

Variables of numeric types are created when you assign a value to them:

Example

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

Int

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

Example

Integers:

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

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

Float

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

Example

Floats:

```
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.

Example

Floats:

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

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

Complex

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

Example

Complex:

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

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

Type Conversion

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

Example

Convert from one type to another:

```
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))
```

```
op:      1.0
        2
        (1+0j)
        <class 'float'>
        <class 'int'>
        <class 'complex'>
```

Note: You cannot convert complex numbers into another number type.

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:

Example

Import the random module, and display a random number between 1 and 9:

```
import random
```

```
print(random.randrange(1, 10)) # Generates a random number every time the code runs.
```

Python Casting

Specify a Variable Type

There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types.

Casting in python is therefore done using constructor functions:

- `int()` - constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number)
- `float()` - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- `str()` - constructs a string from a wide variety of data types, including strings, integer literals and float literals

Example

Integers:

```
x = int(1) # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3
```

Example

Floats:

```
x = float(1) # x will be 1.0
y = float(2.8) # y will be 2.8
z = float("3") # z will be 3.0
w = float("4.2") # w will be 4.2
```

Example

Strings:

```
x = str("s1") # x will be 's1'
y = str(2) # y will be '2'
z = str(3.0) # z will be '3.0'
```

Python Strings

Strings

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

'hello' is the same as "hello".

You can display a string literal with the `print()` function:

Example

```
print("Hello")  
print('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:

Example

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

Multiline Strings

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

Example

You can use three double quotes:

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

Strings are Arrays

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.

However, Python does not have a character data type, a single character is simply a string with a length of 1.

Square brackets can be used to access elements of the string.

Example

Get the character at position 1 (remember that the first character has the position 0):

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

op: e

Looping Through a String

Since strings are arrays, we can loop through the characters in a string, with a `for` loop.

Example

Loop through the letters in the word "banana":

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

op: b
 a
 n
 a
 n
 a

String Length

To get the length of a string, use the `len()` function.

Example

The `len()` function returns the length of a string:

```
a = "Hello, World!"  
print(len(a))
```

op: 13

Check String

To check if a certain phrase or character is present in a string, we can use the keyword `in`.

Example

Check if "free" is present in the following text:

```
txt = "The best things in life are free!"  
print("free" in txt)
```

op: True

Use it in an `if` statement:

Example

Print only if "free" is present:

```
txt = "The best things in life are free!"  
if "free" in txt:  
    print("Yes, 'free' is present.")
```

op: Yes, 'free' is present.

Check if NOT

To check if a certain phrase or character is NOT present in a string, we can use the keyword `not` `in`.

Example

Check if "expensive" is NOT present in the following text:

```
txt = "The best things in life are free!"  
print("expensive" not in txt)
```

op: True

Use it in an `if` statement:

Example

print only if "expensive" is NOT present:

```
txt = "The best things in life are free!"  
if "expensive" not in txt:  
    print("No, 'expensive' is NOT present.")
```

op: No, 'expensive' is NOT present.

Python - Slicing Strings

Slicing

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.

Example

Get the characters from position 2 to position 5 (not included):

```
b = "Hello, World!"  
print(b[2:5])
```

op: llo

Slice From the Start

By leaving out the start index, the range will start at the first character:

Example

Get the characters from the start to position 5 (not included):

```
b = "Hello, World!"  
print(b[:5])
```

op: Hello

Slice To the End

By leaving out the *end* index, the range will go to the end:

Example

Get the characters from position 2, and all the way to the end:

```
b = "Hello, World!"  
print(b[2:])
```

op: llo, World!

Negative Indexing

Use negative indexes to start the slice from the end of the string:

Example

Get the characters:

From: "o" in "World!" (position -5)

To, but not included: "d" in "World!" (position -2):

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

op: orl

Python - Modify Strings

Python has a set of built-in methods that you can use on strings.

Upper Case

Example

The `upper()` method returns the string in upper case:

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

op: HELLO, WORLD!

Lower Case

Example

The `lower()` method returns the string in lower case:

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

Remove Whitespace

Whitespace is the space before and/or after the actual text, and very often you want to remove this space.

Example

The `strip()` method removes any whitespace from the beginning or the end:

```
a = " Hello, World! "  
print(a.strip()) # returns "Hello, World!"
```

Replace String

Example

The `replace()` method replaces a string with another string:

```
a = "Hello, World!"  
print(a.replace("H", "J"))
```

op: Jello, World!

Split String

The `split()` method returns a list where the text between the specified separator becomes the list items.

Example

The `split()` method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"  
print(a.split(",")) # returns ['Hello', ' World!']
```

Python - String Concatenation

String Concatenation

To concatenate, or combine, two strings you can use the `+` operator.

Example

Merge variable `a` with variable `b` into variable `c`:

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

opp: HelloWorld

Python - Format - Strings

String Format

As we learned in the Python Variables chapter, we cannot combine strings and numbers like this:

Example

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

```
op:      Traceback (most recent call last):  
        File "demo_string_format_error.py", line 2, in <module>  
          txt = "My name is John, I am " + age  
        TypeError: must be str, not int
```

But we can combine strings and numbers by using the `format()` method!

The `format()` method takes the passed arguments, formats them, and places them in the string where the placeholders `{}` are:

Example

Use the `format ()` method to insert numbers into strings:

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```

op: My name is John, and I am 36

```
age = 36
height = 5.5
txt = "My name is John, and I am {} and height is {}"
print(txt.format(age,height))
```

op: My name is John, and I am 36 and height is 5.5

You can use index numbers `{0}` to be sure the arguments are placed in the correct placeholders:

Example

```
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))
```

op: I want to pay 49.95 dollars for 3 pieces of item 567

Python - Escape Characters

Escape Character

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.

Escape Characters

Other escape characters used in Python:

Code	Result
<code>\'</code>	Single Quote
<code>\\</code>	Backslash
<code>\n</code>	New Line
<code>\r</code>	Carriage Return
<code>\t</code>	Tab
<code>\b</code>	Backspace


```
txt = 'It\'s alright.'  
print(txt)
```

op: It's alright.

```
txt = "This will insert one \\ (backslash)."  
print(txt)
```

op: This will insert one \ (backslash).

```
txt = "Hello\nWorld!"  
print(txt)
```

op: Hello
 World!

```
txt = "Hello\tWorld!"  
print(txt)
```

op: Hello World!

```
#This example erases one character (backspace):  
txt = "Hello \bWorld!"  
print(txt)
```

op: HelloWorld!

Python Booleans

Booleans represent one of two values: `True` or `False`.

Boolean Values

In programming you often need to know if an expression is `True` or `False`.

You can evaluate any expression in Python, and get one of two answers, `True` or `False`.

When you compare two values, the expression is evaluated and Python returns the Boolean answer:

Example

```
print(10 > 9)  
print(10 == 9)  
print(10 < 9)
```

When you run a condition in an if statement, Python returns `True` or `False`:

Example

Print a message based on whether the condition is `True` or `False`:

```
a = 200
b = 33
```

```
if b > a:
    print("b is greater than a")
else:
    print("b is not greater than a")
```

Evaluate Values and Variables

The `bool()` function allows you to evaluate any value, and give you `True` or `False` in return,

Example

Evaluate a string and a number:

```
print(bool("Hello"))
print(bool(15))
```

Most Values are True

Almost any value is evaluated to `True` if it has some sort of content.

Any string is `True`, except empty strings.

Any number is `True`, except `0`.

Any list, tuple, set, and dictionary are `True`, except empty ones.

Some Values are False

In fact, there are not many values that evaluate to `False`, except empty values, such as `()`, `[]`, `{}`, `""`, the number `0`, and the value `None`. And of course the value `False` evaluates to `False`.

Example

The following will return `False`:

```
bool(False)
bool(None)
bool(0)
bool("")
bool(())
bool([])
bool({})
```

Functions can Return a Boolean

You can create functions that returns a Boolean Value:

Example

Print the answer of a function:

```
def myFunction() :  
    return True  
  
print(myFunction())
```

You can execute code based on the Boolean answer of a function:

Example

Print "YES!" if the function returns True, otherwise print "NO!":

```
def myFunction() :  
    return True  
  
if myFunction():  
    print("YES!")  
else:  
    print("NO!")
```

Python Operators

Python 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

Python Arithmetic Operators

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

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y

%	Modulus	<code>x % y</code>
**	Exponentiation	<code>x ** y</code>
//	Floor division	<code>x // y</code>

Python Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	<code>x = 5</code>	<code>x = 5</code>
+=	<code>x += 3</code>	<code>x = x + 3</code>
-=	<code>x -= 3</code>	<code>x = x - 3</code>
*=	<code>x *= 3</code>	<code>x = x * 3</code>
/=	<code>x /= 3</code>	<code>x = x / 3</code>
%=	<code>x %= 3</code>	<code>x = x % 3</code>
//=	<code>x //= 3</code>	<code>x = x // 3</code>
**=	<code>x **= 3</code>	<code>x = x ** 3</code>
&=	<code>x &= 3</code>	<code>x = x & 3</code>
=	<code>x = 3</code>	<code>x = x 3</code>
^=	<code>x ^= 3</code>	<code>x = x ^ 3</code>
>>=	<code>x >>= 3</code>	<code>x = x >> 3</code>
<<=	<code>x <<= 3</code>	<code>x = x << 3</code>

Python Bitwise operators

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name.

For example, 2 is 10 in binary and 7 is 111.

In the table below: Let `x = 10` (0000 1010 in binary) and `y = 4` (0000 0100 in binary)

Operator	Meaning	Example
&	Bitwise AND	<code>x & y = 0</code> (0000 0000)
	Bitwise OR	<code>x y = 14</code> (0000 1110)
~	Bitwise NOT	<code>~x = -11</code> (1111 0101)
^	Bitwise XOR	<code>x ^ y = 14</code> (0000 1110)
>>	Bitwise right shift	<code>x >> 2 = 2</code> (0000 0010)
<<	Bitwise left shift	<code>x << 2 = 40</code> (0010 1000)

Python Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example
==	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

Python Logical Operators

Logical operators are used to combine conditional statements:

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)

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

Example

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x
```

```
print(x is z)
```

```
# returns True because z is the same object as x
```

```
print(x is y)
```

returns False because x is not the same object as y, even if they have the same content

```
print(x == y)
```

to demonstrate the difference between "is" and "==": this comparison returns True because x is equal to y

Example

```
x = ["apple", "banana"]
```

```
y = ["apple", "banana"]
```

```
z = x
```

```
print(x is not z)
```

returns False because z is the same object as x

```
print(x is not y)
```

returns True because x is not the same object as y, even if they have the same content

```
print(x != y)
```

to demonstrate the difference between "is not" and "!=": this comparison returns False because x is equal to y

Python Membership Operators

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

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Example

```
x = ["apple", "banana"]
```

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

```
# returns True because a sequence with the value "banana" is in the list
```

Example

```
x = ["apple", "banana"]
```

```
print("pineapple" not in x)
```

```
# returns True because a sequence with the value "pineapple" is not in the list
```

Operator Precedence

Operator precedence describes the order in which operations are performed.

The precedence order is described in the table below, starting with the highest precedence at the top:

Operator	Description
()	Parentheses
**	Exponentiation
+x -x ~x	Unary plus, unary minus, and bitwise NOT
* / // %	Multiplication, division, floor division, and modulus
+ -	Addition and subtraction
<< >>	Bitwise left and right shifts
&	Bitwise AND
^	Bitwise XOR
	Bitwise OR
== != > >= < <= is is not in not in	Comparisons, identity, and membership operators
not	Logical NOT
and	AND
or	OR

If two operators have the same precedence, the expression is evaluated from left to right.

Addition + and subtraction - has the same precedence, and therefor we evaluate the expression from left to right:

```
print(5 + 4 - 7 + 3)
```

Python Lists

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

List

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.

Lists are created using square brackets:

Create a List:

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

List Items

List items are ordered, changeable, and allow duplicate values.

List items are indexed, the first item has index `[0]`, the second item has index `[1]` etc.

Ordered

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

Note: There are some [list methods](#) that will change the order, but in general: the order of the items will not change.

Changeable

The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Allow Duplicates

Since lists are indexed, lists can have items with the same value:

Example

Lists allow duplicate values:

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

List Length

To determine how many items a list has, use the `len()` function:

Example

Print the number of items in the list:

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

List Items - Data Types

List items can be of any data type:

Example

String, int and boolean data types:

```
list1 = ["apple", "banana", "cherry"]  
list2 = [1, 5, 7, 9, 3]  
list3 = [True, False, False]
```

A list can contain different data types:

Example

A list with strings, integers and boolean values:

```
list1 = ["abc", 34, True, 40, "male"]
```

type()

From Python's perspective, lists are defined as objects with the data type 'list':

```
<class 'list'>
```

Example

What is the data type of a list?

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

The list() Constructor

It is also possible to use the list() constructor when creating a new list.

Example

Using the list() constructor to make a List:

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

Python Collections (Arrays)

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

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered** and changeable. No duplicate members.

*Set *items* are unchangeable, but you can remove and/or add items whenever you like.

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

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

Access Items

List items are indexed and you can access them by referring to the index number:

Example

Print the second item of the list:

```
thislist = ["apple", "banana", "cherry"]  
print(thislist[1])
```

Negative Indexing

Negative indexing means start from the end

- 1 refers to the last item, - 2 refers to the second last item etc.

Change Item Value

To change the value of a specific item, refer to the index number:

Example

Change the second item:

```
thislist = ["apple", "banana", "cherry"]  
thislist[1] = "blackcurrant"  
print(thislist)
```

op: ['apple', 'blackcurrant', 'cherry']

Change a Range of Item Values

To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values:

Example

Change the values "banana" and "cherry" with the values "blackcurrant" and "watermelon":

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

op: ['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']

If you insert *more* items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

Example

Change the second value by replacing it with *two* new values:

```
thislist = ["apple", "banana", "cherry"]  
thislist[1:2] = ["blackcurrant", "watermelon"]  
print(thislist)
```

op: ['apple', 'blackcurrant', 'watermelon', 'cherry']

Note: The length of the list will change when the number of items inserted does not match the number of items replaced.

If you insert *less* items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

Example

Change the second and third value by replacing it with *one* value:

```
thislist = ["apple", "banana", "cherry"]  
thislist[1:3] = ["watermelon"]  
print(thislist)
```

op: ['apple', 'watermelon']

Insert Items

To insert a new list item, without replacing any of the existing values, we can use the `insert()` method.

The `insert()` method inserts an item at the specified index:

Example

Insert "watermelon" as the third item:

```
thislist = ["apple", "banana", "cherry"]  
thislist.insert(2, "watermelon")  
print(thislist)
```

op: ['apple', 'banana', 'watermelon', 'cherry']

Note: As a result of the example above, the list will now contain 4 items.

Python - Add List Items

Append Items

To add an item to the end of the list, use the `append()` method:

Example

Using the `append()` method to append an item:

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

op: ['apple', 'banana', 'cherry', 'orange']

Insert Items

To insert a list item at a specified index, use the `insert()` method.

The `insert()` method inserts an item at the specified index:

Example

Insert an item as the second position:

```
thislist = ["apple", "banana", "cherry"]  
thislist.insert(1, "orange")  
print(thislist)
```

op: ['apple', 'orange', 'banana', 'cherry']

Extend List

To append elements from *another list* to the current list, use the `extend()` method.

Example

Add the elements of `tropical` to `thislist`:

```
thislist = ["apple", "banana", "cherry"]  
tropical = ["mango", "pineapple", "papaya"]  
thislist.extend(tropical)  
print(thislist)
```

op: ['apple', 'banana', 'cherry', 'mango', 'pineapple', 'papaya']

Add Any Iterable

The `extend()` method does not have to append *lists*, you can add any iterable object (tuples, sets, dictionaries etc.).

Example

Add elements of a tuple to a list:

```
thislist = ["apple", "banana", "cherry"]  
thistuple = ("kiwi", "orange")  
thislist.extend(thistuple)  
print(thislist)
```

op: ['apple', 'banana', 'cherry', 'kiwi', 'orange']

Remove Specified Item

The `remove()` method removes the specified item.

Example

Remove "banana":

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

op: ['apple', 'cherry']

Remove Specified Index

The `pop()` method removes the specified index.

Example

Remove the second item:

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

op: ['apple', 'cherry']

If you do not specify the index, the `pop()` method removes the last item.

Example

Remove the last item:

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

op: ['apple', 'banana']

The `del` keyword also removes the specified index:

Example

Remove the first item:

```
thislist = ["apple", "banana", "cherry"]  
del thislist[0]  
print(thislist)
```

op: ['banana', 'cherry']

The `del` keyword can also delete the list completely.

Example

Delete the entire list:

```
thislist = ["apple", "banana", "cherry"]  
del thislist
```

op: Traceback (most recent call last):

File "demo_list_del2.py", line 3, in <module>

print(thislist) #this will cause an error because you have succsesfully deleted "thislist".

NameError: name 'thislist' is not defined

Clear the List

The `clear()` method empties the list.

The list still remains, but it has no content.

Example

Clear the list content:

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

op: []

Python - Loop Lists

Loop Through a List

You can loop through the list items by using a `for` loop:

Example

Print all items in the list, one by one:

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

Loop Through the Index Numbers

You can also loop through the list items by referring to their index number.

Use the `range()` and `len()` functions to create a suitable iterable.

Example

Print all items by referring to their index number:

```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
    print(thislist[i])
```

Using a While Loop

You can loop through the list items by using a `while` loop.

Use the `len()` function to determine the length of the list, then start at 0 and loop your way through the list items by referring to their indexes.

Remember to increase the index by 1 after each iteration.

Example

Print all items, using a `while` loop to go through all the index numbers

```
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
    print(thislist[i])
    i = i + 1
```

Looping Using List Comprehension

List Comprehension offers the shortest syntax for looping through lists:

Example

A short hand `for` loop that will print all items in a list:

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

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Example:

Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.

Without list comprehension you will have to write a `for` statement with a conditional test inside:

Example

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []
```

```
for x in fruits:
    if "a" in x:
        newlist.append(x)
```

```
print(newlist)
```

```
op: ['apple', 'banana', 'mango']
```

With list comprehension you can do all that with only one line of code:

Example

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
```

```
newlist = [x for x in fruits if "a" in x]
```

```
print(newlist)
```

The Syntax

```
newlist = [expression for item in iterable if condition == True]
```

The return value is a new list, leaving the old list unchanged.

Condition

The *condition* is like a filter that only accepts the items that valuate to `True`.

Example

Only accept items that are not "apple":

```
newlist = [x for x in fruits if x != "apple"]
```

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

The *condition* is optional and can be omitted:

Example

With no `if` statement:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
```

```
newlist = [x for x in fruits]
```

```
print(newlist) #newlist now contains all the members of fruit list
```

Iterable

The *iterable* can be any iterable object, like a list, tuple, set etc.

Example

You can use the `range()` function to create an iterable:

```
newlist = [x for x in range(10)]
```

Example:

```
newlist = [x for x in range(10)]
```

```
print(newlist)
```

op: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Example:

```
newlist = [x for x in range(10) if x < 5]
```

```
print(newlist)
```

op: [0, 1, 2, 3, 4]

Expression

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

Example

Set the values in the new list to upper case:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
```

```
newlist = [x.upper() for x in fruits]
```

```
print(newlist)
```

op: ['APPLE', 'BANANA', 'CHERRY', 'KIWI', 'MANGO']

Example

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
```

```
newlist = ['hello' for x in fruits]
```

```
print(newlist)
```

```
op: ['hello', 'hello', 'hello', 'hello', 'hello']
```

Example:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
```

```
newlist = [x if x != "banana" else "orange" for x in fruits]
```

```
print(newlist) ##Return the item if it is not banana, if it is banana return orange.
```

```
Op: ['apple', 'orange', 'cherry', 'kiwi', 'mango']
```

Sort List Alphanumerically

List objects have a `sort ()` method that will sort the list alphanumerically, ascending, by default:

Example

Sort the list alphabetically:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
```

```
thislist.sort()
```

```
print(thislist)
```

```
op: ['banana', 'kiwi', 'mango', 'orange', 'pineapple']
```

Example

Sort the list numerically:

```
thislist = [100, 50, 65, 82, 23]
```

```
thislist.sort()
```

```
print(thislist)
```

op: [23, 50, 65, 82, 100]

Sort Descending

To sort descending, use the keyword argument `reverse = True`:

Example

Sort the list descending:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)
```

Example

Sort the list descending:

```
thislist = [100, 50, 65, 82, 23]
thislist.sort(reverse = True)
print(thislist)
```

Python - Copy Lists

Copy a List

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`.

There are ways to make a copy, one way is to use the built-in List method `copy()`.

Example

Make a copy of a list with the `copy()` method:

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

Another way to make a copy is to use the built-in method `list()`.

Example

Make a copy of a list with the `list()` method:

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

Join Two Lists

There are several ways to join, or concatenate, two or more lists in Python.

One of the easiest ways are by using the + operator.

Example

Join two list:

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

```
list3 = list1 + list2  
print(list3)
```

Another way to join two lists is by appending all the items from list2 into list1, one by one:

Example

Append list2 into list1:

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

```
for x in list2:  
    list1.append(x)
```

```
print(list1)
```

Or you can use the `extend()` method, which purpose is to add elements from one list to another list:

Example

Use the `extend()` method to add list2 at the end of list1:

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

```
list1.extend(list2)  
print(list1)
```

List Methods

Python has a set of built-in methods that you can use on lists.

Method	Description
<code>append()</code>	Adds an element at the end of the list
<code>clear()</code>	Removes all the elements from the list
<code>copy()</code>	Returns a copy of the list

[count\(\)](#) Returns the number of elements with the specified value
[extend\(\)](#) Add the elements of a list (or any iterable), to the end of the current list
[index\(\)](#) Returns the index of the first element with the specified value
[insert\(\)](#) Adds an element at the specified position
[pop\(\)](#) Removes the element at the specified position
[remove\(\)](#) Removes the item with the specified value
[reverse\(\)](#) Reverses the order of the list
[sort\(\)](#) Sorts the list

Python Tuples

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

Tuple

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.

Example

Create a Tuple:

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

Tuple Items

Tuple items are ordered, unchangeable, and allow duplicate values.

Tuple items are indexed, the first item has index `[0]`, the second item has index `[1]` etc.

Ordered

When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

Unchangeable

Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

Allow Duplicates

Since tuples are indexed, they can have items with the same value:

Example

Tuples allow duplicate values:

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

Tuple Length

To determine how many items a tuple has, use the `len()` function:

Example

Print the number of items in the tuple:

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

op: 3

Create Tuple With One Item

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

Example

One item tuple, remember the comma:

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

#NOT a tuple

```
thistuple = ("apple")  
print(type(thistuple))
```

Tuple Items - Data Types

Tuple items can be of any data type:

Example

String, int and boolean data types:

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

A tuple can contain different data types:

Example

A tuple with strings, integers and boolean values:

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

type()

From Python's perspective, tuples are defined as objects with the data type 'tuple':

```
<class 'tuple'>
```

Example

What is the data type of a tuple?

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

The tuple() Constructor

It is also possible to use the tuple() constructor to make a tuple.

Example

Using the tuple() method to make a tuple:

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

Access Tuple Items

You can access tuple items by referring to the index number, inside square brackets:

Example

Print the second item in the tuple:

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

Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new tuple with the specified items.

Example

Return the third, fourth, and fifth item:

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")  
print(thistuple[2:5])
```


op: ('cherry', 'orange', 'kiwi')

#This will return the items from position 2 to 5.

#Remember that the first item is position 0,

#and note that the item in position 5 is NOT included

Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the tuple:

Example

This example returns the items from index -4 (included) to index -1 (excluded)

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")  
print(thistuple[-4:-1])
```

op: ('orange', 'kiwi', 'melon')

#Negative indexing means starting from the end of the tuple.

#This example returns the items from index -4 (included) to index -1 (excluded)

#Remember that the last item has the index -1,

Check if Item Exists

To determine if a specified item is present in a tuple use the `in` keyword:

Example

Check if "apple" is present in the tuple:

```
thistuple = ("apple", "banana", "cherry")  
if "apple" in thistuple:  
    print("Yes, 'apple' is in the fruits tuple")
```

Python - Update Tuples

Tuples are **unchangeable**, meaning that you cannot change, add, or remove items once the tuple is created.

But there are some workarounds.

Change Tuple Values

Once a tuple is created, you cannot change its values. Tuples are **unchangeable**, or **immutable** as it also is called.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

Example

Convert the tuple into a list to be able to change it:

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
```

```
print(x)
```

```
op: ("apple", "kiwi", "cherry")
```

Add Items

Since tuples are immutable, they do not have a build-in `append()` method, but there are other ways to add items to a tuple.

1. **Convert into a list:** Just like the workaround for *changing* a tuple, you can convert it into a list, add your item(s), and convert it back into a tuple.

Example

Convert the tuple into a list, add "orange", and convert it back into a tuple:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
```

```
op: ('apple', 'banana', 'cherry', 'orange')
```

2. **Add tuple to a tuple.** You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple:

Example

Create a new tuple with the value "orange", and add that tuple:

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y
```

```
print(thistuple)
```

```
op: ('apple', 'banana', 'cherry', 'orange')
```

Remove Items

Note: You cannot remove items in a tuple.

Tuples are **unchangeable**, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items:

Example

Convert the tuple into a list, remove "apple", and convert it back into a tuple:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
```

Or you can delete the tuple completely:

Example

The `del` keyword can delete the tuple completely:

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

Python - Unpack Tuples**

Unpacking a Tuple

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

Example

Packing a tuple:

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

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

Example

Unpacking a tuple:

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

```
(x,y,z) = fruits
```

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

```
op:   apple
      banana
      cherry
```

Note: The number of variables must match the number of values in the tuple, if not, you must use an asterisk to collect the remaining values as a list.

Using Asterisk *

If the number of variables is less than the number of values, you can add an * to the variable name and the values will be assigned to the variable as a list:

Example

Assign the rest of the values as a list called "red":

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

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

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

```
op:   apple
      banana
      ['cherry', 'strawberry', 'raspberry']
```

If the asterisk is added to another variable name than the last, Python will assign values to the variable until the number of values left matches the number of variables left.

Example

Add a list of values the "tropic" variable:

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

```
(x, *y, z) = fruits
```

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

```
op:   apple
      ['mango', 'papaya', 'pineapple']
      cherry
```

Python - Loop Tuples

Loop Through a Tuple

You can loop through the tuple items by using a `for` loop.

Example

Iterate through the items and print the values:

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
    print(x)
```

```
op:    apple
       banana
       cherry
```

Loop Through the Index Numbers

You can also loop through the tuple items by referring to their index number.

Use the `range()` and `len()` functions to create a suitable iterable.

Example

Print all items by referring to their index number:

```
thistuple = ("apple", "banana", "cherry")
for i in range(len(thistuple)):
    print(thistuple[i])
```

```
op:    apple
       banana
       cherry
```

Using a While Loop

You can loop through the tuple items by using a `while` loop.

Use the `len()` function to determine the length of the tuple, then start at 0 and loop your way through the tuple items by referring to their indexes.

Remember to increase the index by 1 after each iteration.

Example

Print all items, using a `while` loop to go through all the index numbers:

```
thistuple = ("apple", "banana", "cherry")
i = 0
while i < len(thistuple):
    print(thistuple[i])
    i = i + 1
```

```
op:  apple
     banana
     cherry
```

Join Two Tuples

To join two or more tuples you can use the + operator:

Example

Join two tuples:

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

```
tuple3 = tuple1 + tuple2
print(tuple3)
```

Multiply Tuples

If you want to multiply the content of a tuple a given number of times, you can use the * operator:

Example

Multiply the fruits tuple by 2:

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2
```

```
print(mytuple)
```

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

Python Sets

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

Set

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*.

*** Note:** Set *items* are unchangeable, but you can remove items and add new items.

Sets are written with curly brackets.

Example

Create a Set:

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

Note: the set list is unordered, meaning: the items will appear in a random order.

Refresh this page to see the change in the result.

Set Items

Set items are unordered, unchangeable, and do not allow duplicate values.

Unordered

Unordered means that the items in a set do not have a defined order.

Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

Unchangeable

Set items are unchangeable, meaning that we cannot change the items after the set has been created.

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

Duplicates Not Allowed

Sets cannot have two items with the same value.

Example

Duplicate values will be ignored:

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

```
print(thisset)
```

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

Note: The values `True` and `1` are considered the same value in sets, and are treated as duplicates:

Example

`True` and `1` is considered the same value:

```
thisset = {"apple", "banana", "cherry", True, 1, 2}
```

```
print(thisset)
```

Get the Length of a Set

To determine how many items a set has, use the `len()` function.

Example

Get the number of items in a set:

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

```
print(len(thisset))
```

Set Items - Data Types

Set items can be of any data type:

Example

String, int and boolean data types:

```
set1 = {"apple", "banana", "cherry"}
```

```
set2 = {1, 5, 7, 9, 3}
```

```
set3 = {True, False, False}
```

A set can contain different data types:

Example

A set with strings, integers and boolean values:

```
set1 = {"abc", 34, True, 40, "male"}
```

type()

From Python's perspective, sets are defined as objects with the data type 'set':

```
<class 'set'>
```

Example

What is the data type of a set?

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

```
print(type(myset))
```

The set() Constructor

It is also possible to use the `set()` constructor to make a set.

Example

Using the `set()` constructor to make a set:

```
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
```

```
print(thisset)
```


Access Items

You cannot access items in a set by referring to an index or a key.

But you can loop through the set items using a `for` loop, or ask if a specified value is present in a set, by using the `in` keyword.

Example

Loop through the set, and print the values:

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

```
for x in thisset:  
    print(x)
```

Add Items

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

To add one item to a set use the `add()` method.

Example

Add an item to a set, using the `add()` method:

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

```
thisset.add("orange")
```

```
print(thisset)
```

```
op: {'cherry', 'orange', 'banana', 'apple'}
```

Add Sets

To add items from another set into the current set, use the `update()` method.

Example

Add elements from `tropical` into `thisset`:

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

```
tropical = {"pineapple", "mango", "papaya"}
```

```
thisset.update(tropical)
```

```
print(thisset)
```

Add Any Iterable

The object in the `update()` method does not have to be a set, it can be any iterable object (tuples, lists, dictionaries etc.).

Example

Add elements of a list to a set:

```
thisset = {"apple", "banana", "cherry"}  
mylist = ["kiwi", "orange"]
```

```
thisset.update(mylist)
```

```
print(thisset)
```

```
op: {'banana', 'cherry', 'apple', 'orange', 'kiwi'}
```

Remove Item

To remove an item in a set, use the `remove()`, or the `discard()` method.

Example

Remove "banana" by using the `remove()` method:

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

```
thisset.remove("banana")
```

```
print(thisset)
```

Note: If the item to remove does not exist, `remove()` will raise an error.

Example

Remove "banana" by using the `discard()` method:

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

```
thisset.discard("banana")
```

```
print(thisset)
```

Note: If the item to remove does not exist, `discard()` will **NOT** raise an error.

You can also use the `pop()` method to remove an item, but this method will remove a random item, so you cannot be sure what item that gets removed.

The return value of the `pop()` method is the removed item.

Example

Remove a random item by using the `pop()` method:

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

```
x = thisset.pop()
```

```
print(x)
```

```
print(thisset)
```

The `clear()` method empties the set:

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

```
thisset.clear()
```

```
print(thisset)
```

Example

The `del` keyword will delete the set completely:

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

```
del thisset
```

```
print(thisset)
```

Loop Items

You can loop through the set items by using a `for` loop:

Example

Loop through the set, and print the values:

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

```
for x in thisset:
```

```
    print(x)
```

Join Two Sets

There are several ways to join two or more sets in Python.

You can use the `union()` method that returns a new set containing all items from both sets, or the `update()` method that inserts all the items from one set into another:

Example

The `union()` method returns a new set with all items from both sets:

```
set1 = {"a", "b", "c"}
```

```
set2 = {1, 2, 3}
```

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

The `update ()` method inserts the items in set2 into set1:

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

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

Python Dictionaries

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

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

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.

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.

Dictionary Items

Dictionary items are ordered, changeable, and does not allow duplicates.

Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

Example

Print the "brand" value of the dictionary:

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

```
}  
print(thisdict["brand"])
```

Changeable

Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

Duplicates Not Allowed

Dictionaries cannot have two items with the same key:

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

```
op: {'brand': 'Ford', 'model': 'Mustang', 'year': 2020} #Duplicate values will overwrite existing  
values
```

Dictionary Length

To determine how many items a dictionary has, use the `len()` function:

Example

Print the number of items in the dictionary:

```
print(len(thisdict))
```

Dictionary Items - Data Types

The values in dictionary items can be of any data type:

Example

String, int, boolean, and list data types:

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

```
op: {'brand': 'Ford', 'electric': False, 'year': 1964, 'colors': ['red', 'white', 'blue']}
```

type()

From Python's perspective, dictionaries are defined as objects with the data type 'dict':

```
<class 'dict'>
```

Example

Print the data type of a dictionary:

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

The dict() Constructor

It is also possible to use the dict() constructor to make a dictionary.

Example

Using the dict() method to make a dictionary:

```
thisdict = dict(name = "John", age = 36, country = "Norway")  
print(thisdict)
```

Python - Access Dictionary Items

Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets:

Example

Get the value of the "model" key:

```
thisdict = {  
    "brand": "Maruti",  
    "model": "Swift",  
    "year": 2021  
}  
x = thisdict["model"]  
print(x)
```

op: Swift

There is also a method called `get()` that will give you the same result:

Example

Get the value of the "model" key:

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

Get Keys

The `keys()` method will return a list of all the keys in the dictionary.

Example

Get a list of the keys:

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

The list of the keys is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the keys list.

Example:

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

```
x = car.keys()
```

```
print(x) #before the change  
print(len(x))
```

```
car["color"] = "white"
```

```
print(x) #after the change
```

```
op:    dict_keys(['brand', 'model', 'year'])  
      3  
      dict_keys(['brand', 'model', 'year', 'color'])
```

Get Values

The `values()` method will return a list of all the values in the dictionary.

Example

Get a list of the values:

```
x = thisdict.values()
```

The list of the values is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the values list.

Example

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

```
x = car.values()
```

```
print(x) #before the change
```

```
car["year"] = 2020  
car["model"] = "Maruti"
```

```
print(x) #after the change
```

```
op:    dict_values(['Ford', 'Mustang', 1964])  
       dict_values(['Ford', 'Maruti', 2020])
```

Get Items

The `items()` method will return each item in a dictionary, as tuples in a list.

Example

Get a list of the key:value pairs

```
x = thisdict.items()
```

The returned list is a *view* of the items of the dictionary, meaning that any changes done to the dictionary will be reflected in the items list.

Example

Make a change in the original dictionary, and see that the items list gets updated as well:

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

```
x = car.items()
```

```
print(x) #before the change
```

```
car["year"] = 2020
```

```
print(x) #after the change
```



```
op: dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 2020)])
```

Example

Add a new item to the original dictionary, and see that the items list gets updated as well:

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

```
x = car.items()
```

```
print(x) #before the change
```

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

```
print(x) #after the change
```

```
op: dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964), ('color', 'red')])
```

Check if Key Exists

To determine if a specified key is present in a dictionary use the `in` keyword:

Example

Check if "model" is present in the dictionary:

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

```
if "model" in thisdict:
```

```
    print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

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

```
if "color" in thisdict:
    print("Yes, 'color' is one of the keys in the thisdict dictionary")
else:
    print("There's no color key")
```

op: ??

Python - Change Dictionary Items

Change Values

You can change the value of a specific item by referring to its key name:

Example

Change the "year" to 2018:

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
thisdict["year"] = 2018
```

Update Dictionary

The `update()` method will update the dictionary with the items from the given argument.

The argument must be a dictionary, or an iterable object with key:value pairs.

Example

Update the "year" of the car by using the `update()` method:

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

Python - Add Dictionary Items

Adding Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

Example

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

op: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}

Update Dictionary

The `update()` method will update the dictionary with the items from a given argument. If the item does not exist, the item will be added.

The argument must be a dictionary, or an iterable object with key:value pairs.

Example

Add a color item to the dictionary by using the `update()` method:

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

Removing Items

There are several methods to remove items from a dictionary:

Example

The `pop()` method removes the item with the specified key name:

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

op: {'brand': 'Ford', 'year': 1964}

Example

The `del` keyword removes the item with the specified key name:

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

Example

The `clear()` method empties the dictionary:

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

Loop Through a Dictionary

You can loop through a dictionary by using a `for` loop.

When looping through a dictionary, the return value are the *keys* of the dictionary, but there are methods to return the *values* as well

Example

Print all key names in the dictionary, one by one:

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

Example

Print all *values* in the dictionary, one by one:

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

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

Example

You can also use the `values()` method to return values of a dictionary:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
for x in thisdict.values():  
    print(x)
```

Example

You can use the `keys()` method to return the keys of a dictionary:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
for x in thisdict.keys():  
    print(x)
```

Example

Loop through both *keys* and *values*, by using the `items()` method:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
for x, y in thisdict.items():
```

```
print(x, y)
```

```
op:    brand Ford
      model Mustang
      year 1964
```

Copy a Dictionary

You cannot copy a dictionary simply by typing `dict2 = dict1`, because: `dict2` will only be a *reference* to `dict1`, and changes made in `dict1` will automatically also be made in `dict2`.

There are ways to make a copy, one way is to use the built-in Dictionary method `copy()`.

Example

Make a copy of a dictionary with the `copy()` method:

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

Another way to make a copy is to use the built-in function `dict()`.

Example

Make a copy of a dictionary with the `dict()` function:

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

Python If ... Else

Python Conditions and If statements

Python supports the usual logical conditions from mathematics:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`

- Less than or equal to: $a \leq b$
- Greater than: $a > b$
- Greater than or equal to: $a \geq b$

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

If statement:

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

Elif

The elif keyword is Python's way of saying "if the previous conditions were not true, then try this condition".

Example

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

elif can also be used to test multiple "true" conditions.

Else

The else keyword catches anything which isn't caught by the preceding conditions.

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")
```

And

The and keyword is a logical operator, and is used to combine conditional statements:

Example

Test if **a** is greater than **b**, AND if **c** is greater than **a**:

```
a = 200
b = 33
c = 500
if a > b and c > a:
    print("Both conditions are True")
```

Or

The **or** keyword is a logical operator, and is used to combine conditional statements:

Example

Test if **a** is greater than **b**, OR if **a** is greater than **c**:

```
a = 200
b = 33
c = 500
if a > b or a > c:
    print("At least one of the conditions is True")
```

Not

The **not** keyword is a logical operator, and is used to reverse the result of the conditional statement:

Example

Test if **a** is NOT greater than **b**:

```
a = 33
b = 200
if not a > b:
    print("a is NOT greater than b")
```

Nested If

You can have **if** statements inside **if** statements, this is called *nested if* statements.

Example

```
x = 41

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


The pass Statement

`if` statements cannot be empty, but if you for some reason have an `if` statement with no content, put in the `pass` statement to avoid getting an error.

Example

```
a = 33
b = 200
```

```
if b > a:
    pass
```

Python Loops

Python has two primitive loop commands:

- while loops
 - for loops
-

The while Loop

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

Example

Print `i` as long as `i` is less than 6:

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

The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, `i`, which we set to 1.

The break Statement

With the `break` statement we can stop the loop even if the while condition is true:

Example

Exit the loop when `i` is 3:

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

The continue Statement

With the continue statement we can stop the current iteration, and continue with the next:

Example

Continue to the next iteration if i is 3:

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

The else Statement

With the else statement we can run a block of code once when the condition no longer is true:

Example

Print a message once the condition is false:

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

Python 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

Print each fruit in a fruit list:

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

The for loop does not require an indexing variable to set beforehand.

Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

Example

Loop through the letters in the word "banana":

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

The break Statement

With the break statement we can stop the loop before it has looped through all the items:

Example

Exit the loop when x is "banana":

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

The range() Function

To loop through a set of code a specified number of times, we can use the range() function,

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example

Using the range() function:

```
for x in range(6):  
    print(x)
```

Note that range(6) is not the values of 0 to 6, but the values 0 to 5.

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):

Example

Using the start parameter:

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

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):

Example

Increment the sequence with 3 (default is 1):

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

Else in For Loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished:

Example

Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x in range(6):  
    print(x)  
else:  
    print("Finally finished!")
```

Nested Loops

A nested loop is a loop inside a loop.

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

Example

Print each adjective for every fruit:

```
adj = ["red", "big", "tasty"]  
fruits = ["apple", "banana", "cherry"]
```

```
for x in adj:  
    for y in fruits:  
        print(x, y)
```

Python Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Creating a Function

In Python a function is defined using the def keyword:

Example

```
def my_function():  
    print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

Example

```
def my_function():  
    print("Hello from a function")
```

```
my_function() # function call
```

Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
def my_function(fname):  
    print(fname + " Welcome!!")
```

```
my_function("Emil")  
my_function("Tobias")  
my_function("Linus")
```

Parameters or Arguments?

The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

From a function's perspective:

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

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

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Example

This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, sname):  
    print(fname + "is a friend of " + sname)  
  
my_function("Ram", "Bhim")
```

Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

Example

If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*kids):  
    print("The youngest child is " + kids[2])  
  
my_function("Ram", "Bhim", "Farhan")
```

Keyword Arguments

You can also send arguments with the *key = value* syntax.

This way the order of the arguments does not matter.

Example

```
def my_function(child3, child2, child1):  
    print("The youngest child is " + child3)  
  
my_function(child1 = "Ram", child2 = "Bhim", child3 = "Farhan")
```

Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

Example

```
def my_function(country = "Norway"):  
    print("I am from " + country)  
  
my_function("Sweden")
```

```
my_function("India")
my_function()
my_function("Brazil")
```

Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

Example

```
def my_function(food):
    for x in food:
        print(x)
```

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

```
my_function(fruits)
```

Return Values

To let a function return a value, use the `return` statement:

Example

```
def my_function(x):
    return 5 * x
```

```
print(my_function(3))
print(my_function(5))
print(my_function(9))
```

The pass Statement

function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the `pass` statement to avoid getting an error.

Example

```
def myfunction():
    pass
```

Recursion

Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

Example

Recursion Example

```
def tri_recursion(k):  
    if(k > 0):  
        result = k + tri_recursion(k - 1)  
        print(result)  
    else:  
        result = 0  
    return result  
  
print("\n\nRecursion Example Results")  
tri_recursion(6)
```

Python Lambda

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

Syntax

lambda *arguments* : *expression*

The expression is executed and the result is returned:

Example

Add 10 to argument a, and return the result:

```
x = lambda a : a + 10  
print(x(5))
```

Lambda functions can take any number of arguments:

Example

Multiply argument a with argument b and return the result:

```
x = lambda a, b : a * b  
print(x(5, 6))
```

Example

Summarize argument a, b, and c and return the result:

```
x = lambda a, b, c : a + b + c  
print(x(5, 6, 2))
```


Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):  
    return lambda a : a * n
```

Use that function definition to make a function that always doubles the number you send in:

Example

```
def myfunc(n):  
    return lambda a : a * n
```

```
mydoubler = myfunc(2)
```

```
print(mydoubler(11))
```

op: 22

Example

```
def myfunc(n):  
    return lambda a : a * n
```

```
mydoubler = myfunc(2)
```

```
mytripler = myfunc(3)
```

```
print(mydoubler(11))
```

```
print(mytripler(11))
```

Python Classes and Objects**

Python Classes/Objects

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword `class`:

Example

Create a class named `MyClass`, with a property named `x`:

```
class MyClass:  
    x = 5
```

```
print(MyClass)
```

```
op: <class '__main__.MyClass'>
```

Create Object

Now we can use the class named MyClass to create objects:

Example

Create an object named p1, and print the value of x:

```
p1 = MyClass()  
print(p1.x)
```

The __init__() Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in __init__() function.

All classes have a function called __init__(), which is always executed when the class is being initiated.

Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Example

Create a class named Person, use the __init__() function to assign values for name and age:

```
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

```
p1 = Person("John", 36)
```

```
print(p1.name)  
print(p1.age)
```

Note: The __init__() function is called automatically every time the class is being used to create a new object.

The __str__() Function

The __str__() function controls what should be returned when the class object is represented as a string.

If the __str__() function is not set, the string representation of the object is returned:

Example

The string representation of an object WITHOUT the __str__() function:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
p1 = Person("John", 36)
```

```
print(p1)
```

```
op: <__main__.Person object at 0x15039e602100>
```

Example

The string representation of an object WITH the __str__() function:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f'{self.name}({self.age})'
```

```
p1 = Person("John", 36)
```

```
print(p1)
```

```
op: John(36)
```

Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Let us create a method in the Person class:

Example

Insert a function that prints a greeting, and execute it on the p1 object:

```
class Person:
    def __init__(self, name, age):
        self.name = name
```

```
self.age = age
```

```
def myfunc(self):  
    print("Hello my name is " + self.name)
```

```
p1 = Person("John", 36)  
p1.myfunc()
```

op: Hello my name is John

Note: The `self` parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

The self Parameter

The `self` parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named `self`, you can call it whatever you like, but it has to be the first parameter of any function in the class:

Example

Use the words *mysillyobject* and *abc* instead of *self*:

```
class Person:  
    def __init__(mysillyobject, name, age):  
        mysillyobject.name = name  
        mysillyobject.age = age  
  
    def myfunc(abc):  
        print("Hello my name is " + abc.name)
```

```
p1 = Person("John", 36)  
p1.myfunc()
```

op: Hello my name is John

Modify Object Properties

You can modify properties on objects like this:

Example

Set the age of p1 to 40:

```
p1.age = 40
```

Delete Object Properties

You can delete properties on objects by using the `del` keyword:

Example

Delete the age property from the p1 object:

```
del p1.age
```

Delete Objects

You can delete objects by using the `del` keyword:

Example

Delete the p1 object:

```
del p1
```

The pass Statement

`class` definitions cannot be empty, but if you for some reason have a `class` definition with no content, put in the `pass` statement to avoid getting an error.

Example

```
class Person:  
    pass
```

Python Inheritance

Python Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

Example

Create a class named `Person`, with `firstname` and `lastname` properties, and a `printname` method:

```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
```

#Use the Person class to create an object, and then execute the printname method:

```
x = Person("John", "Doe")
x.printname()
```

op: John Doe

Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

Example

Create a class named **Student**, which will inherit the properties and methods from the **Person** class:

```
class Student(Person):
    pass
```

Note: Use the `pass` keyword when you do not want to add any other properties or methods to the class.

Now the Student class has the same properties and methods as the Person class.

```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
```

```
class Student(Person):
    pass
```

```
x = Student("Mike", "Olsen")
x.printname()
```

op: Mike Olsen

Add the `__init__()` Function

So far we have created a child class that inherits the properties and methods from its parent.

We want to add the `__init__()` function to the child class (instead of the `pass` keyword).

Note: The `__init__()` function is called automatically every time the class is being used to create a new object.

Example

Add the `__init__()` function to the `Student` class:

```
class Student(Person):
    def __init__(self, fname, lname):
        #add properties etc.
```

When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function.

Note: The child's `__init__()` function **overrides** the inheritance of the parent's `__init__()` function.

To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function:

```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
```

```
class Student(Person):
    def __init__(self, fname, lname):
        Person.__init__(self, fname, lname)
```

```
x = Student("Mike", "Olsen")
y = Person("Han", "Solo")
x.printname()
y.printname()
```

```
op:    Mike Olsen
       Han Solo
```

Now we have successfully added the `__init__()` function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function.

Use the super() Function

Python also has a `super()` function that will make the child class inherit all the methods and properties from its parent:

```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)

class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname) # We didn't use the Person class "__init__()" function.

x = Student("Mike", "Olsen")
y = Person("Han", "Solo")
x.printname()
y.printname()

op:    Mike Olsen
       Han Solo
```

Add Properties

```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)

class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname)
        self.graduationyear = 2023

x = Student("Han", "Solo")
x.printname()
print(x.graduationyear)
```


op: Han Solo
2023

In the example below, the year 2019 should be a variable, and passed into the `Student` class when creating student objects. To do so, add another parameter in the `__init__()` function:

Example

Add a `year` parameter, and pass the correct year when creating objects:

```
class Student(Person):  
    def __init__(self, fname, lname, year):  
        super().__init__(fname, lname)  
        self.graduationyear = year
```

```
x = Student("Han", "Solo", 2023)
```

Add Methods

```
class Person:  
    def __init__(self, fname, lname):  
        self.firstname = fname  
        self.lastname = lname
```

```
    def printname(self):  
        print(self.firstname, self.lastname)
```

```
class Student(Person):  
    def __init__(self, fname, lname, year):  
        super().__init__(fname, lname)  
        self.graduationyear = year
```

```
    def welcome(self):  
        print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```

```
x = Student("Mike", "Olsen", 2019)  
x.welcome()
```

op: Welcome Mike Olsen to the class of 2019

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

Class and Instance Variables

Instance variables are for data, unique to each instance and class variables are for attributes and methods shared by all instances of the class. Instance variables are variables whose value is assigned inside a constructor or method with self whereas class variables are variables whose value is assigned in the class.

Defining instance variables using a constructor.

Program to show that the variables with a value assigned in the class declaration, are class variables and variables inside methods and constructors are instance variables.

```
class Dog:
```

```
    # Class Variable
```

```
    animal = 'dog'
```

```
    # The init method or constructor
```

```
    def __init__(self, breed, color):
```

```
        # Instance Variable
```

```
        self.breed = breed
```

```
        self.color = color
```

```
# Objects of Dog class
```

```
Rodger = Dog("Pug", "brown")
```

```
Buzo = Dog("Bulldog", "black")
```

```
print('Rodger details:')
```

```
print('Rodger is a', Rodger.animal)
```

```
print('Breed: ', Rodger.breed)
```

```
print('Color: ', Rodger.color)
```

```
print("\nBuzo details:')
```

```
print('Buzo is a', Buzo.animal)
```

```
print('Breed: ', Buzo.breed)
```

```
print('Color: ', Buzo.color)
```

```
# Class variables can be accessed using class name also
```

```
print("\nAccessing class variable using class name")
```

```
print(Dog.animal)
```

Output

```
Rodger details:  
Rodger is a dog  
Breed:  Pug  
Color:  brown
```

```
Buzo details:  
Buzo is a dog  
Breed:  Bulldog  
Color:  black
```

```
Accessing class variable using class name  
dog
```

Destructors in Python

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much as in C++ because Python has a garbage collector that handles memory management automatically. The `__del__()` method is known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

Syntax of destructor declaration :

```
def __del__(self):  
    # body of destructor
```

Python program to illustrate destructor

```
class Employee:
```

```
    # Initializing
```

```
    def __init__(self):  
        print('Employee created.')
```

```
    # Deleting (Calling destructor)
```

```
    def __del__(self):  
        print('Destructor called, Employee deleted.')
```

```
obj = Employee()  
del obj
```

Output

```
Employee created.  
Destructor called, Employee deleted.
```

Python Polymorphism

The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

Function Polymorphism

An example of a Python function that can be used on different objects is the `len()` function.

String

For strings `len()` returns the number of characters:

Example

```
x = "Hello World!"
```

```
print(len(x))
```

Tuple

For tuples `len()` returns the number of items in the tuple:

Example

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

```
print(len(mytuple))
```

Dictionary

For dictionaries `len()` returns the number of key/value pairs in the dictionary:

Example

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

```
print(len(thisdict))
```

Class Polymorphism

Polymorphism is often used in Class methods, where we can have multiple classes with the same method name.

For example, say we have three classes: `Car`, `Boat`, and `Plane`, and they all have a method called `move()`:

Example

Different classes with the same method:

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def move(self):
        print("Drive!")

class Boat:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def move(self):
        print("Sail!")

class Plane:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

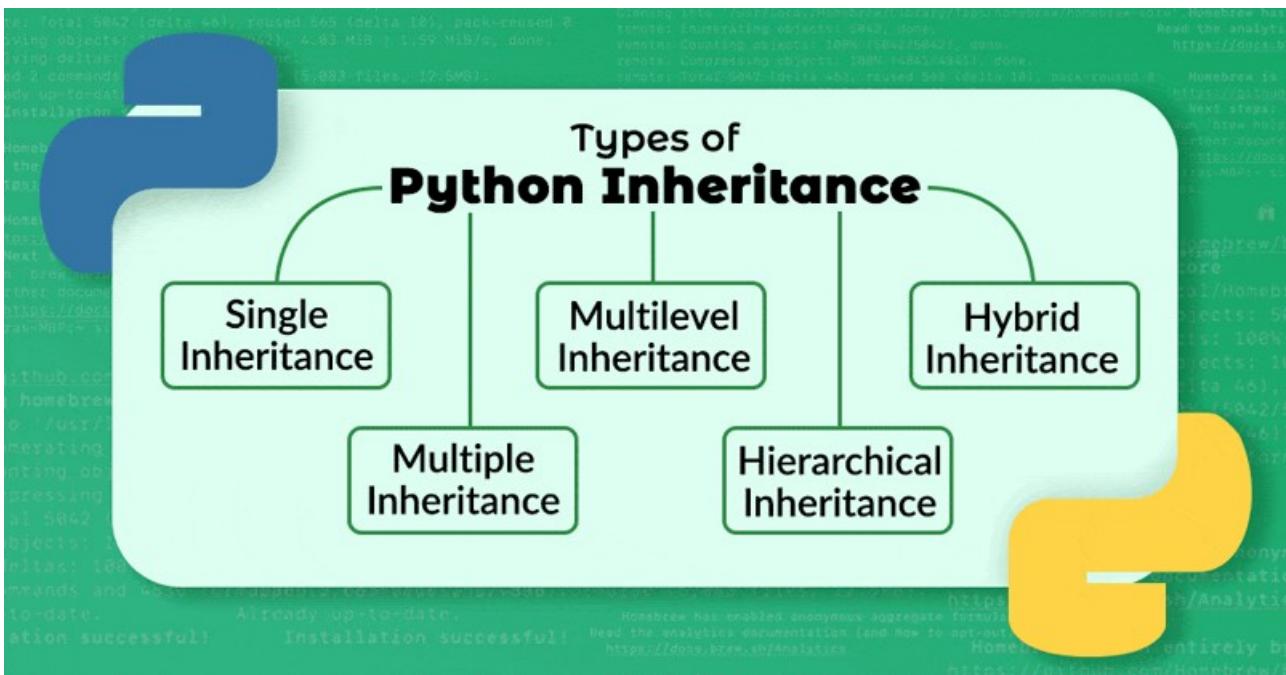
    def move(self):
        print("Fly!")

car1 = Car("Ford", "Mustang") #Create a Car class
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat class
plane1 = Plane("Boeing", "747") #Create a Plane class

for x in (car1, boat1, plane1):
    x.move()

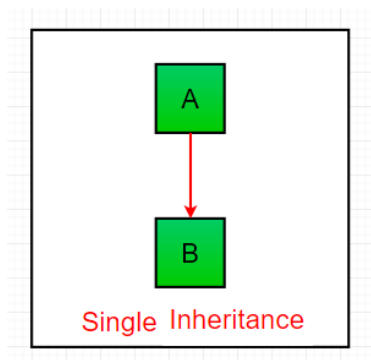
op:    Drive!
       Sail!
       Fly!
```

Look at the for loop at the end. Because of polymorphism we can execute the same method for all three classes.



Single Inheritance:

Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.



Python program to demonstrate
single inheritance

```
# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")
```

Derived class

```
class Child(Parent):
    def func2(self):
        print("This function is in child class.")
```

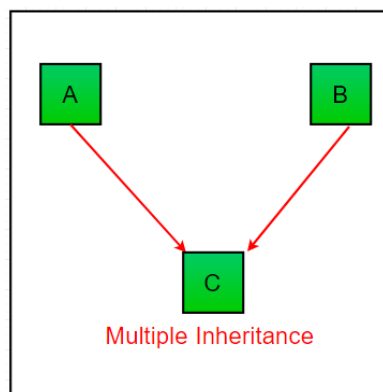
```
# Driver's code
object = Child()
object.func1()
object.func2()
```

Output:

```
This function is in parent class.
This function is in child class.
```

Multiple Inheritance:

When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class.



Python program to demonstrate multiple inheritance

Base class1

```
class Mother:
    mothername = ""
    def mother(self):
        print(self.mothername)
```

Base class2

```
class Father:
    fathername = ""
    def father(self):
        print(self.fathername)
```

Derived class

```

class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)

# Driver's code

s1 = Son()

s1.fathername = "RAM"

s1.mothername = "SITA"

s1.parents()

```

Output:

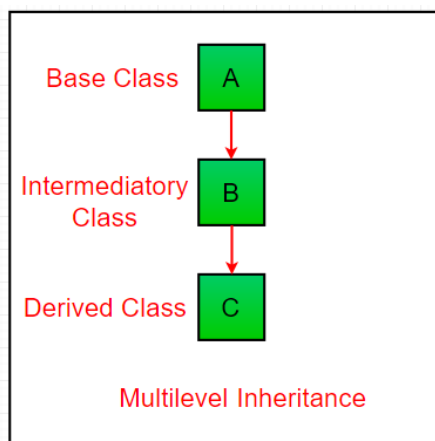
```

Father : RAM
Mother : SITA

```

Multilevel Inheritance:

In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and a grandfather.



```

# Python program to demonstrate multilevel inheritance

# Base class
class Grandfather:
    def __init__(self, grandfathername):
        self.grandfathername = grandfathername

# Intermediate class
class Father(Grandfather):
    def __init__(self, fathername, grandfathername):
        self.fathername = fathername

```



```

        # invoking constructor of Grandfather class
        Grandfather.__init__(self, grandfathername)

# Derived class
class Son(Father):
    def __init__(self, sonname, fathername, grandfathername):
        self.sonname = sonname

        # invoking constructor of Father class
        Father.__init__(self, fathername, grandfathername)

    def print_name(self):
        print('Grandfather name :', self.grandfathername)
        print("Father name :", self.fathername)
        print("Son name :", self.sonname)

# Driver code
s1 = Son('Ratan', 'Naval', 'Jamsetji')
print(s1.grandfathername)
s1.print_name()

```

Output:

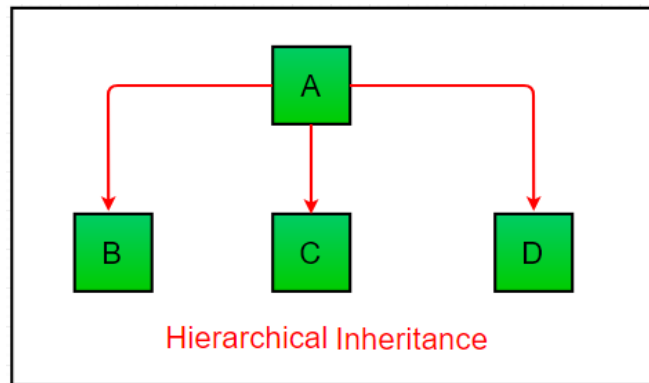
```

Jamsetji
Grandfather name : Jamsetji
Father name : Naval
Son name : Ratan

```

Hierarchical Inheritance:

When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.



Python program to demonstrate Hierarchical inheritance

Base class

```
class Parent:
    def func1(self):
        print("This function is in parent class.")
```

Derived class1

```
class Child1(Parent):
    def func2(self):
        print("This function is in child 1.")
```

Derived class2

```
class Child2(Parent):
    def func3(self):
        print("This function is in child 2.")
```

Driver's code

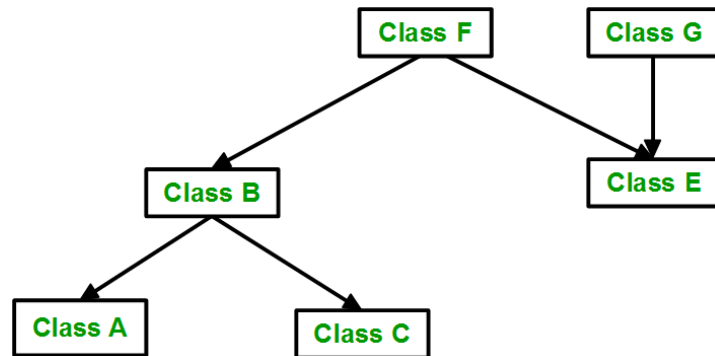
```
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

Output:

```
This function is in parent class.
This function is in child 1.
This function is in parent class.
This function is in child 2.
```

Hybrid Inheritance:

Inheritance consisting of multiple types of inheritance is called hybrid inheritance.



Python program to demonstrate hybrid inheritance

```
class School:
```

```
    def func1(self):
```

```
        print("This function is in school.")
```

```
class Student1(School):
```

```
    def func2(self):
```

```
        print("This function is in student 1. ")
```

```
class Student2(School):
```

```
    def func3(self):
```

```
        print("This function is in student 2.")
```

```
class Student3(Student1, School):
```

```
    def func4(self):
```

```
        print("This function is in student 3.")
```

```
# Driver's code
```

```
object = Student3()
```

```
object.func1()
```

```
object.func2()
```

Output:

```
This function is in school.
```

```
This function is in student 1.
```

Encapsulation in Python

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as **private variables**.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc. The goal of information hiding is to ensure that an object's state is always valid by controlling access to attributes that are hidden from the outside world.

Protected members

Protected members (in C++ and JAVA) are those members of the class that cannot be accessed outside the class but can be accessed from within the class and its subclasses. To accomplish this in Python, just follow **the convention** by prefixing the name of the member by a **single underscore** “_”.

```
# Python program to demonstrate protected members
```

```
# Creating a base class
```

```
class Base:
```

```
    def __init__(self):
```

```
        # Protected member
```

```
        self._a = 2
```

```
# Creating a derived class
```

```
class Derived(Base):
```

```
    def __init__(self):
```

```
        # Calling constructor of Base class
```

```
        Base.__init__(self)
```

```
        print("Calling protected member of base class: ", self._a)
```

```
        # Modify the protected variable:
```

```
        self._a = 3
```

```
        print("Calling modified protected member outside class: ", self._a)
```

```
obj1 = Derived()
```

```
obj2 = Base()
```

```
# Calling protected member
```

```
# Can be accessed but should not be done due to convention
```

```
print("Accessing protected member of obj1: ", obj1._a)
```

```
# Accessing the protected variable outside
```

```
print("Accessing protected member of obj2: ", obj2._a)
```

Output:

```
Calling protected member of base class:  2
Calling modified protected member outside class:  3
Accessing protected member of obj1:  3
Accessing protected member of obj2:  2
```

Class or Static Variables in Python

All objects share class or static variables. An instance or non-static variables are different for different objects (every object has a copy).

Explanation:

In Python, a static variable is a variable that is shared among all instances of a class, rather than being unique to each instance. It is also sometimes referred to as a class variable, because it belongs to the class itself rather than any particular instance of the class.

Static variables are defined inside the class definition, but outside of any method definitions. They are typically initialized with a value, just like an instance variable, but they can be accessed and modified through the class itself, rather than through an instance.

Features of Static Variables

- Static variables are allocated memory once when the object for the class is created for the first time.
- Static variables are created outside of methods but inside a class
- Static variables can be accessed through a class but not directly with an instance.
- Static variables behaviour doesn't change for every object.

Python program to show that the variables with a value assigned in class declaration, are class variables

Class for Computer Science Student

class CSSStudent:

```
    stream = 'cse'                # Class Variable

    def __init__(self,name,roll):

        self.name = name          # Instance Variable

        self.roll = roll          # Instance Variable
```

```
# Objects of CSSStudent class
```

```
a = CSSStudent('Harry', 1)
```

```
b = CSSStudent('Ron', 2)
```

```
print(a.stream) # prints "cse"
```

```
print(b.stream) # prints "cse"
```

```
print(a.name) # prints "Harry"
```

```
print(b.name) # prints "Ron"
```

```
print(a.roll) # prints "1"
```

```
print(b.roll) # prints "2"
```

```
# Class variables can be accessed using class name also
```

```
print(CSSStudent.stream) # prints "cse"
```

```
# Now if we change the stream for just a it won't be changed for b
```

```
a.stream = 'ece'
```

```
print(a.stream) # prints 'ece'
```

```
print(b.stream) # prints 'cse'
```

```
# To change the stream for all instances of the class we can change it directly from the class
```

```
CSSStudent.stream = 'mech'
```

```
print(a.stream) # prints 'ece'
```

```
print(b.stream) # prints 'mech'
```

Output:

```
cse
cse
Harry
Ron
1
2
cse
ece
cse
ece
mech
```

What is Class Method in Python?

The `@classmethod` decorator is a built-in [function decorator](#) that is an expression that gets evaluated after your function is defined. The result of that evaluation shadows your function definition. A [class method](#) receives the class as an implicit first argument, just like an instance method receives the instance

What is the Static Method in Python?

A [static method](#) does not receive an implicit first argument. A static method is also a method that is bound to the class and not the object of the class. This method can't access or modify the class state. It is present in a class because it makes sense for the method to be present in class.

Class method vs Static Method

The difference between the Class method and the static method is:

- A class method takes `cls` as the first parameter while a static method needs no specific parameters.
- A class method can access or modify the class state while a static method can't access or modify it.
- In general, static methods know nothing about the class state. They are utility-type methods that take some parameters and work upon those parameters. On the other hand class methods must have class as a parameter.
- We use `@classmethod` decorator in python to create a class method and we use `@staticmethod` decorator to create a static method in python.

```
class MyClass:
    def __init__(self, value):
        self.value = value

    @staticmethod
    def get_max_value(x, y):
        return max(x, y)

# Create an instance of MyClass
obj = MyClass(10)

print(MyClass.get_max_value(20, 30))

print(obj.get_max_value(20, 30))
```

OP: 30
 30

```

# Python program to demonstrate
# use of class method and static method.

from datetime import date

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # a class method to create a Person object by birth year.
    @classmethod
    def fromBirthYear(cls, name, year):
        return cls(name, date.today().year - year)

    # a static method to check if a Person is adult or not.
    @staticmethod
    def isAdult(age):
        return age > 18

person1 = Person('mayank', 21)
person2 = Person.fromBirthYear('mayank', 1996)

print(person1.age)
print(person2.age)

# print the result
print(Person.isAdult(22))

op:      21
        27
        True

```

Operator Overloading in Python

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because '+' operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called *Operator Overloading*.

```

# Python program to show use of
# + operator for different purposes.

print(1 + 2)

# concatenate two strings

```



```
print("Harry"+"Potter")
```

```
# Product two numbers  
print(3 * 4)
```

```
# Repeat the String  
print("John"*4)
```

To perform operator overloading, Python provides some special function or magic function that is automatically invoked when it is associated with that particular operator. For example, when we use + operator, the magic method `__add__` is automatically invoked in which the operation for + operator is defined.

```
# Python Program illustrate how  
# to overload an binary + operator  
# And how it actually works
```

```
class A:  
    def __init__(self, a):  
        self.a = a  
  
    # adding two objects  
    def __add__(self, o):  
        return self.a + o.a
```

```
ob1 = A(1)  
ob2 = A(2)  
ob3 = A("Operator")  
ob4 = A("Overloaded")
```

```
print(ob1 + ob2)  
print(ob3 + ob4)
```

```
# Actual working when Binary Operator is used.
```

```
print(A.__add__(ob1, ob2))  
print(A.__add__(ob3, ob4))
```

```
#And can also be Understand as :
```

```
print(ob1.__add__(ob2))  
print(ob3.__add__(ob4))
```

Output

```
3  
OperatorOverloaded  
3  
OperatorOverloaded  
3  
OperatorOverloaded
```

Here, We defined the special function “`__add__()`” and when the objects *ob1 and ob2* are coded as “**ob1 + ob2**“, the special function is automatically called as **ob1.__add__(ob2)** which simply means that ob1 calls the `__add__()` function with ob2 as an Argument and It actually means

A.__add__(ob1, ob2). Hence, when the Binary operator is overloaded, the object before the operator calls the respective function with object after operator as parameter.

*# Python Program to perform addition
of two complex numbers using binary
+ operator overloading.*

```
class complex:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    # adding two objects
    def __add__(self, other):
        return self.a + other.a, self.b + other.b
```

```
Ob1 = complex(1, 2)
Ob2 = complex(2, 3)
Ob3 = Ob1 + Ob2
print(Ob3)
```

Output
(3, 5)

*# Python program to overload
a comparison operators*

```
class A:
    def __init__(self, a):
        self.a = a
    def __gt__(self, other):
        if(self.a>other.a):
            return True
        else:
            return False

ob1 = A(2)
ob2 = A(3)
if(ob1>ob2):
    print("ob1 is greater than ob2")
else:
    print("ob2 is greater than ob1")
```

Output:

ob2 is greater than ob1

*# Python program to overload equality
and less than operators*

```
class A:
    def __init__(self, a):
        self.a = a
```

```

def __lt__(self, other):
    if(self.a<other.a):
        return "ob1 is lessthan ob2"
    else:
        return "ob2 is less than ob1"
def __eq__(self, other):
    if(self.a == other.a):
        return "Both are equal"
    else:
        return "Not equal"

```

```

ob1 = A(2)
ob2 = A(3)
print(ob1 < ob2)

```

```

ob3 = A(4)
ob4 = A(4)
print(ob1 == ob2)

```

Output:

```

ob1 is lessthan ob2
Not equal

```

Binary Operators

Operator	Magic Method
+	<code>__add__(self, other)</code>
-	<code>__sub__(self, other)</code>
*	<code>__mul__(self, other)</code>
/	<code>__truediv__(self, other)</code>
//	<code>__floordiv__(self, other)</code>
%	<code>__mod__(self, other)</code>
**	<code>__pow__(self, other)</code>
>>	<code>__rshift__(self, other)</code>
<<	<code>__lshift__(self, other)</code>
&	<code>__and__(self, other)</code>
	<code>__or__(self, other)</code>
^	<code>__xor__(self, other)</code>

Comparison Operators:

Operator	Magic Method
<	<code>__lt__(self, other)</code>
>	<code>__gt__(self, other)</code>
<=	<code>__le__(self, other)</code>
>=	<code>__ge__(self, other)</code>
==	<code>__eq__(self, other)</code>
!=	<code>__ne__(self, other)</code>

Assignment Operators:

Operator	Magic Method
<code>-=</code>	<code>__isub__(self, other)</code>
<code>+=</code>	<code>__iadd__(self, other)</code>
<code>*=</code>	<code>__imul__(self, other)</code>
<code>/=</code>	<code>__idiv__(self, other)</code>
<code>//=</code>	<code>__ifloordiv__(self, other)</code>
<code>%=</code>	<code>__imod__(self, other)</code>
<code>**=</code>	<code>__ipow__(self, other)</code>
<code>>>=</code>	<code>__irshift__(self, other)</code>
<code><<=</code>	<code>__ilshift__(self, other)</code>
<code>&=</code>	<code>__iand__(self, other)</code>
<code> =</code>	<code>__ior__(self, other)</code>
<code>^=</code>	<code>__ixor__(self, other)</code>

Method Overloading and Method Overriding in Python

Method Overloading:

Method Overloading is an example of Compile time polymorphism. In this, more than one method of the same class shares the same method name having different signatures. Method overloading is used to add more to the behavior of methods and there is no need of more than one class for method overloading.

Note: Python does not support method overloading. We may overload the methods but can only use the latest defined method.

Function to take multiple arguments

```
def add(datatype, *args):

    # if datatype is int
    # initialize answer as 0
    if datatype == 'int':
        answer = 0

    # if datatype is str
    # initialize answer as ""
    if datatype == 'str':
        answer = ""

    # Traverse through the arguments
    for x in args:

        # This will do addition if the
```

```
# arguments are int. Or concatenation
# if the arguments are str
answer = answer + x
```

```
print(answer)
```

```
# Integer
add('int', 5, 6)
```

```
# String
add('str', 'Hi ', 'Hello')
```

Output:

```
11
```

```
Hi Hello
```

Method Overriding:

Method overriding is an example of run time polymorphism. In this, the specific implementation of the method that is already provided by the parent class is provided by the child class. It is used to change the behavior of existing methods and there is a need for at least two classes for method overriding. In method overriding, inheritance always required as it is done between parent class(superclass) and child class(child class) methods.

```
class A:
```

```
    def fun1(self):
        print('feature_1 of class A')
```

```
    def fun2(self):
        print('feature_2 of class A')
```

```
class B(A):
```

```
    # Modified function that is
    # already exist in class A
    def fun1(self):
        print('Modified feature_1 of class A by class B')
```

```
    def fun3(self):
        print('feature_3 of class B')
```

```
# Create instance
obj = B()
```

```
# Call the override function
obj.fun1()
```

Output: Modified feature_1 of class A by class B

S.NO	Method Overloading	Method Overriding
1.	In the method overloading, methods or functions must have the same name and different signatures.	Whereas in the method overriding, methods or functions must have the same name and same signatures.
2.	Method overloading is a example of compile time polymorphism.	Whereas method overriding is a example of run time polymorphism.
3.	In the method overloading, inheritance may or may not be required.	Whereas in method overriding, inheritance always required.
4.	Method overloading is performed between methods within the class.	Whereas method overriding is done between parent class and child class methods.
5.	It is used in order to add more to the behavior of methods.	Whereas it is used in order to change the behavior of exist methods.
6.	In method overloading, there is no need of more than one class.	Whereas in method overriding, there is need of at least of two classes.

Inheritance Class Polymorphism

What about classes with child classes with the same name? Can we use polymorphism there?

Yes. If we use the example above and make a parent class called `Vehicle`, and make `Car`, `Boat`, `Plane` child classes of `Vehicle`, the child classes inherits the `Vehicle` methods, but can override them:

Example

Create a class called `Vehicle` and make `Car`, `Boat`, `Plane` child classes of `Vehicle`:

```
class Vehicle:
```

```
    def __init__(self, brand, model):
```

```
        self.brand = brand
```

```
        self.model = model
```

```
    def move(self):
```

```
        print("Move!")
```

```
class Car(Vehicle):
```

```
    pass
```

```
class Boat(Vehicle):
```

```
    def move(self):
```

```
        print("Sail!")
```

```
class Plane(Vehicle):
```

```
    def move(self):
```

```
        print("Fly!")
```

```
car1 = Car("Ford", "Mustang") #Create a Car object
```

```
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
```

```
plane1 = Plane("Boeing", "747") #Create a Plane object
```

```
for x in (car1, boat1, plane1):  
    print(x.brand)  
    print(x.model)  
    x.move()
```

```
op:    Ford  
      Mustang  
      Move!  
      Ibiza  
      Touring 20  
      Sail!  
      Boeing  
      747  
      Fly!
```

Child classes inherits the properties and methods from the parent class.

In the example above you can see that the `Car` class is empty, but it inherits `brand`, `model`, and `move()` from `Vehicle`.

The `Boat` and `Plane` classes also inherit `brand`, `model`, and `move()` from `Vehicle`, but they both override the `move()` method.

Because of polymorphism we can execute the same method for all classes.

Python Exception Handling

Different types of exceptions in python:

In Python, there are several built-in exceptions that can be raised when an error occurs during the execution of a program. Here are some of the most common types of exceptions in Python:

***SyntaxError:** This exception is raised when the interpreter encounters a syntax error in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.

***TypeError:** This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.

***NameError:** This exception is raised when a variable or function name is not found in the current scope.

***IndexError:** This exception is raised when an index is out of range for a list, tuple, or other sequence types.

***KeyError:** This exception is raised when a key is not found in a dictionary.

***ValueError:** This exception is raised when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer when the string does not represent a valid integer.

***AttributeError:** This exception is raised when an attribute or method is not found on an object, such as trying to access a non-existent attribute of a class instance.

***IOError:** This exception is raised when an I/O operation, such as reading or writing a file, fails due to an input/output error.

***ZeroDivisionError:** This exception is raised when an attempt is made to divide a number by zero.

***ImportError:** This exception is raised when an import statement fails to find or load a module.

Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the `try` statement:

Example

The `try` block will generate an exception, because `x` is not defined:

```
try:
    print(x)
except:
    print("An exception occurred")
```

op: An exception occurred

Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

Python program to handle simple runtime error

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

```
try:
```

```
    print ("Second element = %d" %(a[1]))
```

```
    # Throws error since there are only 3 elements in array
```

```
    print ("Fourth element = %d" %(a[3]))
```

```
except:
```

```
    print ("An error occurred")
```

Output

```
Second element = 2
```


An error occurred

Catching Specific Exception

A try statement can have more than one except clause, to specify handlers for different exceptions. Please note that at most one handler will be executed.

Program to handle multiple errors with one except statement

```
def fun(a):  
    if a < 4:  
        # throws ZeroDivisionError for a = 3  
        b = a/(a-3)  
    # throws NameError if a >= 4  
    print("Value of b = ", b)  
  
try:  
    fun(3)  
    fun(5)  
  
# note that braces () are necessary here for multiple exceptions  
  
except ZeroDivisionError:  
    print("ZeroDivisionError Occurred and Handled")  
  
except NameError:  
    print("NameError Occurred and Handled")
```

Try with Else Clause

In Python, you can also use the else clause on the try-except block which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.

Example: Try with else clause

Program to depict else clause with try-except

Function which returns a/b

```
def AbyB(a , b):  
    try:  
        c = ((a+b) / (a-b))
```

```
except ZeroDivisionError:
    print ("a/b result in 0")
else:
    print (c)
```

AbyB(2.0, 3.0)

AbyB(3.0, 3.0)

Output:

```
-5.0
a/b result in 0
```

Finally Keyword in Python

Python provides a keyword [finally](#), which is always executed after the try and except blocks. The final block always executes after the normal termination of the try block or after the try block terminates due to some exception.

Syntax:

```
try:
    # Some Code....

except:
    # optional block
    # Handling of exception (if required)

else:
    # execute if no exception

finally:
    # Some code .....(always executed)
```

Python program to demonstrate finally

No exception Exception raised in try block

```
try:
    k = 5//0 # raises divide by zero exception.
    print(k)
```

handles zerodivision exception

```
except ZeroDivisionError:
    print("Can't divide by zero")
```

finally:

```
# this block is always executed  
# regardless of exception generation.  
print('This is always executed')
```

Output:

```
Can't divide by zero  
This is always executed
```

Python Modules

What is a Module?

Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

Create a Module

To create a module just save the code you want in a file with the file extension `.py`:

Example

Save this code in a file named `mymodule.py`

```
def greeting(name):  
    print("Hello, " + name)
```

Use a Module

Now we can use the module we just created, by using the `import` statement:

Example

Import the module named `mymodule`, and call the `greeting` function:

```
import mymodule
```

```
mymodule.greeting("Jonathan")
```

```
op: Hello, Jonathan
```

Note: When using a function from a module, use the syntax: `module_name.function_name`.

Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Example

Save this code in the file `mymodule.py`

```
person1 = {  
    "name": "John",  
    "age": 36,  
    "country": "Norway"  
}
```

Example

Import the module named `mymodule`, and access the `person1` dictionary:

```
import mymodule
```

```
a = mymodule.person1["age"]  
print(a)
```

Naming a Module

You can name the module file whatever you like, but it must have the file extension `.py`

Re-naming a Module

You can create an alias when you import a module, by using the `as` keyword:

Example

Create an alias for `mymodule` called `mx`:

```
import mymodule as mx
```

```
a = mx.person1["age"]  
print(a)
```

Built-in Modules

There are several built-in modules in Python, which you can import whenever you like.

Example

Import and use the `platform` module:

```
import platform
```

```
x = platform.system()  
print(x)
```

op: Windows

os module

random module
math module
time module
sys module
collections module
statistics module....etc

Using the dir() Function

There is a built-in function to list all the function names (or variable names) in a module. The `dir()` function:

Example

List all the defined names belonging to the platform module:

```
import platform

x = dir(platform)
print(x)
```

Note: The `dir()` function can be used on *all* modules, also the ones you create yourself.

Import From Module

You can choose to import only parts from a module, by using the `from` keyword.

Example

The module named `mymodule` has one function and one dictionary:

```
def greeting(name):
    print("Hello, " + name)

person1 = {
    "name": "John",
    "age": 36,
    "country": "Norway"
}
```

Example

Import only the `person1` dictionary from the module:

```
from mymodule import person1

print(person1["age"])
```

Note: When importing using the `from` keyword, do not use the module name when referring to elements in the module. Example: `person1["age"]`, **not** `mymodule.person1["age"]`

Python File Open

File handling is an important part of any web application.

Python has several functions for creating, reading, updating, and deleting files.

File Handling

The key function for working with files in Python is the `open()` function.

The `open()` function takes two parameters; *filename*, and *mode*.

There are four different methods (modes) for opening a file:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

The code above is the same as:

```
f = open("demofile.txt", "rt")
```

Because "r" for read, and "t" for text are the default values, you do not need to specify them.

Note: Make sure the file exists, or else you will get an error.

Python File Open

Open a File on the Server

Assume we have the following file, located in the same folder as Python:

demofile.txt

Hello! Welcome to demofile.txt

This file is for testing purposes.

Good Luck!

To open the file, use the built-in `open()` function.

The `open()` function returns a file object, which has a `read()` method for reading the content of the file:

Example

```
f = open("demofile.txt", "r")  
print(f.read())
```

If the file is located in a different location, you will have to specify the file path, like this:

Example

Open a file on a different location:

```
f = open("D:\\myfiles\\welcome.txt", "r")  
print(f.read())
```

Read Only Parts of the File

By default the `read()` method returns the whole text, but you can also specify how many characters you want to return:

Example

Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")  
print(f.read(5))
```

Python File Write

Write to an Existing File

To write to an existing file, you must add a parameter to the `open()` function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Example

Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")  
f.write("Now the file has more content!")  
f.close()
```

#open and read the file after the appending:

```
f = open("demofile2.txt", "r")  
print(f.read())
```

Example

Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")  
f.write("Woops! I have deleted the content!")
```

```
f.close()
```

#open and read the file after the overwriting:

```
f = open("demofile3.txt", "r")
```

```
print(f.read())
```

Note: the "w" method will overwrite the entire file.

Create a New File

To create a new file in Python, use the `open()` method, with one of the following parameters:

"x" - Create - will create a file, returns an error if the file exist

"a" - Append - will create a file if the specified file does not exist

"w" - Write - will create a file if the specified file does not exist

Example

Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

Result: a new empty file is created!

Example

Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

Python Delete File

Delete a File

To delete a file, you must import the OS module, and run its `os.remove()` function:

Example

Remove the file "demofile.txt":

```
import os
```

```
os.remove("demofile.txt")
```

Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

Example

Check if file exists, *then* delete it:

```
import os
```

```
if os.path.exists("demofile.txt"):
```

```
    os.remove("demofile.txt")
```

```
else:
```

```
    print("The file does not exist")
```

Delete Folder

To delete an entire folder, use the `os.rmdir()` method:

Example

Remove the folder "myfolder":

```
import os  
os.rmdir("myfolder")
```

Note: You can only remove *empty* folders.

....Good Luck !!!!

