Python Intro

What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

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

What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

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

Good to know

- The most recent major version of Python is Python 3, which we shall be using in this tutorial. However, Python 2, although not being updated with anything other than security updates, is still quite popular.
- In this tutorial Python will be written in a text editor. It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

Python Syntax compared to other programming languages

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

Python Install

Many PCs and Macs will have python already installed.

To check if you have python installed on a Windows PC, search in the start bar for Python or run the following on the Command Line (cmd.exe)

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

To check if you have python installed on a Linux or Mac, then on linux open the command line or on Mac open the Terminal and type:

```
python --version
```

Python Quickstart

Python is an interpreted programming language, this means that as a developer you write Python (.py) files in a text editor and then put those files into the python interpreter to be executed.

The way to run a python file is like this on the command line:

```
C:\Users\Your Name>python helloworld.py
```

Where "helloworld.py" is the name of your python file.

Let's write our first Python file, called helloworld.py, which can be done in any text editor.

helloworld.py

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

Simple as that. Save your file. Open your command line, navigate to the directory where you saved your file, and run:

```
C:\Users\Your Name>python helloworld.py
```

The output should read:

```
Hello, World!
```

The Python Command Line

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

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

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

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

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

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

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

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

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

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

```
exit()
```

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.

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

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

Syntax Error:

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

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

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

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

Syntax Error:

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

```
#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:

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

Multi Line Comments

Python does not really have a syntax for multi line comments.

To add a multiline comment you could insert a `#` for each line:

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

Or, not quite as intended, you can use a multiline string.

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:

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

Python Variables

Creating Variables

Variables are containers for storing data values.

Unlike other programming languages, Python has no command for declaring a variable.

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

```python
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.

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

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

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

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)

Assign Value to Multiple Variables

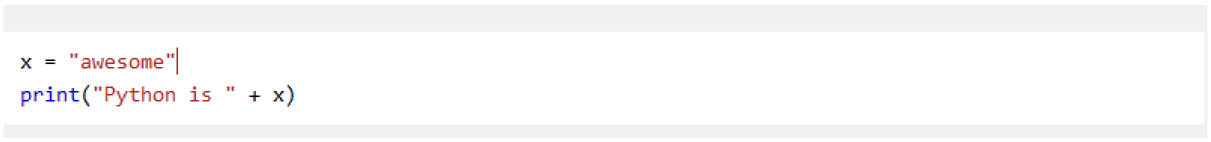
- Python allows you to assign values to multiple variables in one line:

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

Output Variables

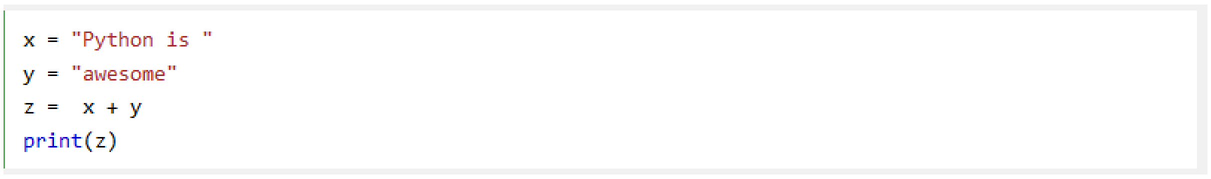
The Python `print` statement is often used to output variables.

To combine both text and a variable, Python uses the `+` character:



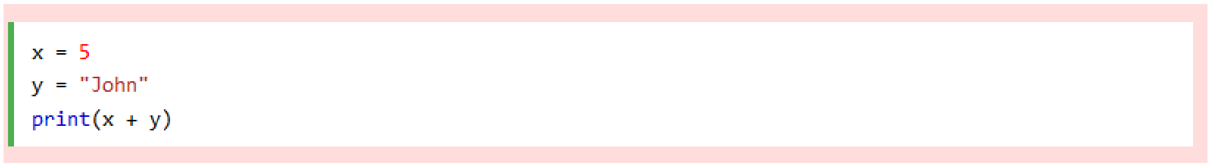
```python
x = "awesome"
print("Python is " + x)
```

You can also use the `+` character to add a variable to another variable:



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

If you try to combine a string and a number, Python will give you an error:



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

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`

Getting the Data Type

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

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

Setting the Data Type

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

`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

Setting the Specific Data Type

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

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

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:

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

To verify the type of any object in Python, use the `type()` function:

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

Int

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

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

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

Float

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

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

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

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

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

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

Complex

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

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

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

Type Conversion

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

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

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

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

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

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

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

Random Number

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

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

```
import random

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

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:

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

Evaluate Values and Variables

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

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.

The following will return True:

```
bool("abc")
bool(123)
bool(["apple", "cherry", "banana"])
```

Some Values are False

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

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

Functions can Return a Boolean

Python also has many built-in functions that returns a boolean value, like the `isinstance()` function, which can be used to determine if an object is of a certain data type:

Check if an object is an integer or not:

```
x = 200
print(isinstance(x, int))
```

Python Operators

Python Operators

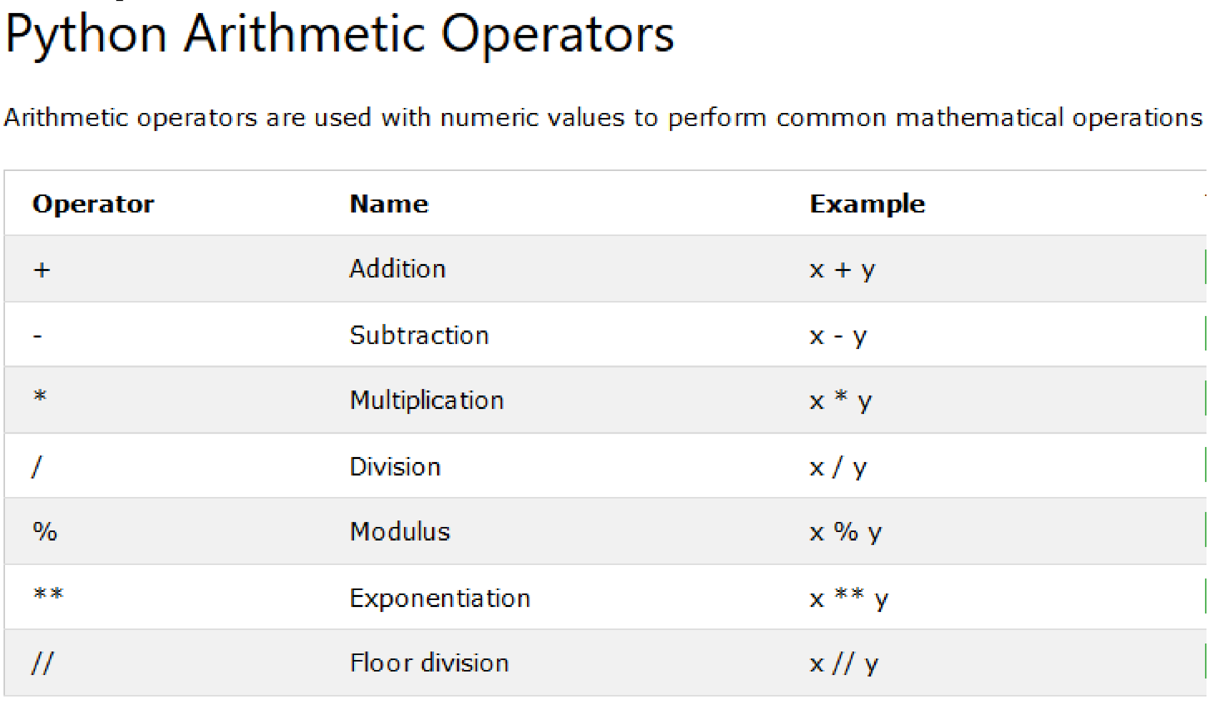
Operators are used to perform operations on variables and values.

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:



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	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Python Assignment Operators

Assignment operators are used to assign values to variables:

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

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

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

Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

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

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 rounding down to the previous whole number), 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

Integers:

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

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
```

Strings:

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

User Input

Python allows for user input.

That means we are able to ask the user for input.

The method is a bit different in Python 3.6 than Python 2.7.

Python 3.6 uses the `input()` method.

Python 2.7 uses the `raw_input()` method.

The following example asks for the username, and when you entered the username, it gets printed on the screen:

Python 3.6

```
username = input("Enter username:")
print("Username is: " + username)
```

Python 2.7

```
username = raw_input("Enter username:")
print("Username is: " + username)
```

String Literals

String literals 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:

```
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:

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

Multiline Strings

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

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)
```

Or three single 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.

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

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

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.

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

Negative Indexing

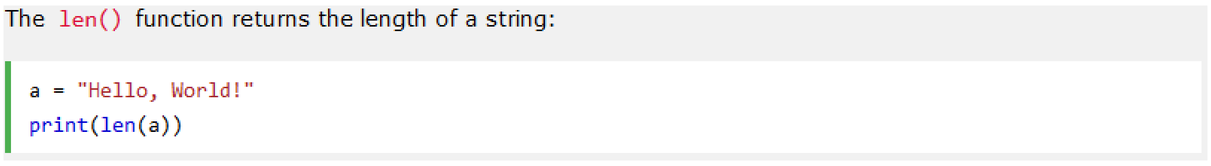
Use negative indexes to start the slice from the end of the string:

Get the characters from position 5 to position 1, starting the count from the end of the string:

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

String Length

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


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

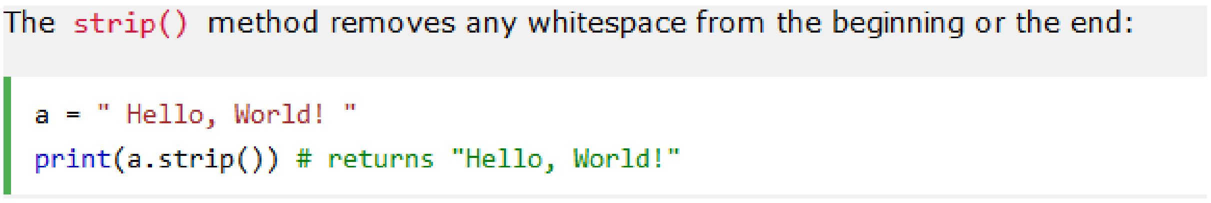
```
a = "Hello, World!"
print(len(a))
```

String Methods

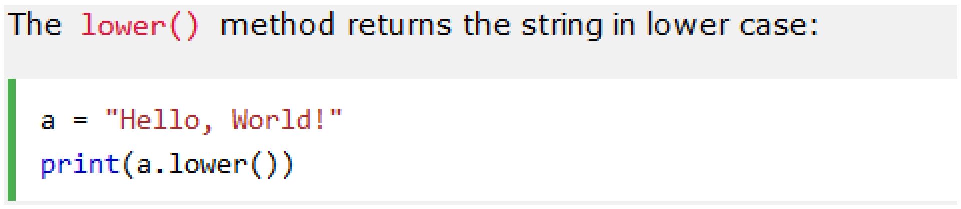
Python has a set of built-in methods that you can use on strings.

Example

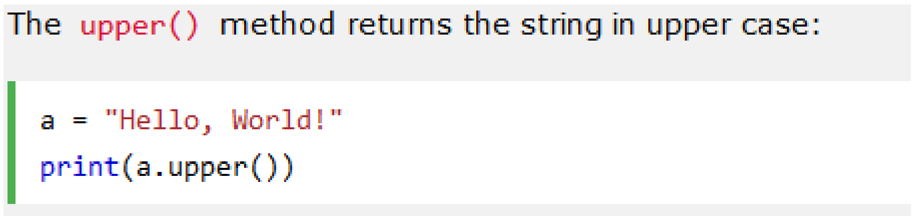
The `strip()` method removes any whitespace from the beginning or the end:


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

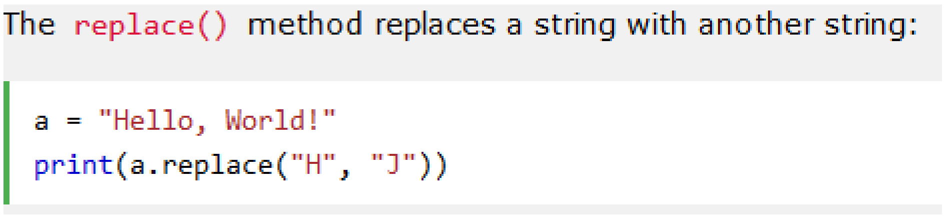
```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
```


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

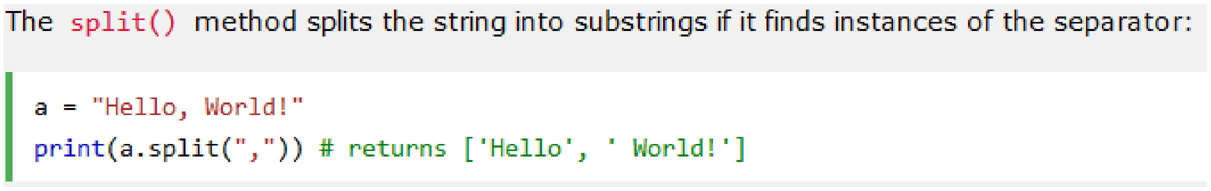
```
a = "Hello, World!"
print(a.lower())
```


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

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


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

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


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

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

Check String

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

Check if the phrase "ain" is present in the following text:

```
txt = "The rain in Spain stays mainly in the plain"
x = "ain" in txt
print(x)
```

Check if the phrase "ain" is NOT present in the following text:

```
txt = "The rain in Spain stays mainly in the plain"
x = "ain" not in txt
print(x)
```

String Concatenation

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

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

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

To add a space between them, add a `" "`:

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

String Format

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

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

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:

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

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

The format() method takes unlimited number of arguments, and are placed into the respective placeholders:

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

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

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

Escape 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.

An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

You will get an error if you use double quotes inside a string that is surrounded by double quotes:

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

To fix this problem, use the escape character `\"`:

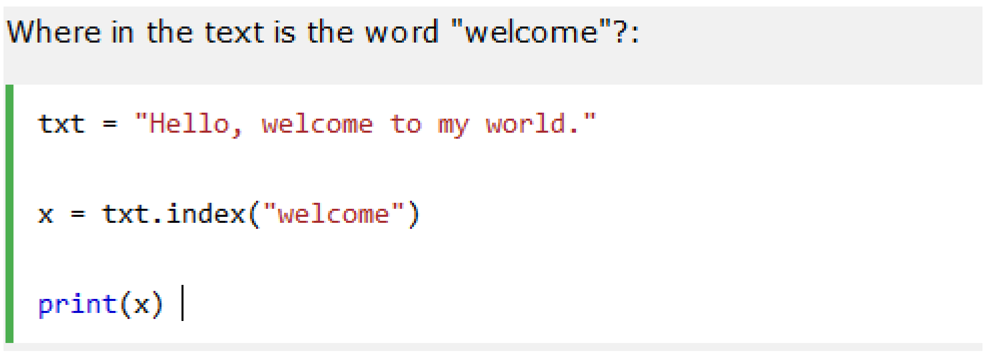
The escape character allows you to use double quotes when you normally would not be allowed:

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

Other escape characters used in Python:

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

Python String index() Method


Where in the text is the word "welcome"?:

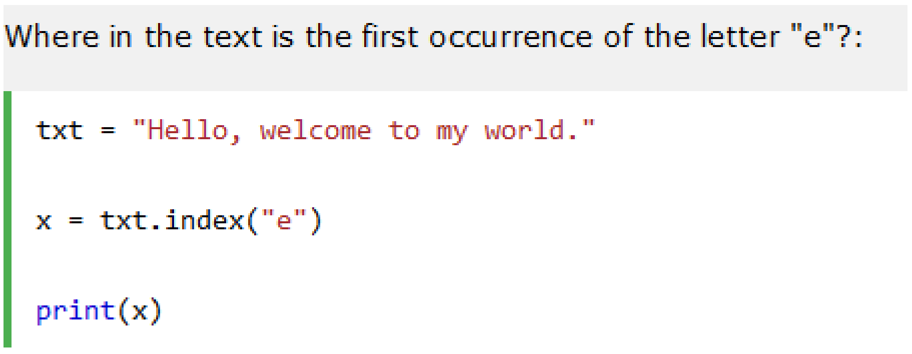
```python
txt = "Hello, welcome to my world."

x = txt.index("welcome")

print(x)
```

Definition and Usage

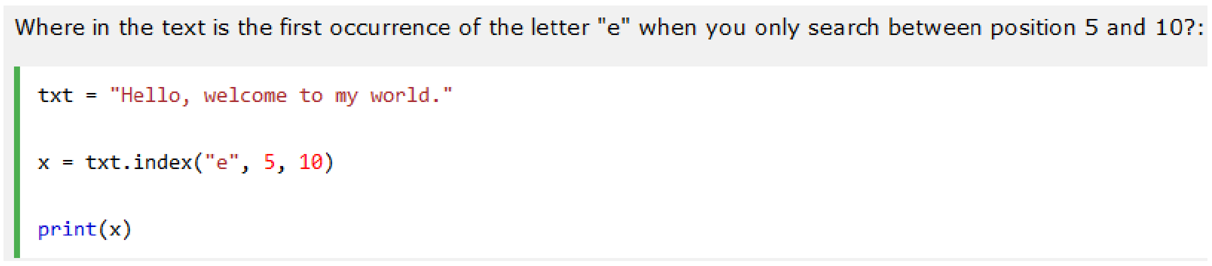
The `index()` method finds the first occurrence of the specified value.

The `index()` method raises an exception if the value is not found.

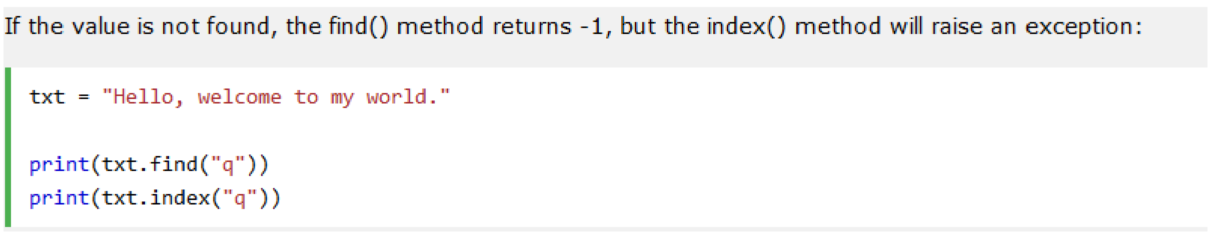
The `index()` method is almost the same as the `find()` method, the only difference is that the `find()` method returns -1 if the value is not found. (See example below)


Where in the text is the first occurrence of the letter "e"?:

```python
txt = "Hello, welcome to my world."

x = txt.index("e")

print(x)
```

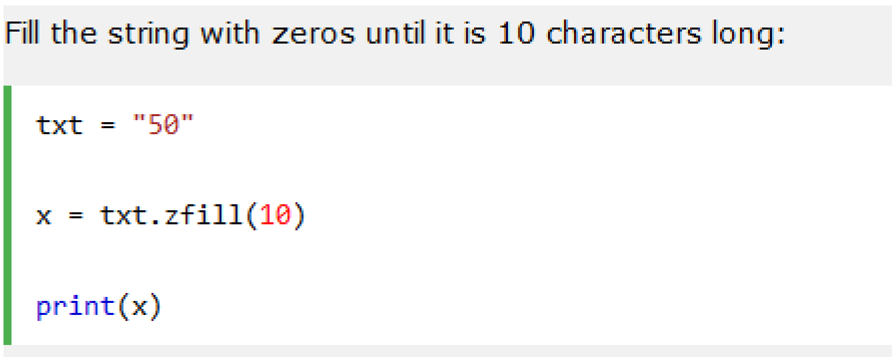

Where in the text is the first occurrence of the letter "e" when you only search between position 5 and 10?:

```python
txt = "Hello, welcome to my world."

x = txt.index("e", 5, 10)

print(x)
```


If the value is not found, the find() method returns -1, but the index() method will raise an exception:

```python
txt = "Hello, welcome to my world."

print(txt.find("q"))
print(txt.index("q"))
```

Python String zfill() Method



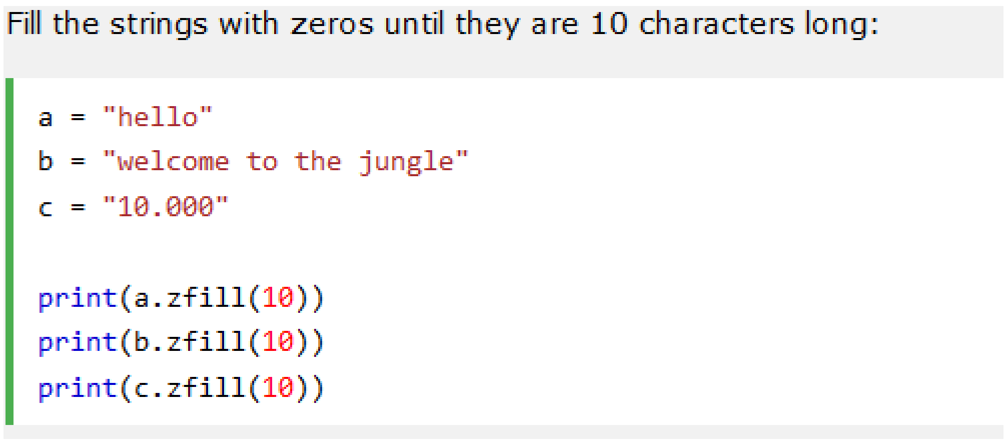
Fill the string with zeros until it is 10 characters long:

```
txt = "50"

x = txt.zfill(10)

print(x)
```

Definition and Usage

The `zfill()` method adds zeros (0) at the beginning of the string, until it reaches the specified length.

If the value of the len parameter is less than the length of the string, no filling is done.



Fill the strings with zeros until they are 10 characters long:

```
a = "hello"
b = "welcome to the jungle"
c = "10.000"

print(a.zfill(10))
print(b.zfill(10))
print(c.zfill(10))
```

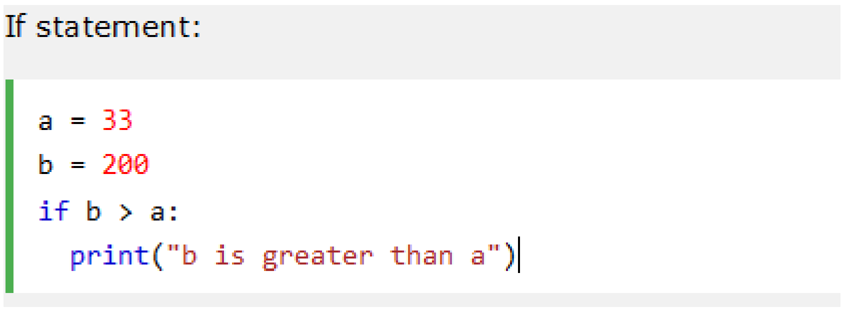
Python If ... Else

Python supports the usual logical conditions from mathematics:

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

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

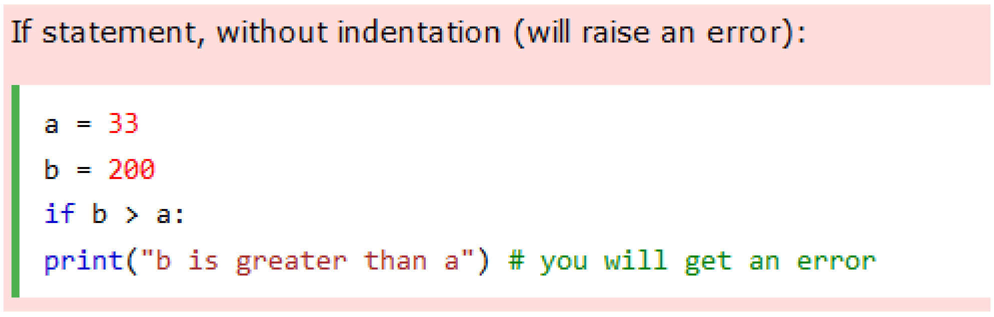
- An "if statement" is written by using the if keyword.



If statement:

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

In this example we use two variables, a and b, which are used as part of the if statement to test whether b is greater than a. As a is 33, and b is 200, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".



If statement, without indentation (will raise an error):

```
a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an error
```

Elif

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

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

In this example a is equal to b, so the first condition is not true, but the elif condition is true, so we print to screen that "a and b are equal".

Else

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

```
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")
```

In this example a is greater than b, so the first condition is not true, also the elif condition is not true, so we go to the else condition and print to screen that "a is greater than b".

You can also have an `else` without the `elif`:

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

Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

One line if statement:

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

Short Hand If ... Else

One line if else statement:

```
a = 2
b = 330
print("A") if a > b else print("B")
```

You can also have multiple else statements on the same line:

One line if else statement, with 3 conditions:

```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

And

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

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:

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")
```

Nested If

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

```
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.

```
a = 33
b = 200

if b > a:
  pass
```

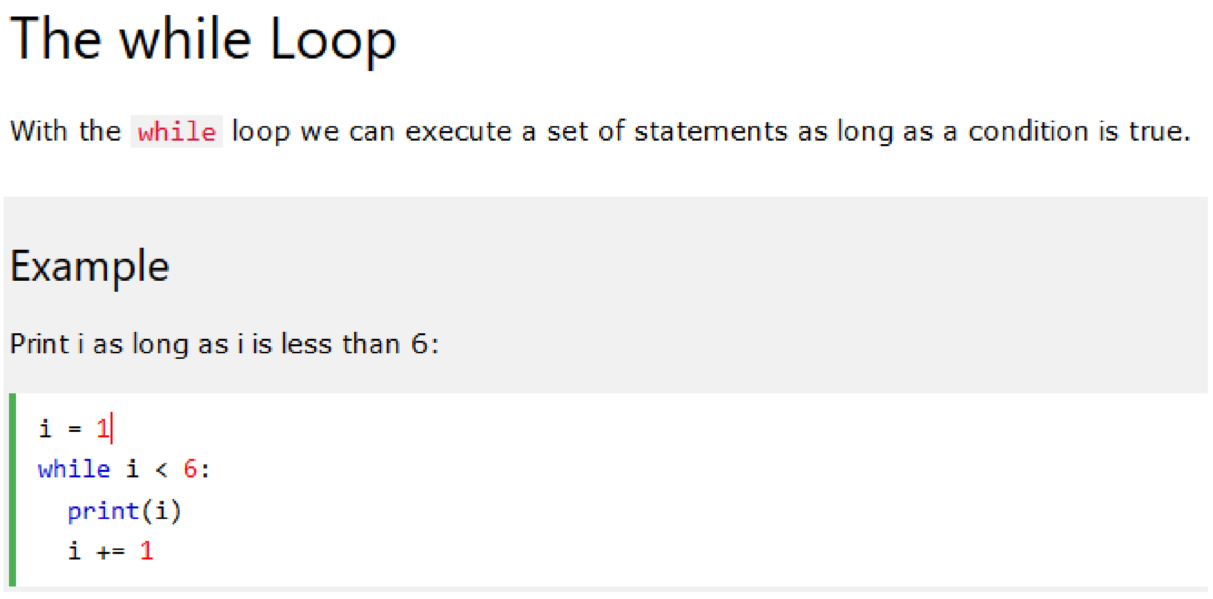
Python Loops

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.



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:

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:

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:

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

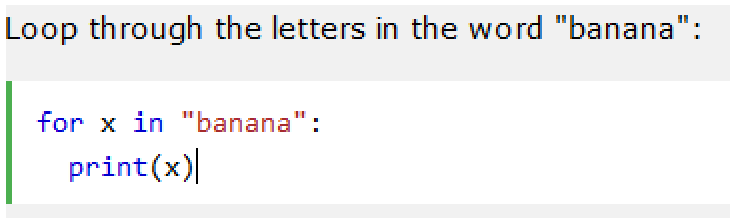
A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

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.

Looping Through a String

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

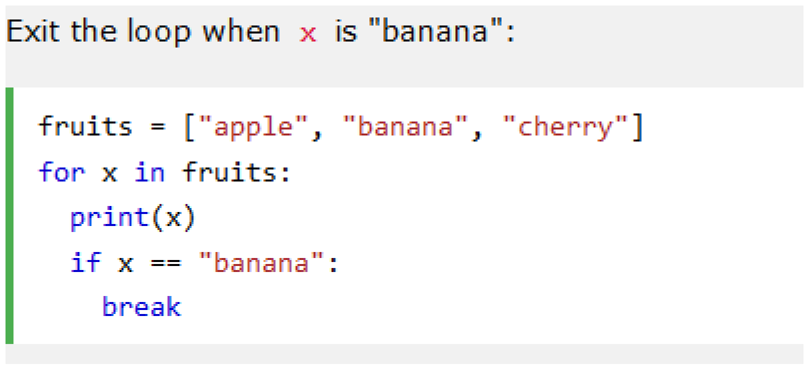


Loop through the letters in the word "banana":

```
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:



Exit the loop when x is "banana":

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

The continue Statement

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

Do not print banana:

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

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.

Using the range() function:

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

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):

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**):

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:

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":

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)
```

The pass Statement

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

```
for x in [0, 1, 2]:
  pass
```

Python Collections (Arrays)

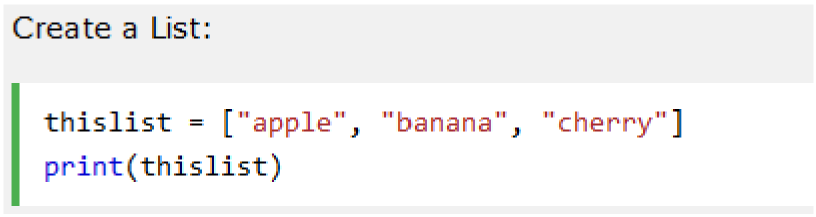
There are four collection data types in the Python programming language:

- **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 and unindexed. No duplicate members.
- **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate members.

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.

List

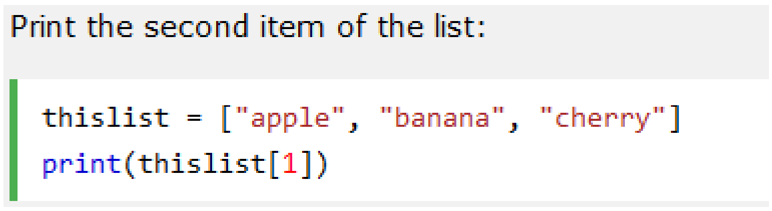
A list is a collection which is ordered and changeable. In Python lists are written with square brackets.


Create a List:

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

Access Items

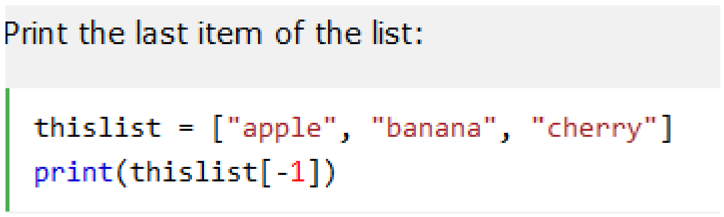
You access the list items by referring to the index number:


Print the second item of the list:

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

Negative Indexing

Negative indexing means beginning from the end, `-1` refers to the last item, `-2` refers to the second last item etc.

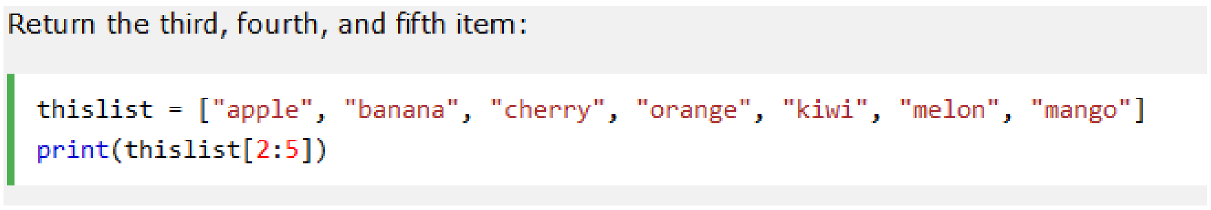

Print the last item of the list:

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

Range of Indexes

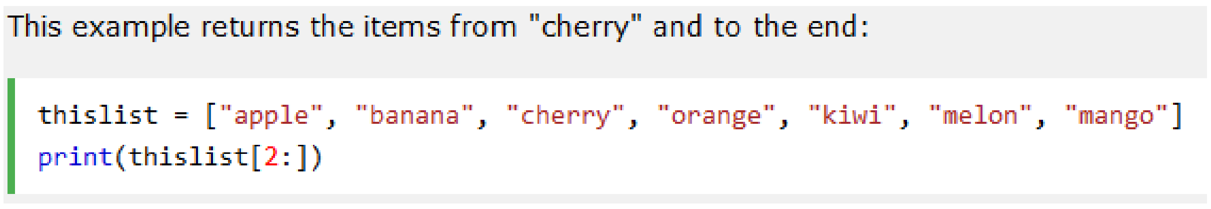
You can specify a range of indexes by specifying where to start and where to end the range.

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


Return the third, fourth, and fifth item:

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

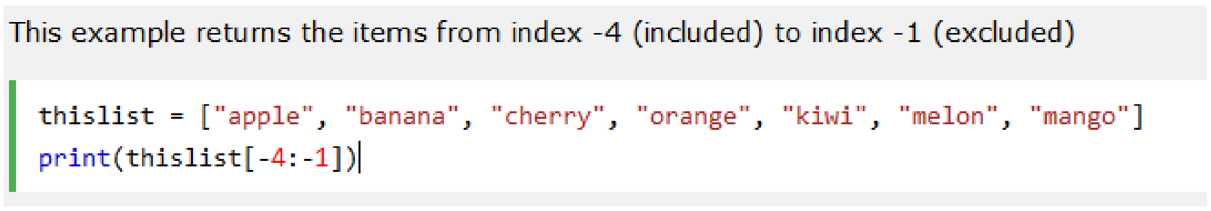
By leaving out the end value, the range will go on to the end of the list:


This example returns the items from "cherry" and to the end:

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

Range of Negative Indexes

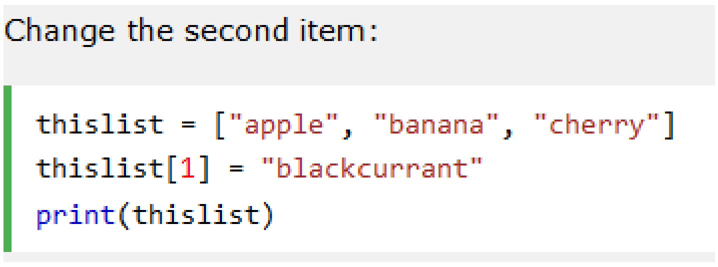
Specify negative indexes if you want to start the search from the end of the list:


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

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

Change Item Value

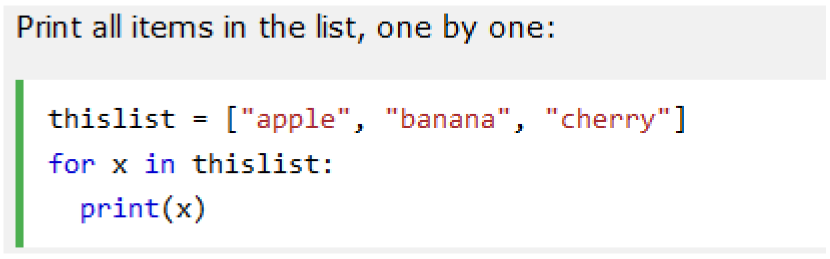
To change the value of a specific item, refer to the index number:


Change the second item:

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

Loop Through a List

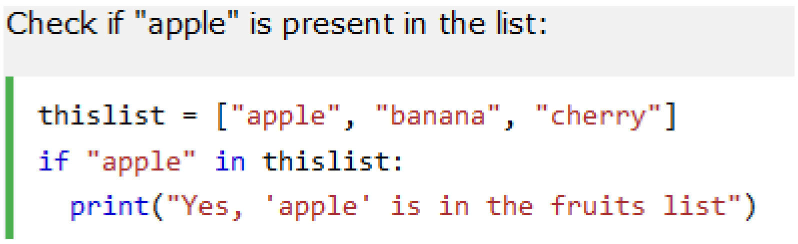
You can loop through the list items by using a `for` loop:


Print all items in the list, one by one:

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

Check if Item Exists

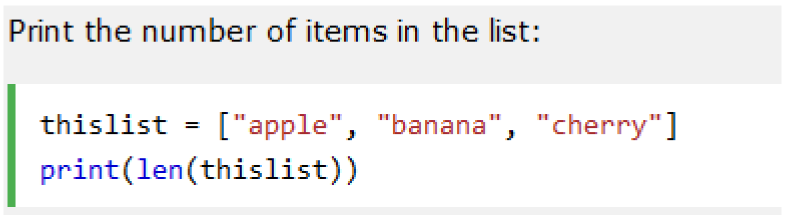
To determine if a specified item is present in a list use the `in` keyword:


Check if "apple" is present in the list:

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

List Length

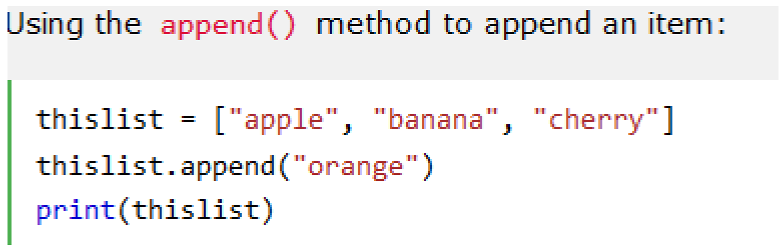
To determine how many items a list has, use the `len()` function:


Print the number of items in the list:

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

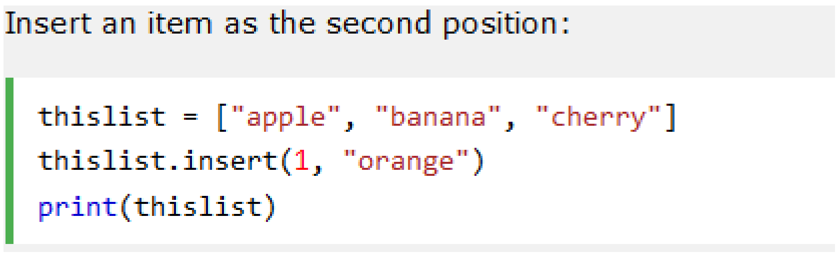
Add Items

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


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

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

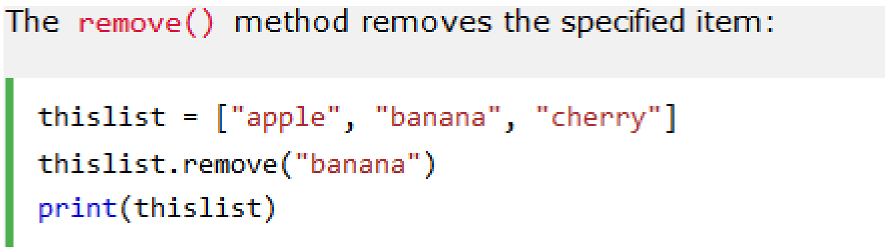
To add an item at the specified index, use the insert() method:


Insert an item as the second position:

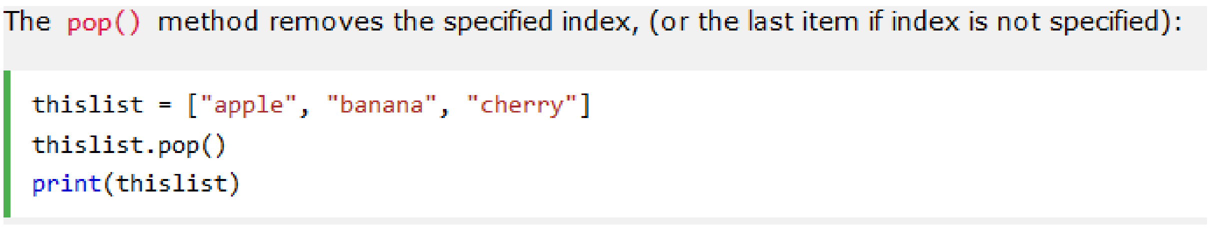
```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

Remove Item

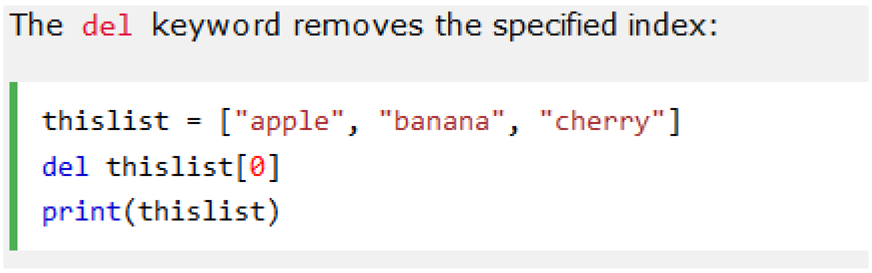
There are several methods to remove items from a list:


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

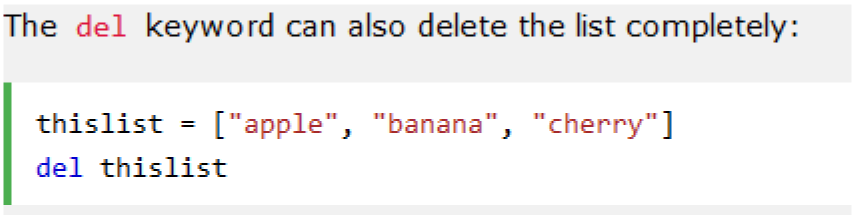
```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```


The `pop()` method removes the specified index, (or the last item if index is not specified):

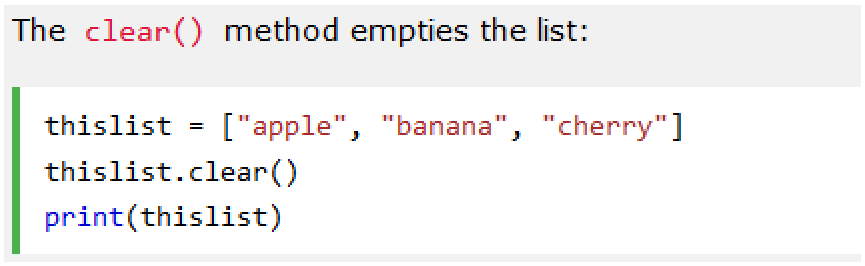
```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```


The `del` keyword removes the specified index:

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


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

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


The `clear()` method empties the list:

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

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()`.

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()`.

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.

Join two list:

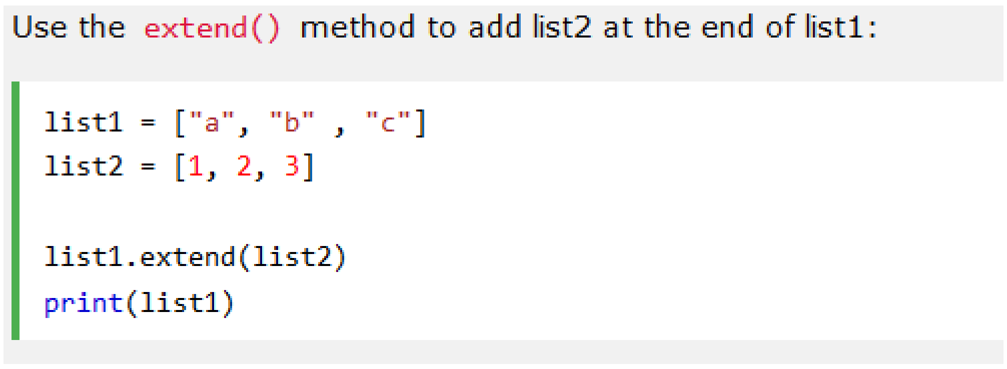
```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

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

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

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:

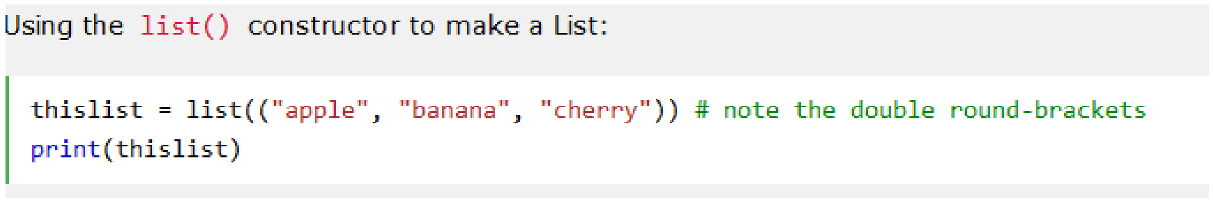


Use the extend() method to add list2 at the end of list1:

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

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

The list() Constructor

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

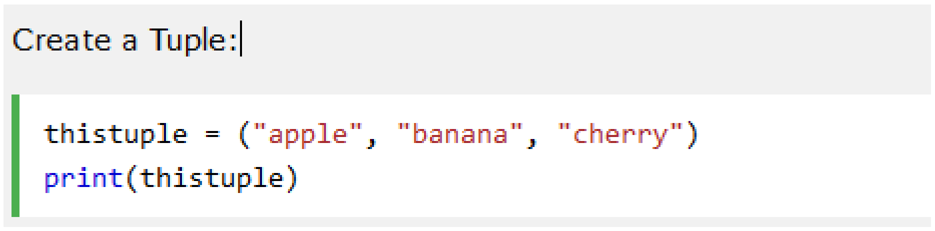


Using the list() constructor to make a List:

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

Tuple

A tuple is a collection which is ordered and **unchangeable**. In Python tuples are written with round brackets.

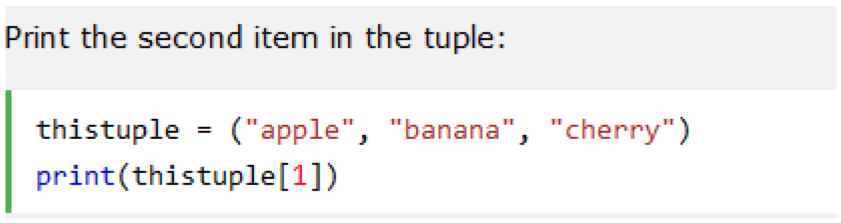


Create a Tuple:

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

Access Tuple Items

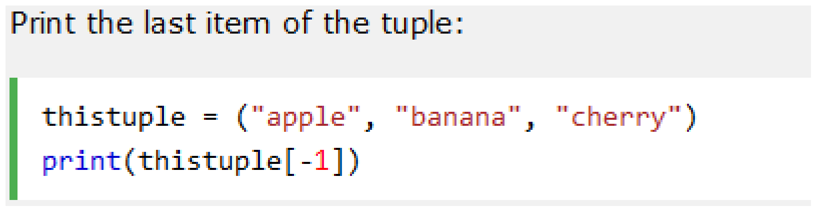
You can access tuple items by referring to the index number, inside square brackets:


Print the second item in the tuple:

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

Negative Indexing

Negative indexing means beginning from the end, `-1` refers to the last item, `-2` refers to the second last item etc.

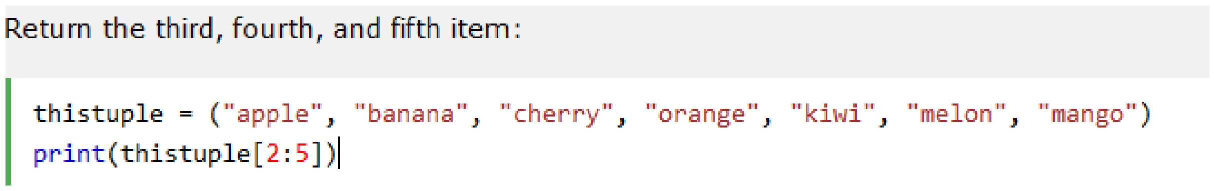

Print the last item of the tuple:

```
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.


Return the third, fourth, and fifth item:

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

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.

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)
```

Loop Through a Tuple

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

Iterate through the items and print the values:

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

Check if Item Exists

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

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

Tuple Length

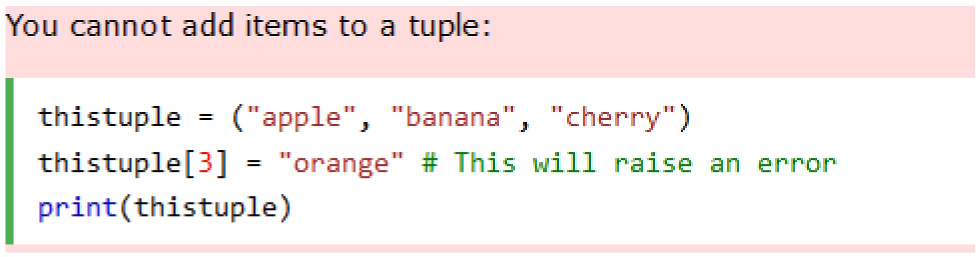
To determine how many items a tuple has, use the `len()` method:

Print the number of items in the tuple:

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

Add Items

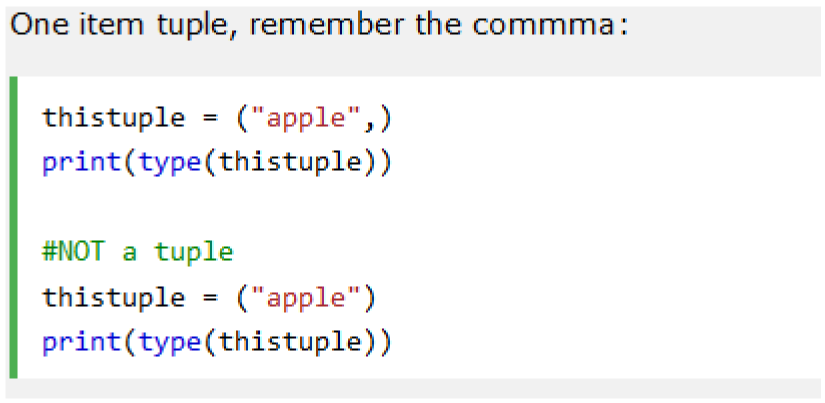
Once a tuple is created, you cannot add items to it. Tuples are **unchangeable**.


You cannot add items to a tuple:

```
thistuple = ("apple", "banana", "cherry")
thistuple[3] = "orange" # This will raise an error
print(thistuple)
```

Create Tuple With One Item

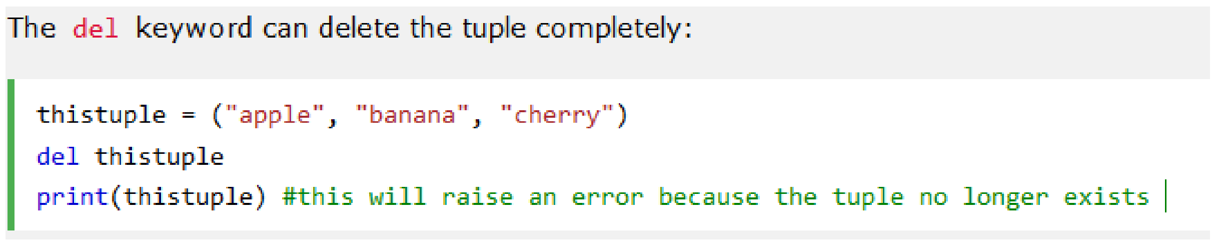
To create a tuple with only one item, you have add a comma after the item, unless Python will not recognize the variable as a tuple.


One item tuple, remember the commma:

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

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

Remove Items

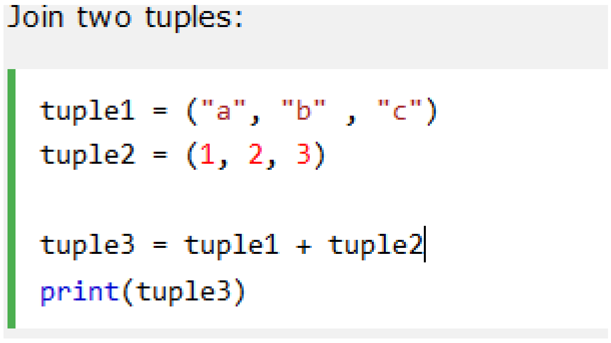
Tuples are **unchangeable**, so you cannot remove items from it, but you can delete the tuple completely:


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
```

Join Two Tuples

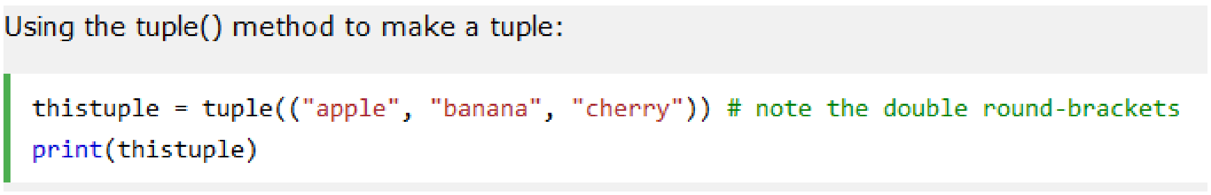
To join two or more tuples you can use the `+` operator:


Join two tuples:

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

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

The tuple() Constructor

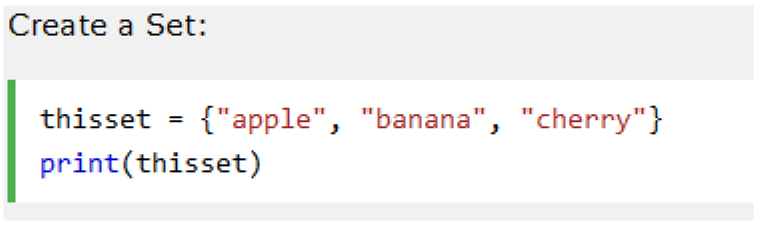
It is also possible to use the tuple() constructor to make a tuple.


Using the tuple() method to make a tuple:

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

Set

A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.


Create a Set:

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

Access Items

You cannot access items in a set by referring to an index, since sets are unordered the items has no index.

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.

Loop through the set, and print the values:

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

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

Check if "banana" is present in the set:

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

print("banana" in thisset)
```

Change Items

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

Add Items

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

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

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

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

thisset.add("orange")

print(thisset)
```

Add multiple items to a set, using the `update()` method:

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

thisset.update(["orange", "mango", "grapes"])

print(thisset)
```

Get the Length of a Set

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

Get the number of items in a set:

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

print(len(thisset))
```

Remove Item

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

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

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

thisset.remove("banana")

print(thisset)
```

You can also use the `pop()`, method to remove an item, but this method will remove the *last* item. Remember that sets are unordered, so you will not know what item that gets removed.

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

Remove the last 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)
```

The `del` keyword will delete the set completely:

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

del thisset

print(thisset)
```

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:

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

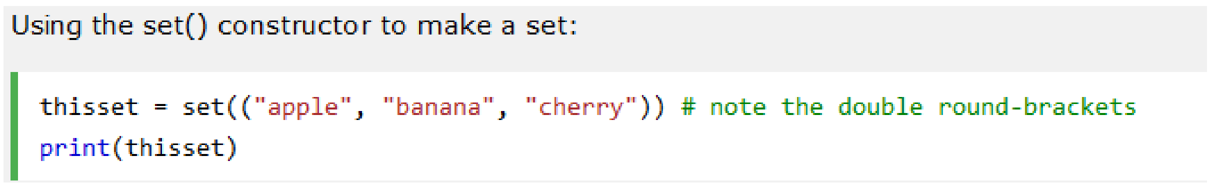
```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

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)
```

The set() Constructor

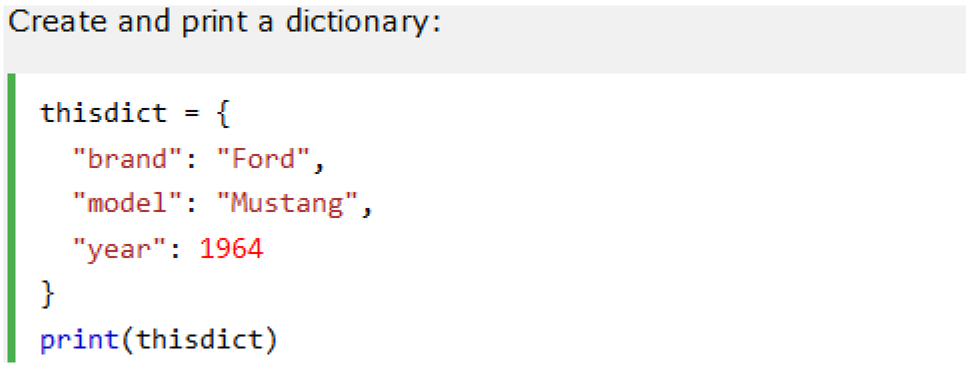
It is also possible to use the set() constructor to make a set.


Using the set() constructor to make a set:

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

Dictionary

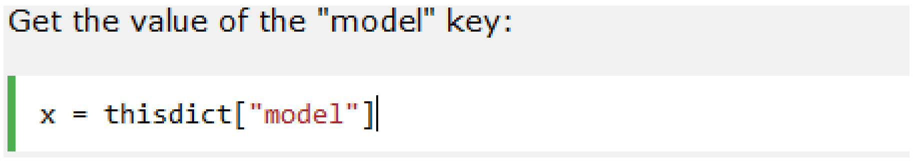
A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.


Create and print a dictionary:

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

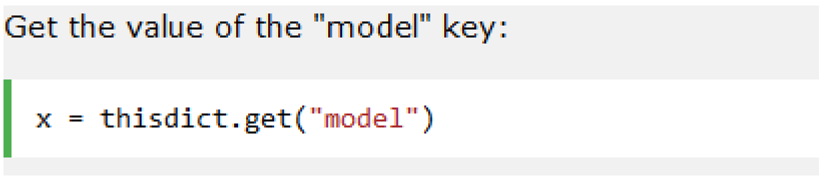
Accessing Items

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


Get the value of the "model" key:

```python
x = thisdict["model"]
```

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


Get the value of the "model" key:

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

Change Values

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

Change the "year" to 2018:

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

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.

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

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

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

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

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

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

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

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

Check if Key Exists

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

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")
```

Dictionary Length

To determine how many items (key-value pairs) a dictionary has, use the `len()` method.

Print the number of items in the dictionary:

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

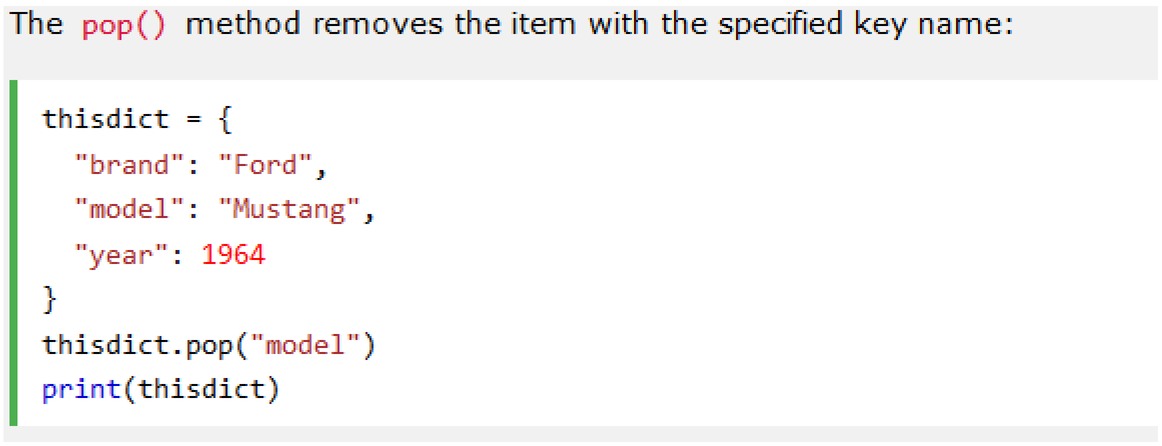
Adding Items

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

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

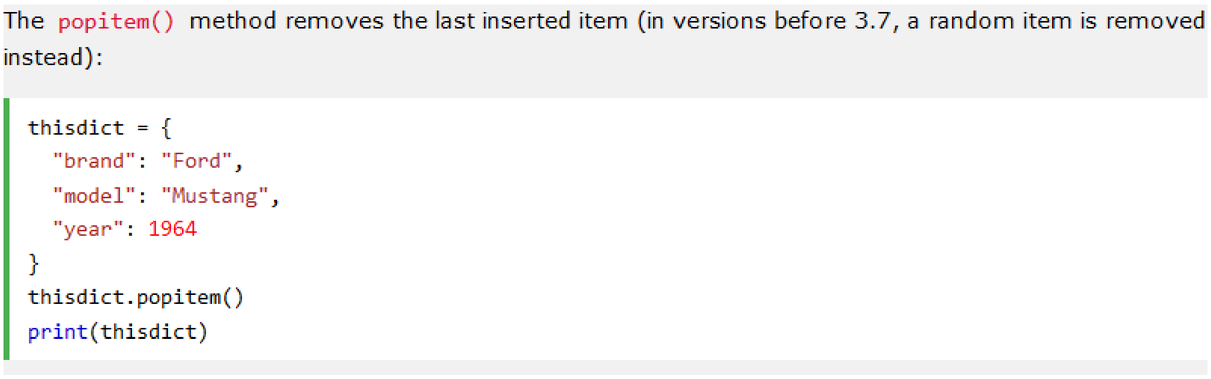
Removing Items

There are several methods to remove items from a dictionary:



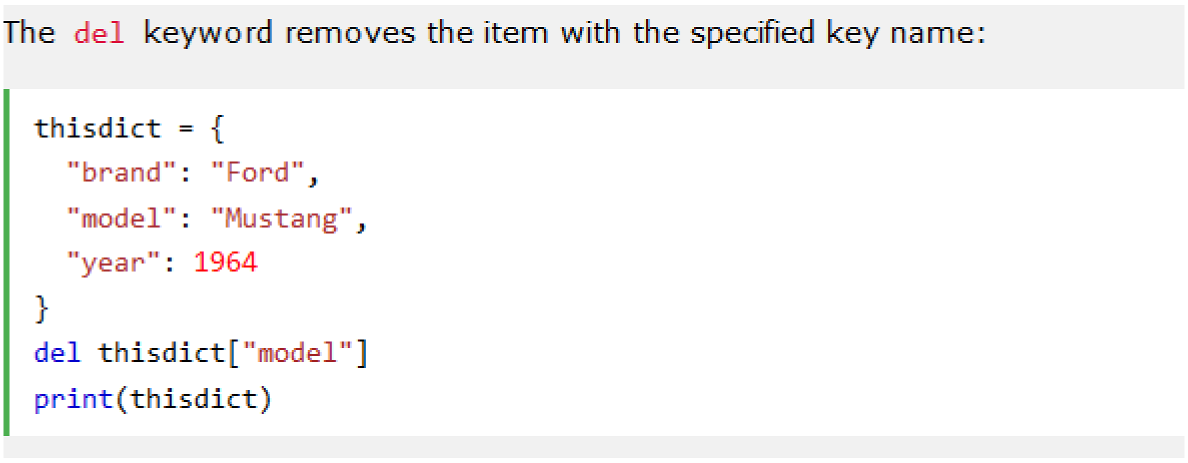
The `pop()` method removes the item with the specified key name:

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



The `popitem()` method removes the last inserted item (in versions before 3.7, a random item is removed instead):

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



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

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

The `del` keyword can also delete the dictionary completely:

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

The `clear()` keyword empties the dictionary:

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

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()`.

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 method `dict()`.

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

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

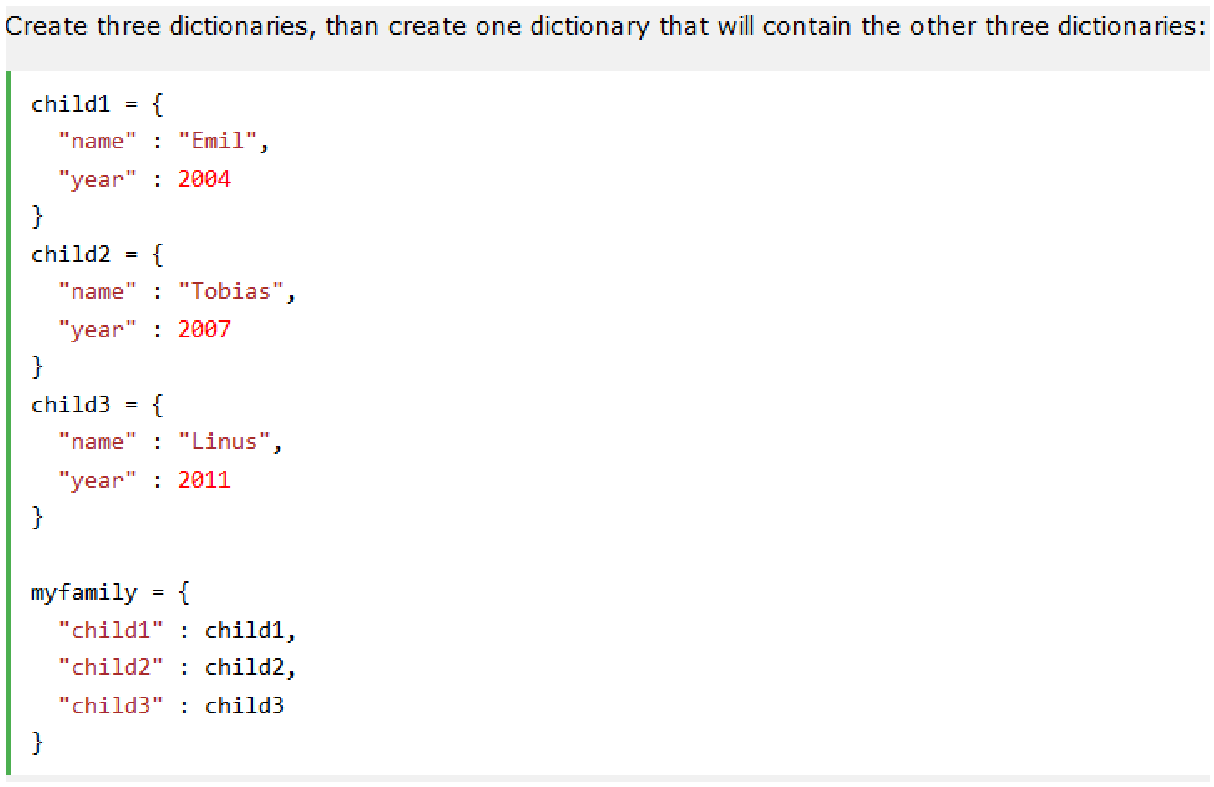
Nested Dictionaries

A dictionary can also contain many dictionaries, this is called nested dictionaries.

Create a dictionary that contain three dictionaries:

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

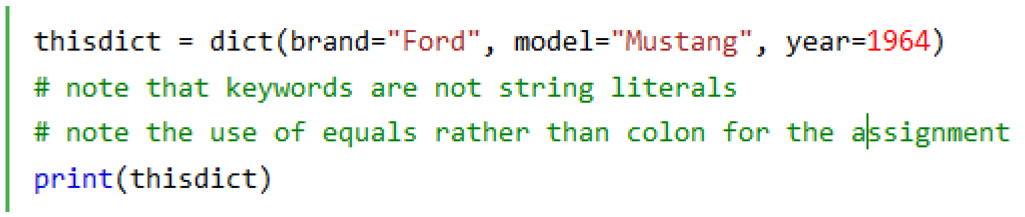
Or, if you want to nest three dictionaries that already exists as dictionaries:


Create three dictionaries, than create one dictionary that will contain the other three dictionaries:

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

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

The dict() Constructor

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



```
thisdict = dict(brand="Ford", model="Mustang", year=1964)
# note that keywords are not string literals
# note the use of equals rather than colon for the assignment
print(thisdict)
```

Python Functions

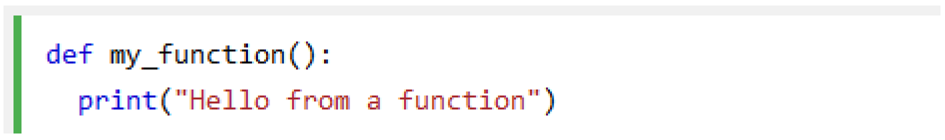
A function is a block of code which only runs when it is called.

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

A function can return data as a result.

Creating a Function

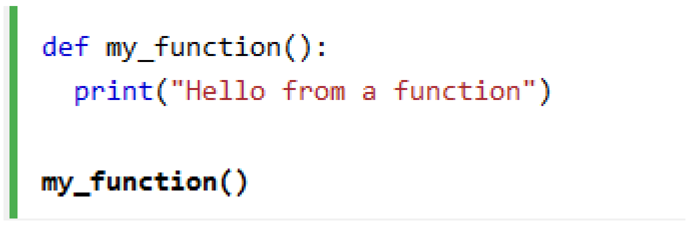
In Python a function is defined using the def keyword:



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

Calling a Function

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



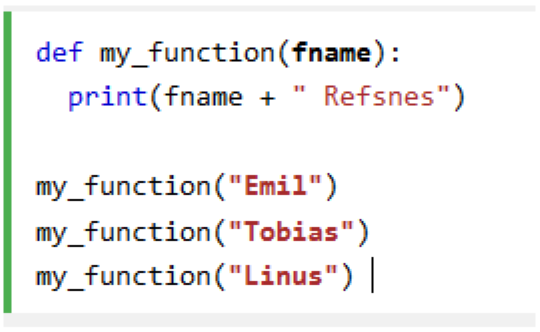
```python
def my_function():
  print("Hello from a function")

my_function()
```

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:



```python
def my_function(fname):
  print(fname + " Refsnes")

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.

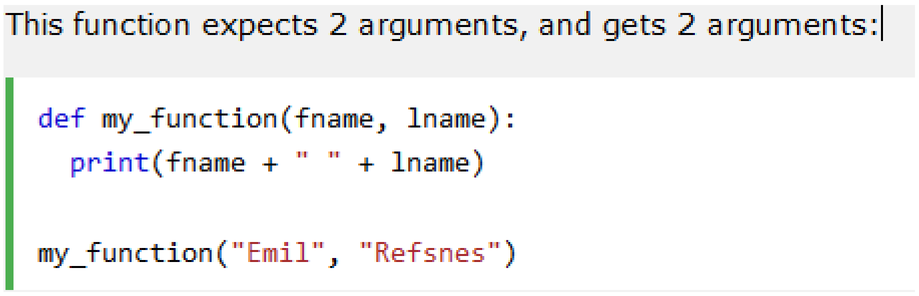
From a function's perspective:

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

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

Number of Arguments

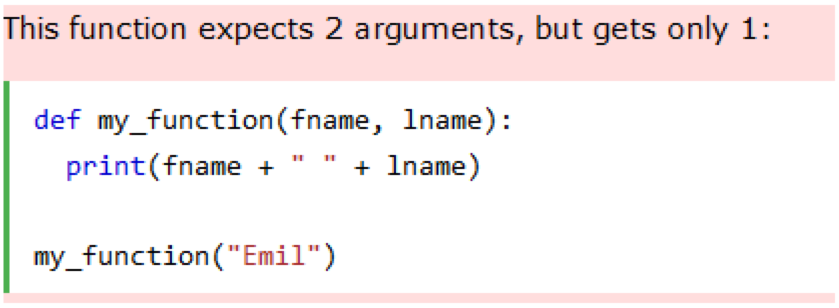
By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.



This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

If you try to call the function with 1 or 3 arguments, you will get an error:



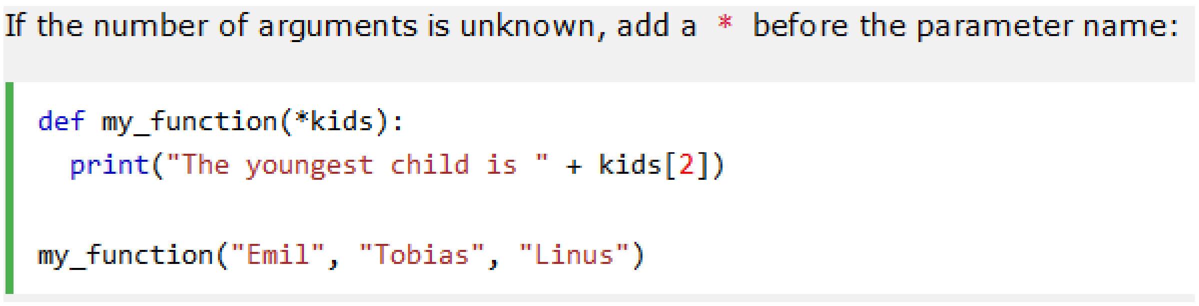
This function expects 2 arguments, but gets only 1:

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil")
```

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:

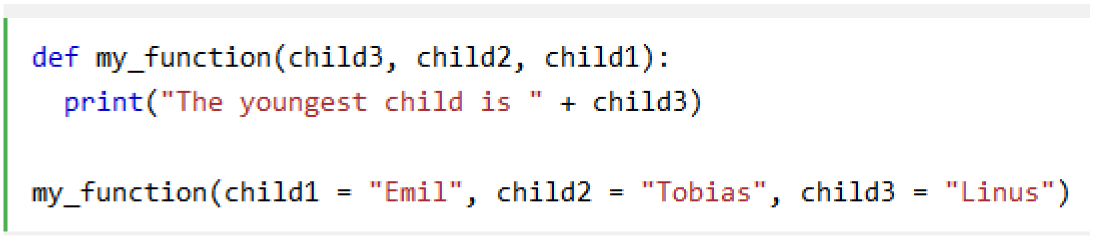

If the number of arguments is unknown, add a `*` before the parameter name:

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

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

Keyword Arguments

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

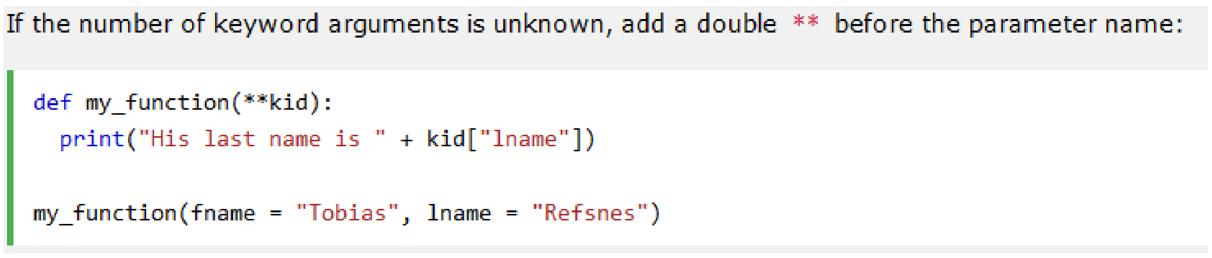
This way the order of the arguments does not matter.


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

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

Arbitrary Keyword Arguments, **kwargs

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

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


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

```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

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:

```
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:

```
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:

```
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.

```
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.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

In this example, tri_recursion() is a function that we have defined to call itself ("recurse"). We use the k variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

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
```

A lambda function that adds 10 to the number passed in as an argument, and print the result:

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

Lambda functions can take any number of arguments:

A lambda function that multiplies argument a with argument b and print the result:

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

A lambda function that sums argument a, b, and c and print 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

mytripler = myfunc(3)

print(mytripler(11))
```

Or, use the same function definition to make both functions, in the same program:

```
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 is an object oriented programming language.

Classes provide a means of bundling data and functionality together.

Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

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.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to class.
- Attributes are always public and can be accessed using dot (.) operator. Eg.: Myclass.Myattribute

Class Objects

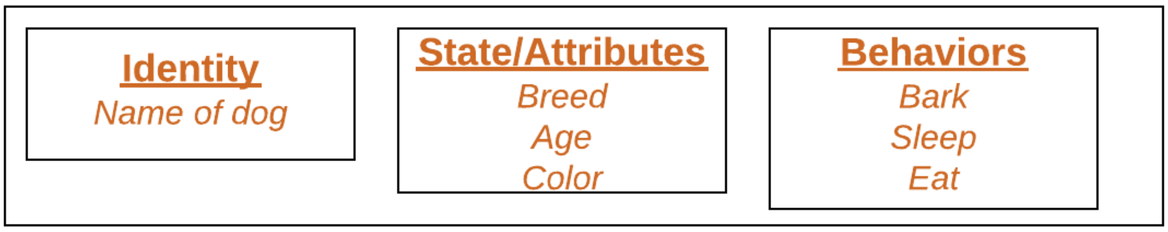
An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual values.

An object consists of :

- **State :** It is represented by attributes of an object. It also reflects the properties of an object.

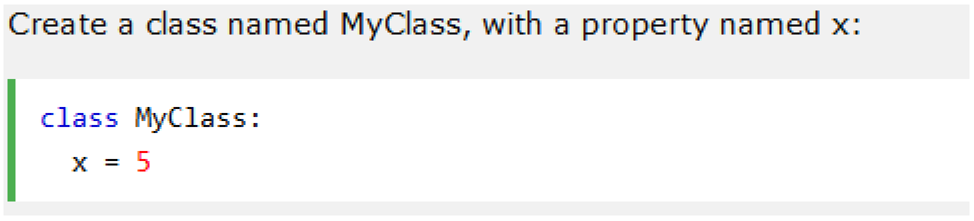
- **Behavior :** It is represented by methods of an object. It also reflects the response of an object with other objects.

- **Identity :** It gives a unique name to an object and enables one object to interact with other objects.



Create a Class

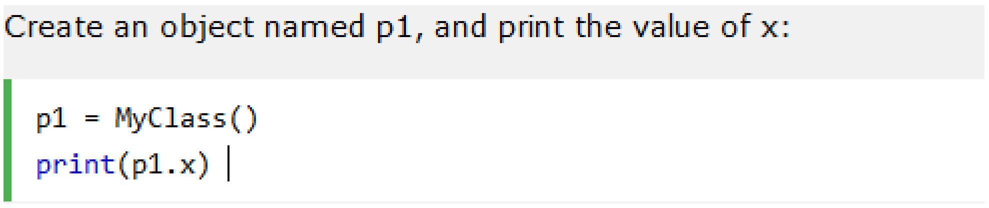
To create a class, use the keyword `class`:


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

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

Create Object

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


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

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

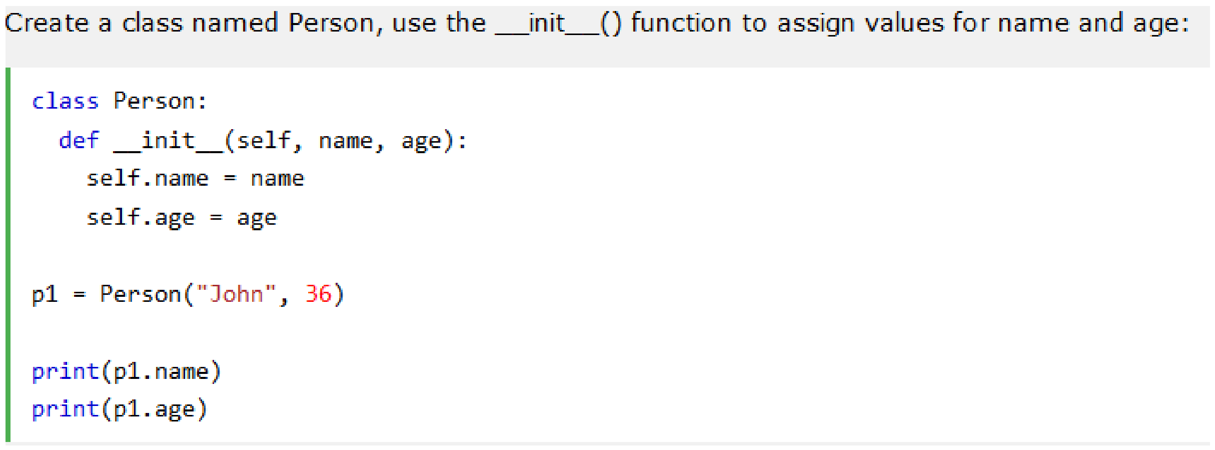
The __init__() Function

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:


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

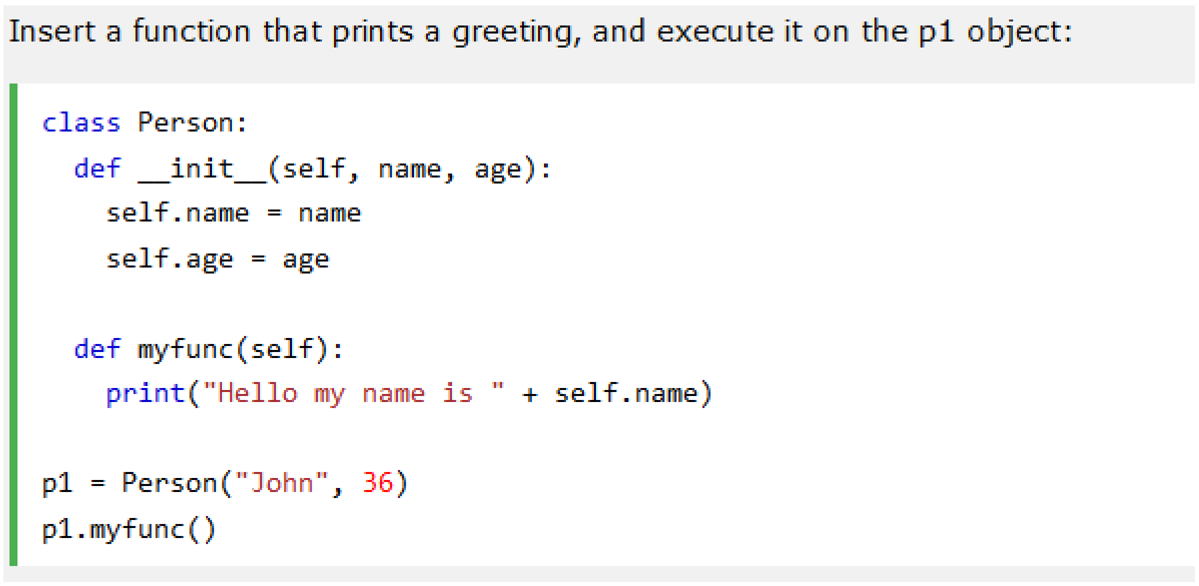
```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

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.

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:



Insert a function that prints a greeting, and execute it on the p1 object:

```python
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()
```

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:

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()
```

Modify Object Properties

You can modify properties on objects like this:

Set the age of p1 to 40:

```
p1.age = 40
```

Delete Object Properties

You can delete properties on objects by using the `del` keyword:

Delete the age property from the p1 object:

```
del p1.age
```

Delete Objects

You can delete objects by using the `del` keyword:

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.

```
class Person:
  pass
```

Access Modifiers in Python : Public, Private and Protected

Various object-oriented languages like C++, Java, Python control access modifications which are used to restrict access to the variables and methods of the class.

Most programming languages has three forms of access modifiers, which are **Public**, **Protected** and **Private** in a class.

Python uses '_' symbol to determine the access control for a specific data member or a member function of a class. Access specifiers in Python have an important role to play in securing data from unauthorized access and in preventing it from being exploited.

A Class in Python has three types of access modifiers –

- **Public Access Modifier**
- **Protected Access Modifier**
- **Private Access Modifier**

Public Access Modifier:

- The members of a class which are declared public are easily accessible from any part of the program. All data members and member functions of a class are public by default.

```
class employee:
    def __init__(self, name, sal):
        self.name=name
        self.salary=sal
```

You can access employee class's attributes and also modify their values, as shown below.

```
>>> e1=Employee("Kiran",10000)
>>> e1.salary
10000
>>> e1.salary=20000
>>> e1.salary
20000
```

Protected Access Modifier:

Python's convention to make an instance variable **protected** is to add a prefix _ (single underscore) to it. This effectively prevents it to be accessed, unless it is from within a sub-class.

```
class employee:
    def __init__(self, name, sal):
        self._name=name  # protected attribute
        self._salary=sal # protected attribute
```

In fact, this doesn't prevent instance variables from accessing or modifying the instance. You can still perform the following operations:

```
>>> e1=employee("Swati", 10000)
>>> e1._salary
10000
>>> e1._salary=20000
>>> e1._salary
20000
```

Hence, the responsible programmer would refrain from accessing and modifying instance variables prefixed with _ from outside its class.

Private Access Modifier:

a double underscore __ prefixed to a variable makes it **private**. It gives a strong suggestion not to touch it from outside the class. Any attempt to do so will result in an AttributeError:

```
class employee:
    def __init__(self, name, sal):
        self.__name=name  # private attribute
        self.__salary=sal # private attribute
```

```
>>> e1=employee("Bill",10000)
>>> e1.__salary
AttributeError: 'employee' object has no attribute '__salary'
```

Python public variables and methods

```
#Python program to create public variables and methods of a class
class Person:
  def __init__(self, firstName, lastName):
    self.firstName = firstName
    self.lastName = lastName

  def display(self):
    print('My name is ', self.firstName, self.lastName)

p = Person("Mitali", "Natani")
print('First name: ', p.firstName)
print('Last name: ', p.lastName)
p.display()
```

Output of the above program:

```
First name:  Mitali
Last name:  Natani
My name is  Mitali Natani
```

Python protected variables and methods

By prefixing variables and methods with a single underscore(_), you can make them protected members. Even after declaring like this, you can still access protected members and Python interpreter will not give any error.

```python
#Python program to create protected variables and methods of a class
class Person:
  def __init__(self, firstName, lastName):
    self._firstName = firstName
    self._lastName = lastName

  def _display(self):
    print('My name is ', self._firstName, self._lastName)

p = Person("Mitali", "Natani")
print('First name: ', p._firstName)
print('Last name: ', p._lastName)
p._display()
```

Output of the above program

```
First name:  Mitali
Last name:  Natani
My name is  Mitali Natani
```

Python Private Variables and Methods

Private variables and methods are the most restrictive members of a class. These members can only be accessed from within the class. There is no `private` keyword in Python to make class members private.

When you prefix variables and methods with double underscore(__), then you make them private member of a class. And if you try to access private members from outside the class then you will get `AttributeError`.

Python program to create private variables and methods

```
class Person:
  def __init__(self, firstName, lastName):
    self.__firstName = firstName
    self.__lastName = lastName

  def __display(self):
    print('My name is ', self.__firstName, self.__lastName)

p = Person("Mitali", "Natani")
print('First name: ', p.__firstName)
print('Last name: ', p.__lastName)
p.__display()
```

Output of the above program

```
AttributeError: 'Person' object has no attribute '__firstName'
```

If you are determined to access private members from outside the class then use this syntax- `object._ClassName__variable` or `object._ClassName__method`. But this is not recommended.

```
class Person:
  def __init__(self, firstName, lastName):
    self.__firstName = firstName
    self.__lastName = lastName

  def __display(self):
    print('My name is ', self.__firstName, self.__lastName)

p = Person("Mitali", "Natani")
print('First name: ', p._Person__firstName)
print('Last name: ', p._Person__lastName)
p._Person__display()
```

Output of the above program

```
First name:  Mitali
Last name:  Natani
My name is Mitai Natani
```

Inheritance

Inheritance in object-oriented programming is pretty similar to real-world inheritance where a child inherits some of the characteristics from his parents, in addition to his/her own unique characteristics.

In object-oriented programming, inheritance signifies an IS-A relation. For instance, a car *is a* vehicle. Inheritance is one of the most amazing concepts of object-oriented programming as it fosters code re-usability.

The basic idea of inheritance in object-oriented programming is that a class can inherit the characteristics of another class. The class which inherits another class is called the *child class* or derived class, and the class which is inherited by another class is called *parent* or base class.

Single Inheritance

Let's take a look at a very simple example of inheritance. Execute the following script:

```
# Create Class Vehicle
class Vehicle:
    def vehicle_method(self):
        print("This is parent Vehicle class method")

# Create Class Car that inherits Vehicle
class Car(Vehicle):
    def car_method(self):
        print("This is child Car class method")
```

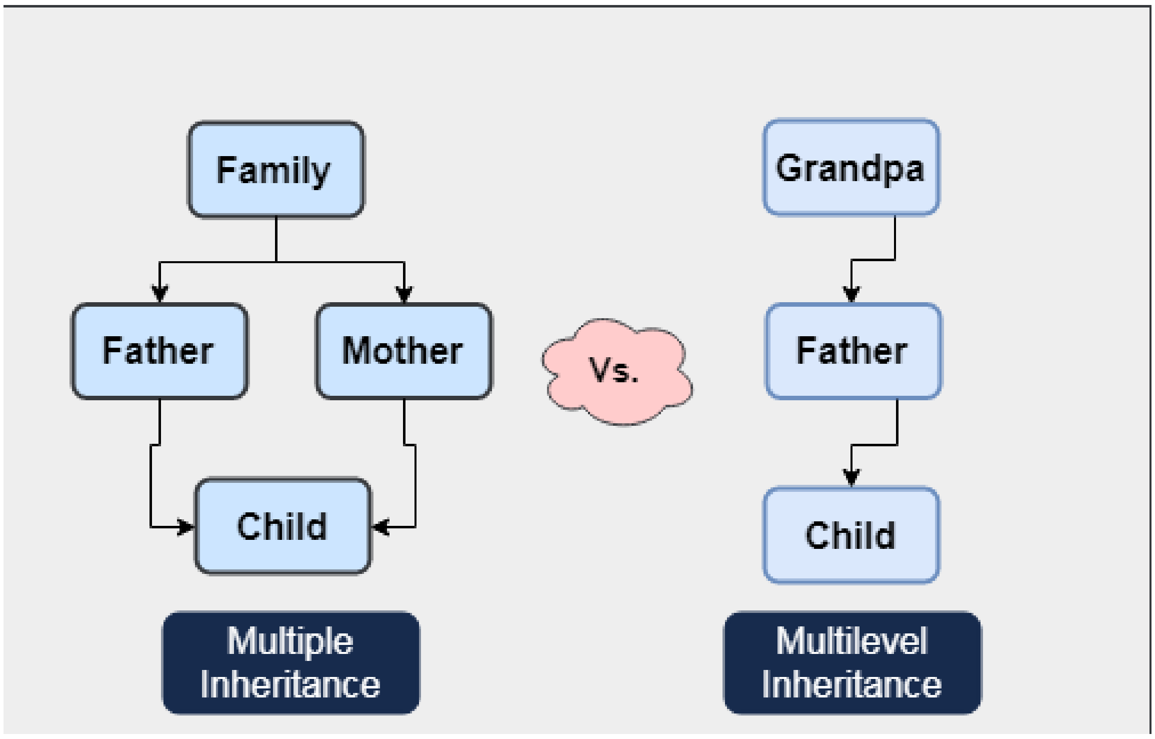
In the script above, we create two classes Vehicle class, and the Car class which inherits the Vehicle class. To inherit a class, you simply have to write the parent class name inside the parenthesis that follows the child class name.
The Vehicle class contains a method **vehicle_method()** and the child class contains a method **car_method().** However, since the Car class inherits the Vehicle class, it will *also* inherit the **vehicle_method().**

Let's see this in action. Execute the following script:

```
car_a = Car()
car_a.vehicle_method() # Calling parent class method
```

In the script above, we create an object of the Car class and call the **vehicle_method()** using that Car class object. You can see that the Car class doesn't have any **vehicle_method()** but since it has inherited the Vehicle class that contains the **vehicle_method(),** the car class can also use it. The output looks likes this:

```
This is parent Vehicle class method
```



Multi-Level Inheritance

In Python, a parent class can have multiple children and similarly, a child class can have multiple parent classes. Let's take a look at the first scenario. Execute the following script:

```
# Create Class Vehicle
class Vehicle:
    def vehicle_method(self):
        print("This is parent Vehicle class method'

# Create Class Car that inherits Vehicle
class Car(Vehicle):
    def car_method(self):
        print("This is child Car class method")

# Create Class Cycle that inherits Vehicle
class Cycle(Vehicle):
    def cycleMethod(self):
        print("This is child Cycle class method")
```

In the script above the parent Vehicle class is inherited by two child classes **Car** and **Cycle**. Both the child classes will have access to the **vehicle_method()** of the parent class. Execute the following script to see that:

```
car_a = Car()
car_a.vehicle_method() # Calling parent class method
car_b = Cycle()
car_b.vehicle_method() # Calling parent class method
```

In the output, you will see the output of the **vehicle_method()** method twice as shown below:

```
This is parent Vehicle class method
This is parent Vehicle class method
```

Multiple Inheritance

When you inherit a child class from more than one base classes, that situation is known as Multiple Inheritance. It, however, exhibits the same behavior as does the single inheritance.

```python
# Parent class 1
class TeamMember(object):
   def __init__(self, name, uid):
      self.name = name
      self.uid = uid

# Parent class 2
class Worker(object):
   def __init__(self, pay, jobtitle):
      self.pay = pay
      self.jobtitle = jobtitle

# Deriving a child class from the two parent classes
class TeamLeader(TeamMember, Worker):
   def __init__(self, name, uid, pay, jobtitle, exp):
      self.exp = exp
      TeamMember.__init__(self, name, uid)
      Worker.__init__(self, pay, jobtitle)
      print("Name: {}, Pay: {}, Exp: {}".format(self.name, self.pay, self.exp))

TL = TeamLeader('Jake', 10001, 250000, 'Scrum Master', 5)
```

Hierarchical Inheritance

When more than one class inherits from a class, it is hierarchical Python inheritance.

```
class A:
                pass
class B(A):
                pass
class C(A):
                pass
issubclass(B,A) and issubclass(C,A)
```

Hybrid Inheritance

Hybrid Python inheritance is a combination of any two kinds of inheritance.

```
class A:
                x=1
class B(A):
                pass
class C(A):
                pass
class D(B,C):
                pass
dobj=D()
print(dobj.x)
```

Super Function

With inheritance, the super() function in python actually comes in quite handy. It allows us to call a method from the parent class. Let's define a new class for this.

```
class Vehicle:
                def start(self):
                                print("Starting engine")
                def stop(self):
                                print("Stopping engine")
class TwoWheeler(Vehicle):
                def say(self):
                                super().start()
                                print("I have two wheels")
                                super().stop()

Pulsar=TwoWheeler()
Pulsar.say()
```

Polymorphism

The term polymorphism literally means having multiple forms. In the context of object-oriented programming, polymorphism refers to the ability of an object to behave in multiple ways.

Polymorphism in programming is implemented via method-overloading and method overriding.

Method Overloading

Method overloading refers to the property of a method to behave in different ways depending upon the number or types of the parameters. Take a look at a very simple example of method overloading. Execute the following script:

```python
# Creates class Car
class Car:
   def start(self, a, b=None):
        if b is not None:
            print (a + b)
        else:
            print (a)
```

In the script above, if the start() method is called by passing a single argument, the parameter will be printed on the screen. However, if we pass 2 arguments to the start() method, it will add both the arguments and will print the result of the sum.

Let's try with single argument first:

```python
car_a = Car()
car_a.start(10)
```

In the output, you will see 10. Now let's try to pass 2 arguments:

```python
car_a.start(10,20)
```

In the output, you will see 30.

Method Overriding

Method overriding refers to having a method with the same name in the child class as in the parent class. The definition of the method differs in parent and child classes but the name remains the same. Let's take a simple example method overriding in Python.

```
# Create Class Vehicle
class Vehicle:
    def print_details(self):
        print("This is parent Vehicle class method")

# Create Class Car that inherits Vehicle
class Car(Vehicle):
    def print_details(self):
        print("This is child Car class method")

# Create Class Cycle that inherits Vehicle
class Cycle(Vehicle):
    def print_details(self):
        print("This is child Cycle class method")
```

In the script above the **Car** and **Cycle** classes inherit the Vehicle class. The vehicle class has **print_details()** method, which is overridden by the child classes. Now if you call the **print_details()** method, the output will depend upon the object through which the method is being called. Execute the following script to see this concept in action:

```
car_a = Vehicle()
car_a. print_details()

car_b = Car()
car_b.print_details()

car_c = Cycle()
car_c.print_details()
```

The output will look like this:

```
This is parent Vehicle class method
This is child Car class method
This is child Cycle class method
```

Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation. In Python, we denote private attribute using underscore as prefix i.e single " _ " or double " __".

```
class Computer:

    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))



c = Computer()
c.sell()

# change the price
c.__maxprice = 1000
print(c.__maxprice)
c.sell()

# using setter function
#c.setMaxPrice(1000)
#c.sell()
```

Decorators in Python

In Python, functions are the first class objects, which means that –

- Functions are objects; they can be referenced to, passed to a variable and returned from other functions as well.
- Functions can be defined inside another function and can also be passed as argument to another function.

Decorators are very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class. Decorators allow us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it.

In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

Even though it is the same underlying concept, we have two different kinds of decorators in Python:

- Function decorators
- Class decorators

A reference to a function "func" or a class "C" is passed to a decorator and the decorator returns a modified function or class. The modified functions or classes usually contain calls to the original function "func" or class "C".

First Steps to Decorators:

function names are references to functions and that we can assign multiple names to the same function:

```
def addar(x):
    return x + 1

add= addar
add(10)
addar(10)
```

This means that we have two names, i.e. "add" and "addar" for the same function. The next important fact is that we can delete either "add" or "addar" without deleting the function itself.

Functions inside Functions

The concept of having or defining functions inside of a function is completely new to C or C++ programmers:

```
def f():

    def g():
        print("Hi, it's me 'g'")
        print("Thanks for calling me")

    print("This is the function 'f'")
    print("I am calling 'g' now:")
    g()


f()
```

We will get the following output, if we start the previous program:

```
>>> This is the function 'f'
I am calling 'g' now:
Hi, it's me 'g'
Thanks for calling me
```

Another example using "proper" return statements in the functions:

```
def temperature(t):
    def celsius2fahrenheit(x):
        return 9 * x / 5 + 32

    result = "It's " + str(celsius2fahrenheit(t)) + " degrees!"
    return result

print(temperature(20))
```

```
>>> It's 68.0 degrees!
```

Functions as Parameters:

If you solely look at the previous examples, this doesn't seem to be very useful. It gets useful in combination with two further powerful possibilities of Python functions. Due to the fact that every parameter of a function is a reference to an object and functions are objects as well, we can pass functions - or better "references to functions" - as parameters to a function. We will demonstrate this in the next simple example:

```
def g():
    print("Hi, it's me 'g'")
    print("Thanks for calling me")

def f(func):
    print("Hi, it's me 'f'")
    print("I will call 'func' now")
    func()

f(g)
```

You may not be satisfied with the output. 'f' should write that it calls 'g' and not 'func'. Of course, we need to know what the 'real' name of func is. For this purpos, we can use the attribute __name__, as it contains this name:

```
def g():
    print("Hi, it's me 'g'")
    print("Thanks for calling me")

def f(func):
    print("Hi, it's me 'f'")
    print("I will call 'func' now")
    func()
    print("func's real name is " + func.__name__)
f(g)
```

Simple decorator

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)
    return function_wrapper

def foo(x):
    print("Hi, foo has been called with " + str(x))

print("We call foo before decoration:")
foo("Hi")

print("We now decorate foo with f:")
foo = our_decorator(foo)

print("We call foo after decoration:")
foo(42)
```

We will rewrite now our initial example. Instead of writing the statement

foo = our_decorator(foo)

we can write

@our_decorator

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)
    return function_wrapper

@our_decorator
def foo(x):
    print("Hi, foo has been called with " + str(x))

foo("Hi")
```

Using a Class as a Decorator

```
class decorator2:

    def __init__(self, f):
        self.f = f

    def __call__(self):
        print("Decorating", self.f.__name__)
        self.f()

@decorator2
def foo():
    print("inside foo()")

foo()
```

Iterators

An iterator can be seen as a pointer to a container, e.g. a list structure that can iterate over all the elements of this container. The iterator is an abstraction, which enables the programmer to access all the elements of a container (a set, a list and so on) without any deeper knowledge of the data structure of this container object. In some object oriented programming languages, like Perl, Java and Python, iterators are implicitly available and can be used in for each loops, corresponding to for loops in Python.

```
cities = ["Paris", "Berlin", "Hamburg", "Frankfurt"]
for location in cities:
     print("location: " + location)
```

What is really happening, when you use an iterable like a string, a list, or a tuple, inside of a for loop is the following: The function "iter" is called on the iterable. The return value of iter is an iterable. We can iterate over this iterable with the next function until the iterable is exhausted and returns a Stop Iteration exception:

```
expertises = ["Novice", "Beginner", "Intermediate", "Proficient"]
expertises_iterator = iter(expertises)
next(expertises_iterator)
next(expertises_iterator)
next(expertises_iterator)
next(expertises_iterator)
next(expertises_iterator)
```

```
other_cities = ["Strasbourg", "Freiburg", "Stuttgart",
                "Vienna / Wien", "Hannover", "Berlin",
                "Zurich"]

city_iterator = iter(other_cities)
while city_iterator:
    try:
        city = next(city_iterator)
        print(city)
    except StopIteration:
        break
```

Generators:

On the surface generators in Python look like functions, but there is both a syntactic and a semantic difference. One distinguishing characteristic is the yield statements. The yield statement turns a functions into a generator.

A generator is a function which returns a generator object. This generator object can be seen like a function which produces a sequence of results instead of a single object.

This sequence of values is produced by iterating over it, e.g. with a for loop.

Everything which can be done with a generator can also be implemented with a class based iterator as well. But the crucial advantage of generators consists in automatically creating the methods __iter__() and next().

```
def city_generator():
    yield("London")
    yield("Hamburg")
    yield("Konstanz")
    yield("Amsterdam")
    yield("Berlin")
    yield("Zurich")
    yield("Schaffhausen")
    yield("Stuttgart")
```

Result:

```
>>> city = city_generator()
>>> next(city)
'London'
>>> next(city)
'Hamburg'
>>> next(city)
'Konstanz'
>>> next(city)
'Amsterdam'
>>> next(city)
'Berlin'
>>> next(city)
'Zurich'
>>> next(city)
'Schaffhausen'
>>> next(city)
'Stuttgart'
>>> next(city)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    next(city)
StopIteration
```

generators can also use return statements, but a generator still needs at least one yield statement to be a generator! A return statement inside of a generator is equivalent to raise StopIteration().

```python
def gen():
    yield 1
    raise StopIteration(45)
```

```
>>> g=gen()
>>> g=gen()
>>> next(g)
1
>>> next(g)
Traceback (most recent call last):
  File "C:\Users\aspl\Desktop\python prog\test.py", line 3, in gen
    raise StopIteration(45)
StopIteration: 45
```

```python
def gen1():
    for char in "Python":
        yield char
    for i in range(5):
        yield i

def gen2():
    yield from "Python"
    yield from range(5)

g1 = gen1()
g2 = gen2()
print("g1: ", end=", ")
for x in g1:
    print(x, end=", ")
print("\ng2: ", end=", ")
for x in g2:
    print(x, end=", ")
print()
```

Exception Handling

An exception is an error that happens during the execution of a program. The name "exception" in computer science has this meaning as well: It implies that the problem (the exception) doesn't occur frequently, i.e. the exception is the "exception to the rule".

Many programming languages like C++, Objective-C, PHP, Java, Ruby, Python, and many others have built-in support for exception handling.

Error handling is generally resolved by saving the state of execution at the moment the error occurred and interrupting the normal flow of the program to execute a special function or piece of code, which is known as the exception handler. Depending on the kind of error ("division by zero", "file open error" and so on) which had occurred, the error handler can "fix" the problem and the program can be continued afterwards with the previously saved data.

Exception Handling in Python

Exceptions handling in Python is very similar to Java. The code, which harbours the risk of an exception, is embedded in a try block. But whereas in Java exceptions are caught by catch clauses, we have statements introduced by an "except" keyword in Python. It's possible to create "custom-made" exceptions: With the raise statement it's possible to force a specified exception to occur.

Let's look at a simple example. Assuming we want to ask the user to enter an integer number. If we use a input(), the input will be a string, which we have to cast into an integer. If the input has not been a valid integer, we will generate (raise) a ValueError. We show this in the following interactive session:

```
>>> n = int(input("Please enter a number: "))
Please enter a number: 23.5
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    n = int(input("Please enter a number: "))
ValueError: invalid literal for int() with base 10: '23.5'
```

With the aid of exception handling, we can write robust code for reading an integer from input:

```
while True:
    try:
        n = input("Please enter an integer: ")
        n = int(n)
        break
    except ValueError:
        print("No valid integer! Please try again ...")
print("Great, you successfully entered an integer!")
```

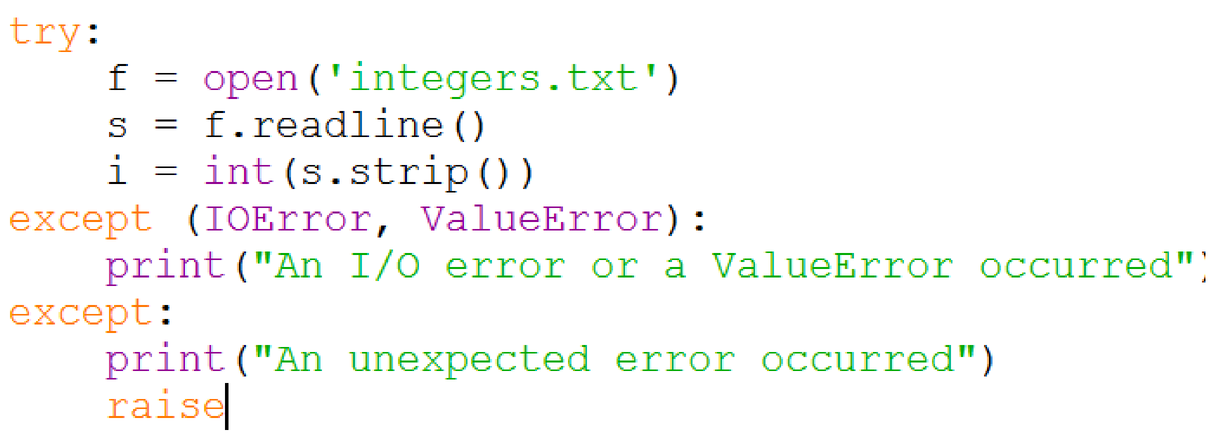
A try statement may have more than one except clause for different exceptions. But at most one except clause will be executed.

Our next example shows a try clause, in which we open a file for reading, read a line from this file and convert this line into an integer. There are at least two possible exceptions:

- an IOError
- ValueError

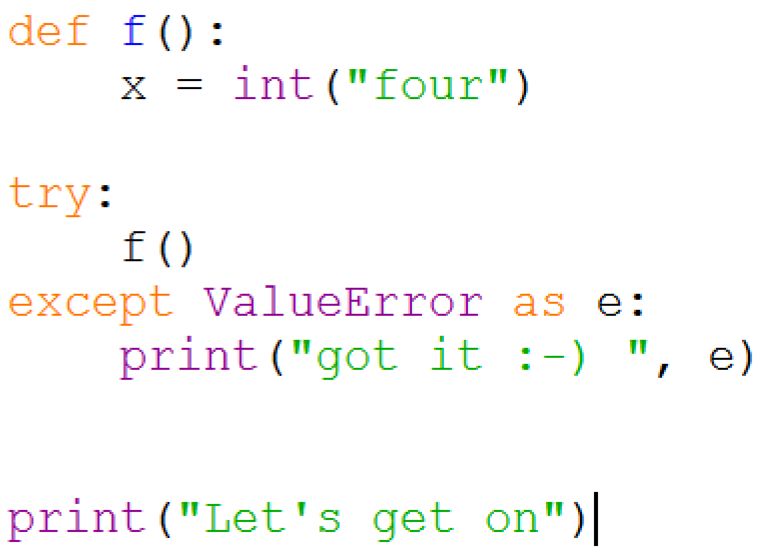
```
try:
    f = open('integers.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print(e)
except ValueError:
    print("No valid integer in line.")
except:
    print("Unexpected error:")
    raise
```

An except clause may name more than one exception in a tuple of error names, as we see in the following example:



```
try:
    f = open('integers.txt')
    s = f.readline()
    i = int(s.strip())
except (IOError, ValueError):
    print("An I/O error or a ValueError occurred"
except:
    print("An unexpected error occurred")
    raise
```

We want to demonstrate now, what happens, if we call a function within a try block and if an exception occurs inside the function call:



```
def f():
    x = int("four")

try:
    f()
except ValueError as e:
    print("got it :-) ", e)


print("Let's get on")
```

We add now a "raise", which generates the ValueError again, so that the exception will be propagated to the caller:

```
def f():
    try:
        x = int("four")
    except ValueError as e:
        print("got it in the function :-) ", e)
        raise

try:
    f()
except ValueError as e:
    print("got it :-) ", e)

print("Let's get on")
```

It's possible to create Exceptions yourself:

```
>>> raise SyntaxError("Sorry, my fault!")
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    raise SyntaxError("Sorry, my fault!")
  File "<string>", line None
SyntaxError: Sorry, my fault!
>>> |
```

```
class MyException(Exception):
    pass

raise MyException("An exception doesn't always prove the rule!")
```

Clean-up Actions (try ... finally)

Finally clauses are called clean-up or termination clauses, because they must be executed under all circumstances, i.e. a "finally" clause is always executed regardless if an exception occurred in a try block or not.

```
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
finally:
    print("There may or may not have been an exception.")
print("The inverse: ", inverse)
```

Combining try, except and finally

"finally" and "except" can be used together for the same try block, as can be seen the following Python example:

```
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
except ValueError:
    print("You should have given either an int or a float")
except ZeroDivisionError:
    print("Infinity")
finally:
    print("There may or may not have been an exception.")
```

Built-in Exceptions:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

Modular Programming

Modular design means that a complex system is broken down into smaller parts or components, i.e. modules.

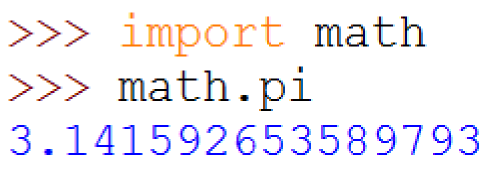
These components can be independently created and tested. In many cases, they can be even used in other systems as well.

When creating a modular system, several modules are built separately and more or less independently. The executable application will be created by putting them together.

Importing Modules

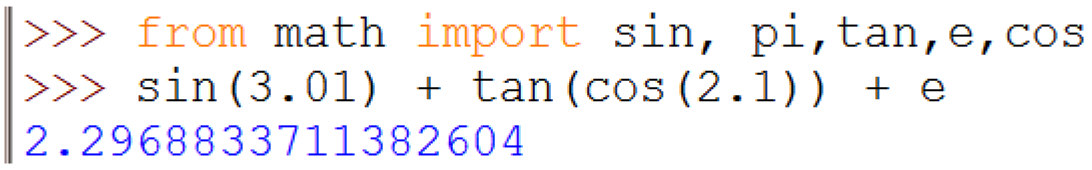
Every file, which has the file extension .py and consists of proper Python code, can be seen or is a module!

A module can contain arbitrary objects, for example files, classes or attributes. All those objects can be accessed after an import. There are different ways to import a modules. We demonstrate this with the math module:



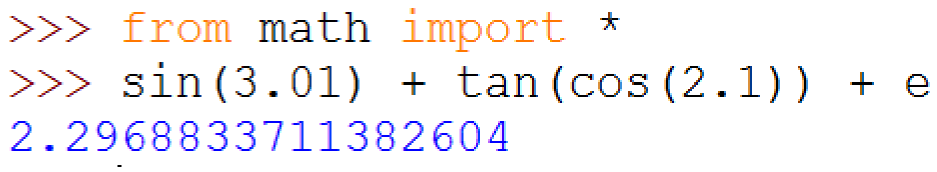
```
>>> import math
>>> math.pi
3.141592653589793
```

If only certain objects of a module are needed, we can import only those:



```
>>> from math import sin, pi,tan,e,cos
>>> sin(3.01) + tan(cos(2.1)) + e
2.2968833711382604
```

Instead of explicitly importing certain objects from a module, it's also possible to import everything in the namespace of the importing module. This can be achieved by using an asterisk in the import:



```
>>> from math import *
>>> sin(3.01) + tan(cos(2.1)) + e
2.2968833711382604
```

Designing and Writing Modules

But how do we create modules in Python? A module in Python is just a file containing Python definitions and statements. The module name is moulded out of the file name by removing the suffix .py. For example, if the file name is fibonacci.py, the module name is fibonacci.

Let's turn our Fibonacci functions into a module. There is hardly anything to be done, we just save the following code in the file fibonacci.py:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

```
>>> import fibonacci

>>> fibonacci.fib(7)
13
```

Packages in Python

A package is basically a directory with Python files and a file with the name __init__.py. This means that every directory inside of the Python path, which contains a file named __init__.py, will be treated as a package by Python. It's possible to put several modules into a Package.

We will demonstrate with a very simple example how to create a package with some Python modules.

First of all, we need a directory. The name of this directory will be the name of the package, which we want to create. We will call our package "simple_package". This directory needs to contain a file with the name "__init__.py". This file can be empty, or it can contain valid Python code. This code will be executed when a package will be imported, so it can be used to initialize a package, e.g. to make sure that some other modules are imported or some values set. Now we can put into this directory all the Python files which will be the submodules of our module.

We create two simple files a.py and b.py just for the sake of filling the package with modules.

The content of a.py:

```
def bar():
    print("Hello, function 'bar' from module 'a' calling")
```

The content of b.py:

```
def foo():
    print("Hello, function 'foo' from module 'b' calling")
```

We will also add an empty file with the name __init__.py inside of simple_package directory.

```
>>> import simple_package

>>> from simple_package import a, b
```

File Handling

A file or a computer file is a chunk of logically related data or information which can be used by computer programs. Usually a file is kept on a permanent storage media, e.g. a hard drive disk. A unique name and path is used by human users or in programs or scripts to access a file for reading and modification purposes.

The most basic tasks involved in file manipulation are reading data from files and writing or appending data to files.

"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)
```

Reading and Writing Files in Python

The syntax for reading and writing files in Python is similar to programming languages like C, C++, Java, Perl, and others but a lot easier to handle.

The way of telling Python that we want to read from a file is to use the open function. The first parameter is the name of the file we want to read and with the second parameter, assigned to the value "r", we state that we want to read from the file:

```
>>> fobj = open("palam.txt", "r")
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    fobj = open("palam.txt", "r")
FileNotFoundError: [Errno 2] No such file or directory: 'palam.txt'
```

The "r" is optional. An open() command with just a file name is opened for reading per default. The open() function returns a file object, which offers attributes and methods.

```
>>> fobj = open("palam.txt")
```

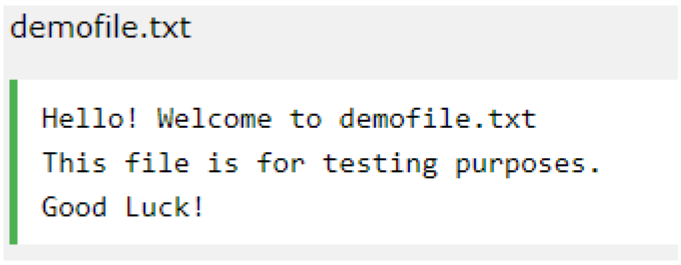
After we have finished working with a file, we have to close it again by using the file object method close():

```
>>> fobj.close()
```

we want to finally open and read a file. The method rstrip() in the following example is used to strip off whitespaces (newlines included) from the right side of the string "line":

```
>>> fobj = open("palam.txt")
for line in fobj:
    print(line.rstrip())
fobj.close()
```

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!
```

The `open()` function returns a file object, which has a `read()` method for reading the content of the file:

```
>>> f = open("demofile.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:

```
>>> f = open("demofile.txt", "r")
print(f.read(5))
```

Read Lines

You can return one line by using the readline() method:

```
>>> f = open("demofile.txt", "r")
print(f.readline())
```

By calling `readline()` two times, you can read the two first lines:

```
>>> f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

By looping through the lines of the file, you can read the whole file, line by line:

```
>>> f = open("demofile.txt", "r")
for x in f:
  print(x)
```

It is a good practice to always close the file when you are done with it.

```
>>> f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

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

```
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())
```

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 appending:
f = open("demofile3.txt", "r")
print(f.read())
```

Delete a File

To delete a file, you must import the OS module, and run its os.remove() function:

```
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:

```
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:

```
import os
os.rmdir("myfolder")
```

Regular Expressions

Regular Expressions are used in programming languages to filter texts or textstrings. It's possible to check, if a text or a string matches a regular expression. A great thing about regular expressions: The syntax of regular expressions is the same for all programming and script languages, e.g. Python, Perl, Java, SED, AWK and even X#.

Regular Expressions can be used in this case to recognize the patterns and extract the required information easily.

Consider the scenario – You are a salesperson and you have a lot of email addresses and a lot of those addresses are fake/invalid.

What you can do is, you can make use of Regular Expressions you can verify the format of the email addresses and filter out the fake IDs from the genuine ones.

The next scenario is pretty similar to the one with the salesperson example.

How do we verify the phone number and then classify it based on the country of origin?

Every correct number will have a particular pattern which can be traced and followed through by using Regular Expressions.

There is other ‘n’ number of scenarios in which Regular Expressions help us.

Syntax of Regular Expression

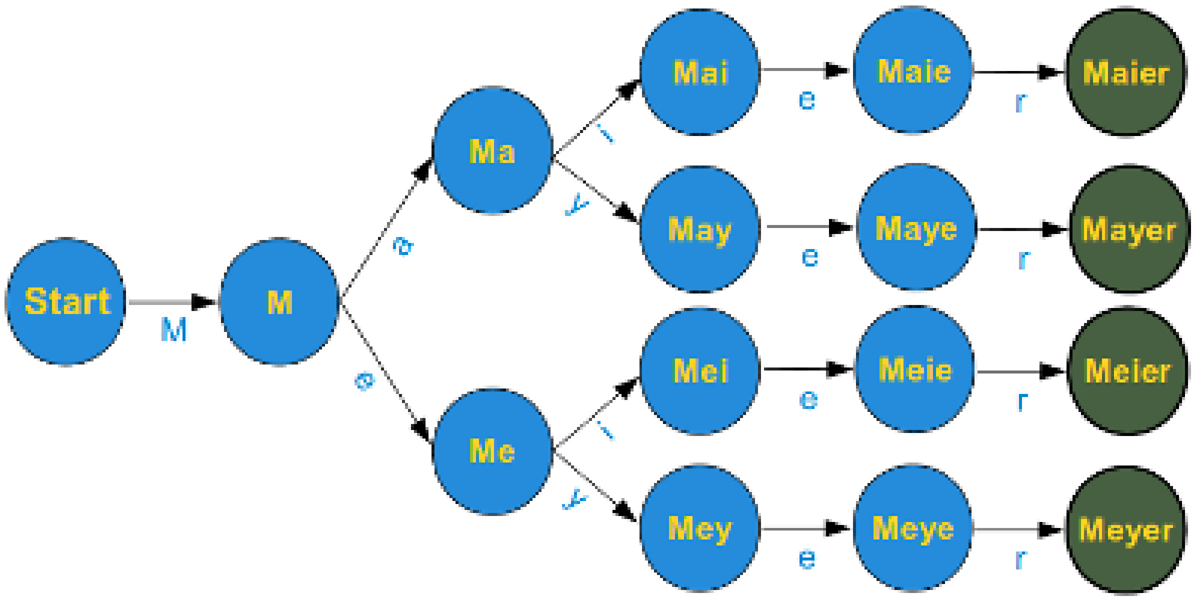
```
import re
if re.search("cat","A cat and a rat can't be friends."):
  print("Some kind of cat has been found :-)")
else:
 print("No cat has been found :-)")
```

Character Classes

Square brackets, "[" and "]", are used to include a character class. [xyz] means e.g. either an "x", an "y" or a "z".
Let's look at a more practical example:

```
>>> r"M[ae][iy]er"
```

This is a regular expression, which matches a surname which is quite common in German. A name with the same pronunciation and four different spellings: Maier, Mayer, Meier, Meyer



Instead of a choice between two characters, we often need a choice between larger character classes. We might need e.g. a class of letters between "a" and "e" or between "0" and "5"

To manage such character classes the syntax of regular expressions supplies a metacharacter "-". [a-e] a simplified writing for [abcde] or [0-5] denotes [012345].

The advantage is obvious and even more impressive, if we have to coin expressions like "any uppercase letter" into regular expressions. So instead of [ABCDEFGHIJKLMNOPQRSTUVWXYZ] we can write [A-Z]. If this is not convincing: Write an expression for the character class "any lower case or uppercase letter" [A-Za-z]

There is something more about the dash, we used to mark the begin and the end of a character class. The dash has only a special meaning if it is used within square brackets and in this case only if it isn't positioned directly after an opening or immediately in front of a closing bracket.

So the expression [-az] is only the choice between the three characters "-", "a" and "z", but no other characters. The same is true for [az-].

The only other special character inside square brackets (character class choice) is the caret "^". If it is used directly after an opening sqare bracket, it negates the choice. [^0-9] denotes the choice "any character but a digit". The position of the caret within the square brackets is crucial. If it is not positioned as the first character following the opening square bracket, it has no special meaning.

[^abc] means anything but an "a", "b" or "c"

[a^bc] means an "a", "b", "c" or a "^"

```
import re
line = "He is a German called Mayer."
if re.search(r"M[ae][iy]er",line): print("I found one!")
```

if we want to match a regular expression at the beginning of a string and only at the beginning?

Predefined Character Classes

The special sequences consist of "\\" and a character from the following list:

\d Matches any decimal digit; equivalent to the set [0-9].

\D The complement of \d. It matches any non-digit character; equivalent to the set [^0-9].

\s Matches any whitespace character; equivalent to [\t\n\r\f\v].

\S The complement of \s. It matches any non-whitespace character; equiv. to [^ \t\n\r\f\v].

\w	Matches any alphanumeric character; equivalent to [a-zA-Z0-9_]. With LOCALE, it will match the set [a-zA-Z0-9_] plus characters defined as letters for the current locale.
\W	Matches the complement of \w.
\b	Matches the empty string, but only at the start or end of a word.
\B	Matches the empty string, but not at the start or end of a word.
\\	Matches a literal backslash.

Example 1:

```
import re

allinform = re.findall('''inform","We need to inform him
                          with the latest information!''')

for i in allinform:
    print(i)
```

Generating an iterator:

Generating an iterator is the simple process of finding out and reporting the starting and the ending index of the string. Consider the following example:

```
import re

Str = "we need to inform him with the latest information"

for i in re.finditer("inform.", Str):
    locTuple = i.span()
    print(locTuple)
```

Matching words with patterns:

Consider an input string where you have to match certain words with the string. To elaborate, check out the following example code:

```
import re

Str = "Sat, hat, mat, pat"

allStr = re.findall("[shmp]at", Str)

for i in allStr:
    print(i)
```

Matching series of range of characters:

We wish to output all the words whose first letter should start in between h and m and compulsorily followed by at. Checking out the following example we should realize the output we should get is hat and mat, correct?

```
import re

Str = "sat, hat, mat, pat"

someStr = re.findall("[h-m]at", Str)

for i in someStr:
    print(i)
```

Let us now change the above program very slightly to obtain a very different result. Check out the below code and try to catch the difference between the above one and the below one:

```
import re

Str = "sat, hat, mat, pat"

someStr = re.findall("[^h-m]at", Str)

for i in someStr:
    print(i)
```

Replacing a string:

we can check out another operation using Regular Expressions where we replace an item of the string with something else. It is very simple and can be illustrated with the following piece of code:

```
import re

Food = "hat rat mat pat"

regex = re.compile("[r]at")

Food = regex.sub("food", Food)

print(Food)
```

Matching a single character:

A single character from a string can be individually matched using Regular Expressions easily. Check out the following code snippet:

```
import re

randstr = "12345"

print("Matches: ", len(re.findall("\d{5}", randstr)))
```

Removing Newline Spaces:

We can remove the newline spaces using Regular Expressions easily in Python. Consider another snippet of code as shown here:

```
import re

randstr = '''
You Never
Walk Alone
Liverpool FC
'''

print(randstr)

regex = re.compile("\n")

randstr = regex.sub(" ", randstr)

print(randstr)
```

Extracting Information From Text

```
import re

Nameage = '''
Janice is 22 and Theon is 33
Gabriel is 44 and Joey is 21
'''

ages = re.findall(r'\d{1,3}', Nameage)
names = re.findall(r'[A-Z][a-z]*',Nameage)

ageDict = {}
x = 0
for eachname in names:
    ageDict[eachname] = ages[x]
    x+=1
print(ageDict)
```

Practical Use Cases of Regular Expressions

We will be checking out 3 main use-cases which are widely used on a daily basis. Following are the concepts we will be checking out:

- Phone Number Verification
- E-mail Address Verification
- Web Scraping

Let us begin this section of Python RegEx by checking out the first case.

Phone Number Verification:

Problem Statement – The need to easily verify phone numbers in any relevant scenario.

Consider the following Phone numbers:

444-122-1234

123-122-78999

111-123-23

67-7890-2019

The general format of a phone number is as follows:

- Starts with 3 digits and '-' sign
- 3 middle digits and '-' sign
- 4 digits in the end

We will be using w in the example below. Note that \w = [a-zA-Z0-9_]

```
import re

phn = "412-555-1212"

if re.search("\w{3}-\w{3}-\w{4}", phn):
    print("Valid phone number")
```

E-mail Verification:

Problem statement – To verify the validity of an E-mail address in any scenario.

Consider the following examples of email addresses:

- Anirudh@gmail.com
- Anirudh @ com
- AC .com
- 123 @.com

Manually, it just takes you one good glance to identify the valid mail IDs from the invalid ones. But how is the case when it comes to having our program do this for us? It is pretty simple considering the following guidelines are followed for this use-case.

Guidelines:

All E-mail addresses should include:

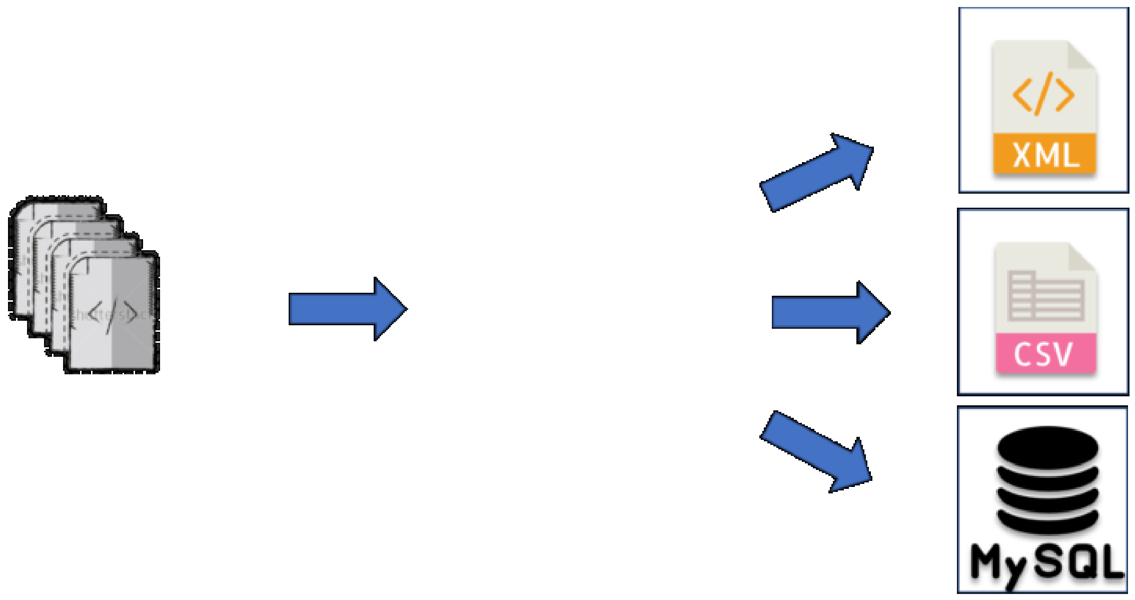
- 1 to 20 lowercase and/or uppercase letters, numbers, plus . _ % +
- An @ symbol
- 2 to 20 lowercase and uppercase letters, numbers and plus
- A period symbol
- 2 to 3 lowercase and uppercase letters

```
import re

email = "ac@aol.com md@.com @seo.com dc@.com"

print("Email Matches: ",
      len(re.findall("[\w._%+-]{1,20}@[\w.-]{2,20}.[A-Za-z]{2,3}", email)))
```

Web Scraping

Problem Statement – Scrapping all of the phone numbers from a website for a requirement.

To understand web scraping, check out the following diagram:



We already know that a single website will consist of multiple web pages. And let us say we need to scrape some information from these pages.

Web scraping is basically used to extract the information from the website. You can save the extracted information in the form of XML, CSV or even a MySQL database. This is achieved easily by making use of Python Regular Expressions.

```python
import urllib.request
from re import findall

url = "http://www.summet.com/dmsi/html/codesamples/addresses.html"

response = urllib.request.urlopen(url)

html = response.read()

htmlStr = html.decode()

pdata = findall("\(\d{3}\) \d{3}-\d{4}", htmlStr)

for item in pdata:
    print(item)


response = urllib.request.urlopen(url)

html = response.read()

htmlStr = html.decode()

pdata = findall("(\d{3}) \d{3}-\d{4}", htmlStr)

for item in pdata:
    print(item)
```

Graphical User Interface

There are various toolkits to make a GUI Application in Python3. It is easy to start programming a GUI application after gaining basic python and OOP knowledge.

- TKinter
- PyQT
- PySide
- PyGTK
- Kivy

TKinter Module:

TKinter is an open source and standard GUI toolkit for Python. TKinter is a wrapper around tcl / TK graphical interface. TKinter is popular because of its simplicity and having very old and active community. Also, it comes included with most binary distributions of Python. TKinter is fully potable for Macintosh, Windows and Linux platforms. It's a good toolkit to start with, since TKinter is mostly preferred for small scale GUI applications.

Widget Class:

Label:

A Label is a Tkinter Widget class, which is used to display text or an image. The label is a widget that the user just views but not interact with.

```
import tkinter as tk

root = tk.Tk()

w = tk.Label(root, text="Hello World!")
w.pack()

root.mainloop()
```

Using Images in Labels

labels can contain text and images. The following example contains two labels, one with a text and the other one with an image.

```
import tkinter as tk

root = tk.Tk()
logo = tk.PhotoImage(file="download.png")

w1 = tk.Label(root, image=logo).pack(side="right")

explanation = """This is a label widget with Image
Image should be of small resoution"""

w2 = tk.Label(root,
              justify=tk.LEFT,
              padx = 10,
              text=explanation).pack(side="left")
root.mainloop()
```

Colorized Labels in various fonts

Some Tk widgets, like the label, text, and canvas widget, allow you to specify the fonts used to display text. This can be achieved by setting the attribute "font". typically via a "font" configuration option. You have to consider that fonts are one of several areas that are not platform-independent.

```
import tkinter as tk

root = tk.Tk()
tk.Label(root,
                 text="Red Text in Times Font",
                 fg = "red",
                 font = "Times").pack()
tk.Label(root,
                 text="Green Text in Helvetica Font",
                 fg = "light green",
                 bg = "dark green",
                 font = "Helvetica 16 bold italic").pack()
tk.Label(root,
                 text="Blue Text in Verdana bold",
                 fg = "blue",
                 bg = "yellow",
                 font = "Verdana 10 bold").pack()

root.mainloop()
```

Message Widget

The widget can be used to display short text messages. The message widget is similar in its functionality to the Label widget, but it is more flexible in displaying text, e.g. the font can be changed while the Label widget can only display text in a single font. It provides a multiline object, that is the text may span more than one line. The text is automatically broken into lines and justified.

```
import tkinter as tk
master = tk.Tk()
whatever_you_do = "Whatever you do will be insignificant"
msg = tk.Message(master, text = whatever_you_do)
msg.config(bg='lightgreen', font=('times', 24, 'italic'))
msg.pack()
tk.mainloop()
```

Tkinter Buttons

The Button widget is a standard Tkinter widget, which is used for various kinds of buttons. A button is a widget which is designed for the user to interact with, i.e. if the button is pressed by mouse click some action might be started. They can also contain text and images like labels. While labels can display text in various fonts, a button can only display text in a single font. The text of a button can span more than one line.

```
import tkinter as tk


def write_slogan():
    print("Tkinter is easy to use!")

root = tk.Tk()
frame = tk.Frame(root)
frame.pack()

button = tk.Button(frame,
                   text="QUIT",
                   fg="red",
                   command=quit)
button.pack(side=tk.LEFT)
slogan = tk.Button(frame,
                   text="Hello",
                   command=write_slogan)
slogan.pack(side=tk.LEFT)

root.mainloop()
```

Variable Classes

Some widgets (like text entry widgets, radio buttons and so on) can be connected directly to application variables by using special options: variable, textvariable, onvalue, offvalue, and value. This connection works both ways: if the variable changes for any reason, the widget it's connected to will be updated to reflect the new value. These Tkinter control variables are used like regular Python variables to keep certain values.

- x = StringVar() # Holds a string; default value ""
- x = IntVar() # Holds an integer; default value 0
- x = DoubleVar() # Holds a float; default value 0.0
- x = BooleanVar() # Holds a boolean, returns 0 for False and 1 for True

Radio Button

A radio button, sometimes called option button, is a graphical user interface element of Tkinter, which allows the user to choose (exactly) one of a predefined set of options. Radio buttons can contain text or images. The button can only display text in a single font. A Python function or method can be associated with a radio button. This function or method will be called, if you press this radio button.

```
from tkinter import *

def sel():
   selection = "You selected the option " + str(var.get())
   label.config(text = selection)

root = Tk()
var = IntVar()
R1 = Radiobutton(root, text = "Option 1", variable = var, value = 1,
                  command = sel)
R1.pack()

R2 = Radiobutton(root, text = "Option 2", variable = var, value = 2,
                  command = sel)
R2.pack()

R3 = Radiobutton(root, text = "Option 3", variable = var, value = 3,
                  command = sel)
R3.pack()

label = Label(root)
label.pack()
root.mainloop()
```

Check Box:

```
from tkinter import *
master = Tk()

def var_states():
   print("male: %d,\nfemale: %d" % (var1.get(), var2.get()))

Label(master, text="Gender:").grid(row=0)
var1 = IntVar()
Checkbutton(master, text="male", variable=var1).grid(row=1)
var2 = IntVar()
Checkbutton(master, text="female", variable=var2).grid(row=2)
Button(master, text='Quit', command=master.quit).grid(row=3)
Button(master, text='Show', command=var_states).grid(row=4)
mainloop()
```

Entry Widgets:

Entry widgets are the basic widgets of Tkinter used to get input, i.e. text strings, from the user of an application. This widget allows the user to enter a single line of text. If the user enters a string, which is longer than the available display space of the widget, the content will be scrolled. This means that the string cannot be seen in its entirety. The arrow keys can be used to move to the invisible parts of the string. If you want to enter multiple lines of text, you have to use the text widget. An entry widget is also limited to single font.

```
import tkinter as tk

def show_entry_fields():
    print("First Name: %s\nLast Name: %s" % (e1.get(), e2.get()))

master = tk.Tk()
tk.Label(master,
         text="First Name").grid(row=0)
tk.Label(master,
         text="Last Name").grid(row=1)

e1 = tk.Entry(master)
e2 = tk.Entry(master)

e1.grid(row=0, column=1)
e2.grid(row=1, column=1)

tk.Button(master,
          text='Quit',
          command=master.quit).grid(row=3,
                                    column=0,
                                    sticky=tk.W,
                                    pady=4)
tk.Button(master,
          text='Show', command=show_entry_fields).grid(row=3,
                                                       column=1,
                                                       sticky=tk.W,
                                                       pady=4)

tk.mainloop()
```

Calculator:

```
import tkinter as tk
from math import *

def evaluate():
    res.configure(text = "Result: " + str(eval(entry.get())))

w = tk.Tk()
tk.Label(w, text="Your Expression:").pack()
entry = tk.Entry(w)
#entry.bind("", evaluate)
entry.pack()
res = tk.Label(w)
res.pack()
b1 = tk.Button(w, text='Show',
                  command=evaluate)
b1.pack(side=tk.LEFT, padx=5, pady=5)
b2 = tk.Button(w, text='Quit', command=w.quit)
b2.pack(side=tk.LEFT, padx=5, pady=5)
w.mainloop()
```

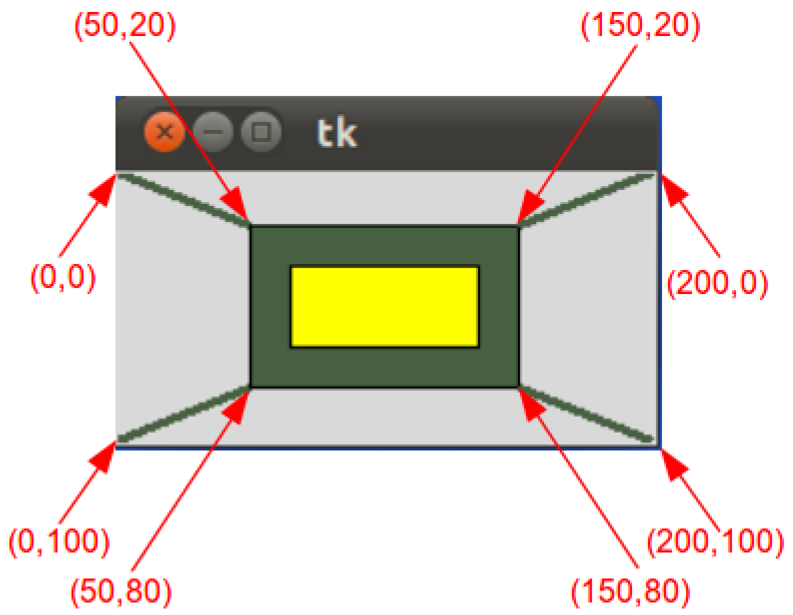
Canvas:

The Canvas widget supplies graphics facilities for Tkinter. Among these graphical objects are lines, circles, images, and even other widgets. With this widget it's possible to draw graphs and plots, create graphics editors, and implement various kinds of custom widgets.

```
from tkinter import *
master = Tk()

canvas_width = 80
canvas_height = 40
w = Canvas(master,
           width=canvas_width,
           height=canvas_height)
w.pack()

y = int(canvas_height / 2)
w.create_line(0, y, canvas_width, y, fill="#476042")


mainloop()
```

Rectangle:

```
from tkinter import *

master = Tk()

w = Canvas(master, width=200, height=100)
w.pack()

w.create_rectangle(50, 20, 150, 80, fill="#476042")
w.create_rectangle(65, 35, 135, 65, fill="yellow")
w.create_line(0, 0, 50, 20, fill="#476042", width=3)
w.create_line(0, 100, 50, 80, fill="#476042", width=3)
w.create_line(150,20, 200, 0, fill="#476042", width=3)
w.create_line(150, 80, 200, 100, fill="#476042", width=3)

mainloop()
```



Slider

A slider is a Tkinter object with which a user can set a value by moving an indicator. Sliders can be vertically or horizontally arranged. A slider is created with the Scale method().

```
from tkinter import *

def show_values():
    print (w1.get(), w2.get())

master = Tk()
w1 = Scale(master, from_=0, to=42)
w1.pack()
w2 = Scale(master, from_=0, to=200, orient=HORIZONTAL)
w2.pack()
Button(master, text='Show', command=show_values).pack()

mainloop()
```

Text Widgets

A text widget is used for multi-line text area. The tkinter text widget is very powerful and flexible and can be used for a wide range of tasks. Though one of the main purposes is to provide simple multi-line areas, as they are often used in forms, text widgets can also be used as simple text editors or even web browsers.

```
import tkinter as tk

root = tk.Tk()
T = tk.Text(root, height=2, width=30)
T.pack()
T.insert(tk.END, "Just a text Widget\nin two lines\n")
tk.mainloop()
```

Dialogue Box

Tkinter (and TK of course) provides a set of dialogues (dialogs in American English spelling), which can be used to display message boxes, showing warning or errors, or widgets to select files and colours. There are also simple dialogues, asking the user to enter string, integers or float numbers.

```
import tkinter as tk
from tkinter import messagebox as mb

def answer():
    mb.showerror("Answer", "Sorry, no answer available")

def callback():
    if mb.askyesno('Verify', 'Really quit?'):
        mb.showwarning('Yes', 'Not yet implemented')
    else:
        mb.showinfo('No', 'Quit has been cancelled')

tk.Button(text='Quit', command=callback).pack()
tk.Button(text='Answer', command=answer).pack()
tk.mainloop()
```

Open file Dialogue

```
import tkinter as tk
from tkinter import filedialog as fd

def callback():
    name= fd.askopenfilename()
    print(name)

errmsg = 'Error!'
tk.Button(text='File Open',
       command=callback).pack(fill=tk.X)
tk.mainloop()
```

Menus:

Menus in GUIs are presented with a combination of text and symbols to represent the choices. Selecting with the mouse (or finger on touch screens) on one of the symbols or text, an action will be started. Such an action or operation can, for example, be the opening or saving of a file, or the quitting or exiting of an application.

CGI (Common Gateway Interface)

What is CGI?

- The Common Gateway Interface, or CGI, is a standard for external gateway programs to interface with information servers such as HTTP servers.
- The current version is CGI/1.1 and CGI/1.2 is under progress.

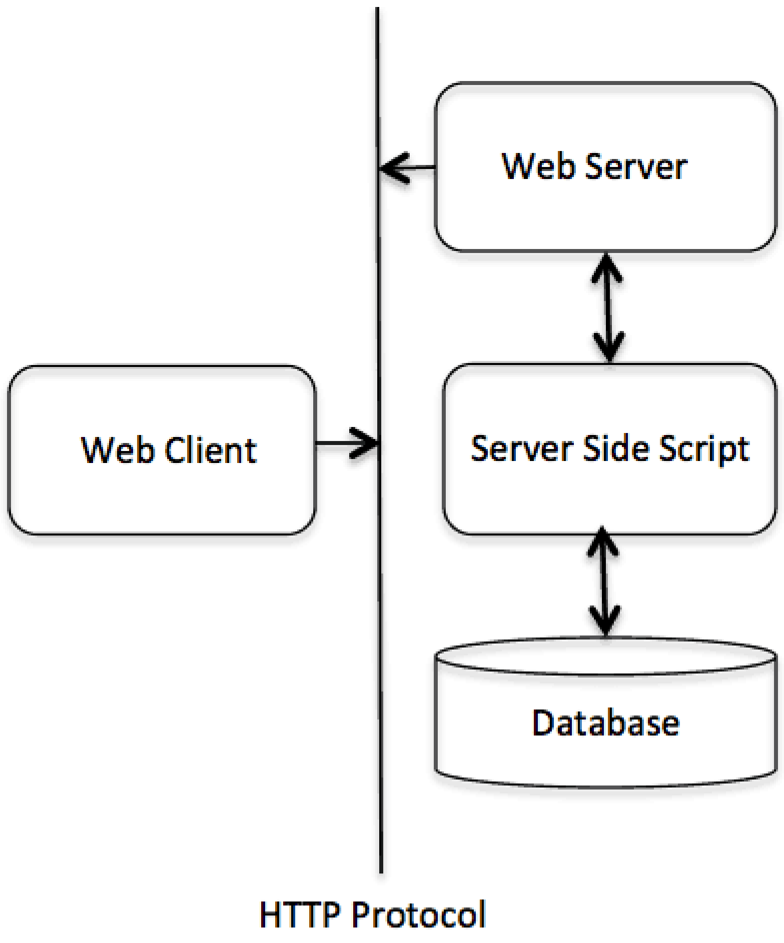
Web Browsing

To understand the concept of CGI, let us see what happens when we click a hyper link to browse a particular web page or URL.

- Your browser contacts the HTTP web server and demands for the URL, i.e., filename.
- Web Server parses the URL and looks for the filename. If it finds that file then sends it back to the browser, otherwise sends an error message indicating that you requested a wrong file.
- Web browser takes response from web server and displays either the received file or error message.

However, it is possible to set up the HTTP server so that whenever a file in a certain directory is requested that file is not sent back; instead it is executed as a program, and whatever that program outputs is sent back for your browser to display. This function is called the Common Gateway Interface or CGI and the programs are called CGI scripts. These CGI programs can be a Python Script, PERL Script, Shell Script, C or C++ program, etc.

CGI Architecture Diagram



Web Server Support and Configuration

Before you proceed with CGI Programming, make sure that your Web Server supports CGI and it is configured to handle CGI Programs. All the CGI Programs to be executed by the HTTP server are kept in a pre-configured directory. This directory is called CGI Directory and by convention it is named as /var/www/cgi-bin. By convention, CGI files have extension as. cgi, but you can keep your files with python extension .py as well.

First CGI Program

```
#!C:/Users/aspl/AppData/Local/Programs/Python/Python37-32/python

print ("Content-type:text/html\r\n\r\n")
print ('<html>')
print ('<head>')
print ('<title>Hello Word - First CGI Program</title>')
print ('</head>')
print ('<body>')
print ('<h2>Hello Word! This is my first CGI program</h2>')
print ('</body>')
print ('</html>')
```

GET and POST Methods

You must have come across many situations when you need to pass some information from your browser to web server and ultimately to your CGI Program. Most frequently, browser uses two methods two pass this information to web server. These methods are GET Method and POST Method.

Passing Information using GET method

The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the ? character as follows –

- The GET method is the default method to pass information from the browser to the web server and it produces a long string that appears in your browser's Location:box.
- Never use GET method if you have password or other sensitive information to pass to the server.
- The GET method has size limtation: only 1024 characters can be sent in a request string.

- The GET method sends information using QUERY_STRING header and will be accessible in your CGI Program through QUERY_STRING environment variable.

Simple URL Example: Get Method

Below is hello_get.py script to handle input given by web browser. We are going to use cgi module, which makes it very easy to access passed information

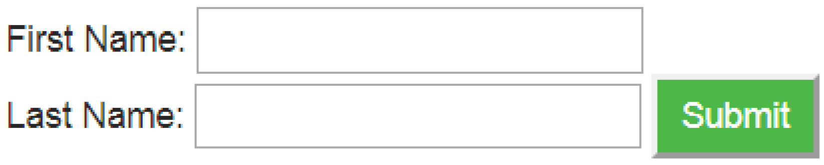
```
#!C:/Users/aspl/AppData/Local/Programs/Python/Python37-32/python

import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
first_name = form.getvalue('first_name')
last_name  = form.getvalue('last_name')

print ("Content-type:text/html\r\n\r\n")
print ("<html>")
print ("<head>")
print ("<title>Hello - Second CGI Program</title>")
print ("</head>")
print ("<body>")
print ("<h2>Hello %s %s</h2>" % (first_name, last_name))
print ("</body>")
print ("</html>")
```

Simple FORM Example: GET Method

This example passes two values using HTML FORM and submit button. We use same CGI script hello_get.py to handle this input.

```
<form action = "/cgi-bin/hello_get.py" method = "get">

First Name: <input type = "text" name = "first_name">
<br />


Last Name: <input type = "text" name = "last_name" />

<input type = "submit" value = "Submit" />

</form>
```

Here is the actual output of the above form, you enter First and Last Name and then click submit button to see the result.



Passing Checkbox Data to CGI Program

Checkboxes are used when more than one option is required to be selected.

Here is example HTML code for a form with two checkboxes –

```
<form action = "/cgi-bin/checkbox.cgi"
method = "POST" target = "_blank">

<input type = "checkbox" name = "maths"
value = "on" /> Maths

<input type = "checkbox" name = "physics"
value = "on" /> Physics
```

```
<input type = "submit" value = "Select
Subject" />
</form>
```

Below is checkbox.cgi script to handle input given by web browser for checkbox button.

```
#!C:/Users/aspl/AppData/Local/Programs/Python/Python37-32/python

import cgi, cgitb

import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
if form.getvalue('maths'):
   math_flag = "ON"
else:
   math_flag = "OFF"

if form.getvalue('physics'):
   physics_flag = "ON"
else:
   physics_flag = "OFF"

print ("Content-type:text/html\r\n\r\n")
print ("<html>")
print ("<head>")
print ("<title>Checkbox - Third CGI Program</title>")
print ("</head>")
print ("<body>")
print ("<h2> CheckBox Maths is : %s</h2>" % math_flag)
print ("<h2> CheckBox Physics is : %s</h2>" % physics_flag)
print ("</body>")
print ("</html>")
```

Database:

Python 3 or higher version install using pip3 as:

pip3 install mysql-connector

Test the MySQL Database connection with Python

```
import mysql.connector
        db_connection = mysql.connector.connect(
        host="localhost",
        user="root",
        passwd=""
    )
```

Creating Database in MySQL using Python

Syntax to Create new database in SQL is

CREATE DATABASE "database_name"

```
import mysql.connector
        db_connection = mysql.connector.connect(
        host="localhost",
        user="root",
        passwd=""
    )
db_cursor = db_connection.cursor()
db_cursor.execute("CREATE DATABASE my_first_db")
db_cursor.execute("SHOW DATABASES")
for db in db_cursor:
        print(db)
```

Create a Table in MySQL with Python

Let's create a simple table "student" which has two columns.

SQL Syntax:

CREATE TABLE student (id INT, name VARCHAR(255))

```
import mysql.connector
        db_connection = mysql.connector.connect(
        host="localhost",
        user="root",
        passwd=""
    )
db_cursor = db_connection.cursor()
db_cursor.execute("CREATE TABLE student (id INT, name VARCHAR(255))")
db_cursor.execute("SHOW TABLES")
for table in db_cursor:
        print(table)
```

Create a Table with Primary Key

Let's create an Employee table with three different columns. We will add a primary key in id column with AUTO_INCREMENT constraint

SQL Syntax,

CREATE TABLE employee(id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255), salary INT(6))

```
import mysql.connector
        db_connection = mysql.connector.connect(
        host="localhost",
        user="root",
        passwd=""
    )
db_cursor = db_connection.cursor()
db_cursor.execute("""CREATE TABLE employee(id INT AUTO_INCREMENT PRIMARY KEY,
                  name VARCHAR(255), salary INT(6))""")
db_cursor.execute("SHOW TABLES")
for table in db_cursor:
        print(table)
```

Insert Operation with MySQL in Python:

Let's perform insertion operation in MySQL Database table which we already create. We will insert data oi STUDENT table and EMPLOYEE table.

SQL Syntax,

INSERT INTO student (id, name) VALUES (01, "John")

INSERT INTO employee (id, name, salary) VALUES(01, "John", 10000)

```
import mysql.connector
        db_connection = mysql.connector.connect(
        host="localhost",
        user="root",
        passwd=""
        database="student"
    )
db_cursor = db_connection.cursor()
student_sql_query = "INSERT INTO student(id,name) VALUES(01, 'John')"
employee_sql_query = """INSERT INTO employee (id, name, salary)
                    VALUES (01, 'John', 10000)"""
db_cursor.execute(student_sql_query)
        db_cursor.execute(employee_sql_query)
db_connection.commit()
print(db_cursor.rowcount, "Record Inserted")
```

Select From a Table

To select from a table in MySQL, use the "SELECT" statement:

```
import mysql.connector
        db_connection = mysql.connector.connect(
        host="localhost",
        user="root",
        passwd=""
        database="student"
    )
mycursor = mydb.cursor()

mycursor.execute("SELECT * FROM customers")

myresult = mycursor.fetchall()

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

Select With a Filter

When selecting records from a table, you can filter the selection by using the "WHERE" statement:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  passwd="",
  database="student"
)

mycursor = mydb.cursor()

sql = "SELECT * FROM customers WHERE address ='Park Lane 38'"

mycursor.execute(sql)

myresult = mycursor.fetchall()

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

Update Table

You can update existing records in a table by using the "UPDATE" statement:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  passwd="",
  database="student"
)
mycursor = mydb.cursor()

sql = "UPDATE customers SET address = 'Canyon 123' WHERE address = 'Valley 345'"

mycursor.execute(sql)

mydb.commit()

print(mycursor.rowcount, "record(s) affected")
```

Delete Record

You can delete records from an existing table by using the "DELETE FROM" statement:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  passwd="",
  database="student"
)
mycursor = mydb.cursor()

sql = "DELETE FROM customers WHERE address = 'Mountain 21'"

mycursor.execute(sql)

mydb.commit()

print(mycursor.rowcount, "record(s) deleted")
```

Delete a Table

You can delete an existing table by using the "DROP TABLE" statement:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  passwd="",
  database="student"
)
mycursor = mydb.cursor()

sql = "DROP TABLE customers"

mycursor.execute(sql)
```