Programming with Python

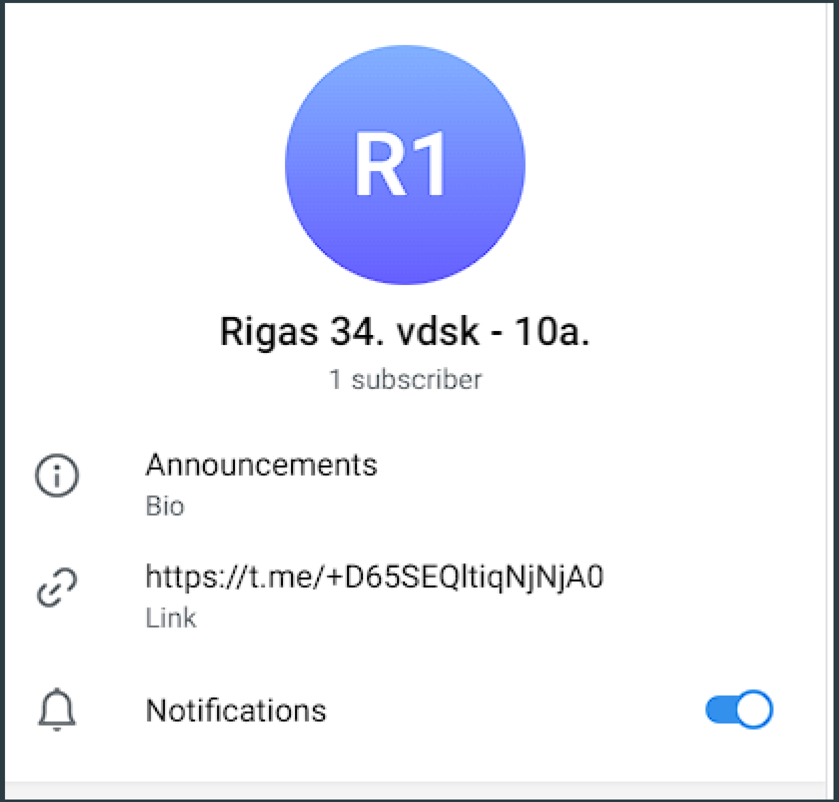
Konstantins Tarasjuks

Agenda

- **Variable naming**
- Variable types
- Functions aka methods

TELEGRAM

- https://t.me/+D65SEQltiqNjNjA0


R1

Rigas 34. vdsk - 10a.

1 subscriber

Announcements

Bio

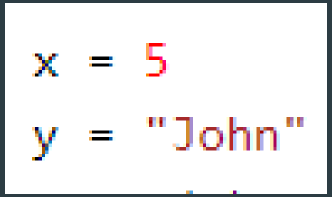
https://t.me/+D65SEQltiqNjNjA0

Link

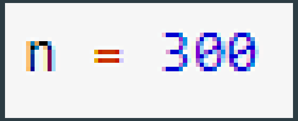
Notifications

Python Variable

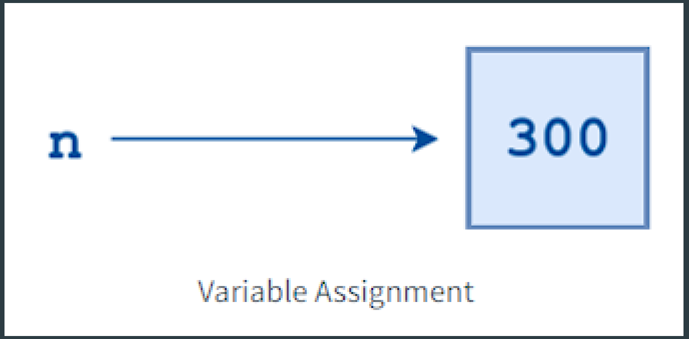
- Python has no command for declaring a variable.



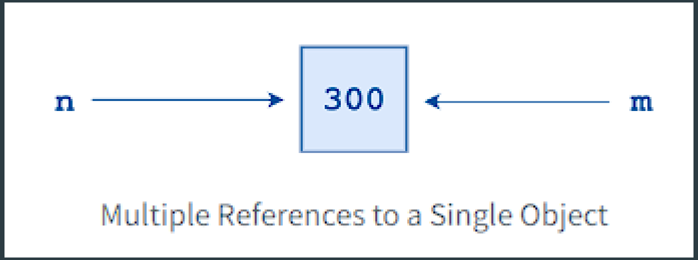
```
x = 5
y = "John"
```



```
n = 300
```



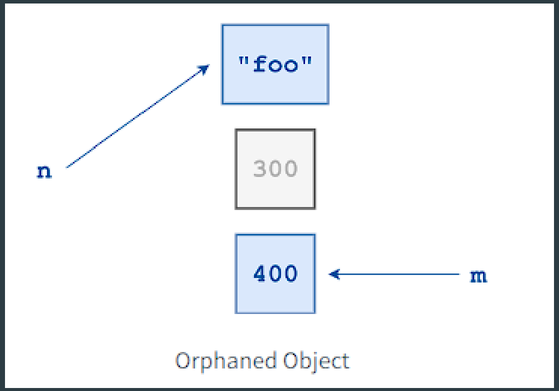
Variable Assignment



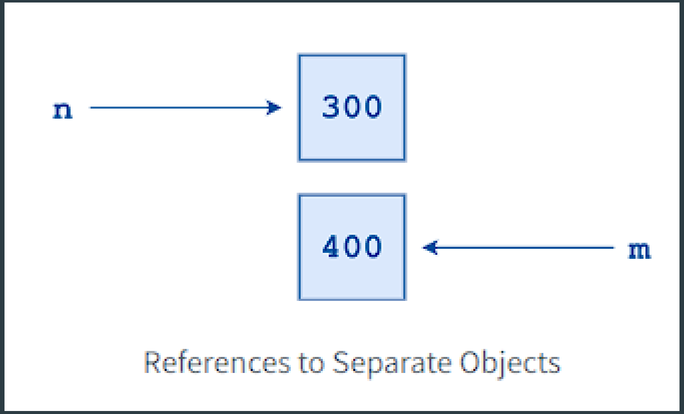
Multiple References to a Single Object

Python Variable

Variable reference



Orphaned Object



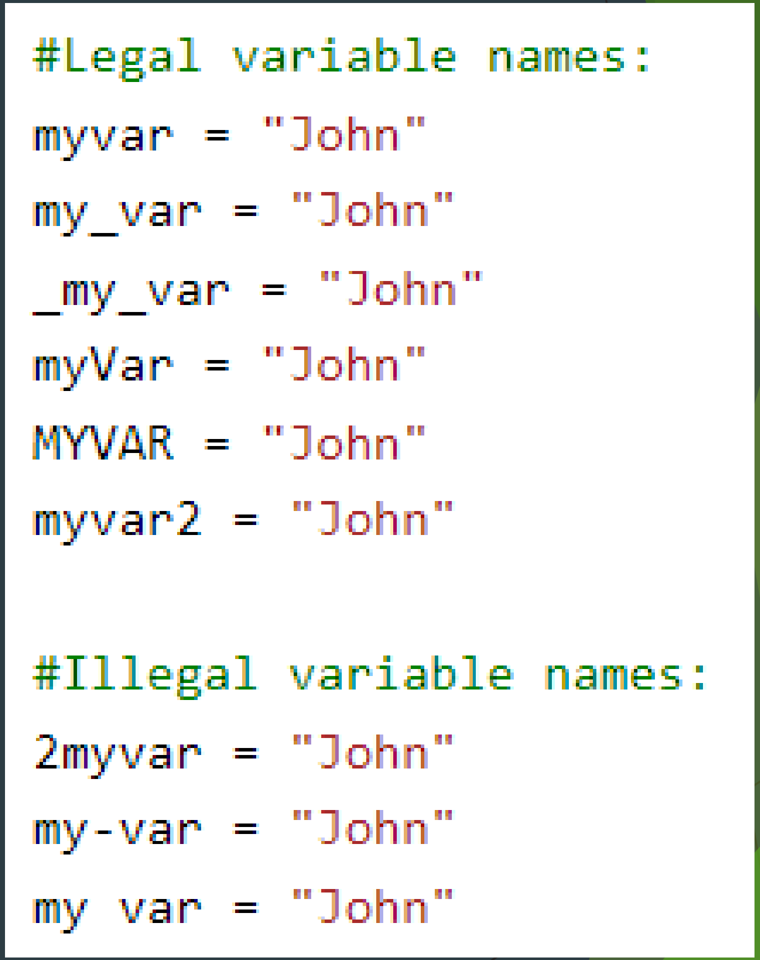
References to Separate Objects

Python Variable naming

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)



```
#Legal variable names:
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"

#Illegal variable names:
2myvar = "John"
my-var = "John"
my var = "John"
```

Method Names and Variables - PEP8

- ***Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability.***

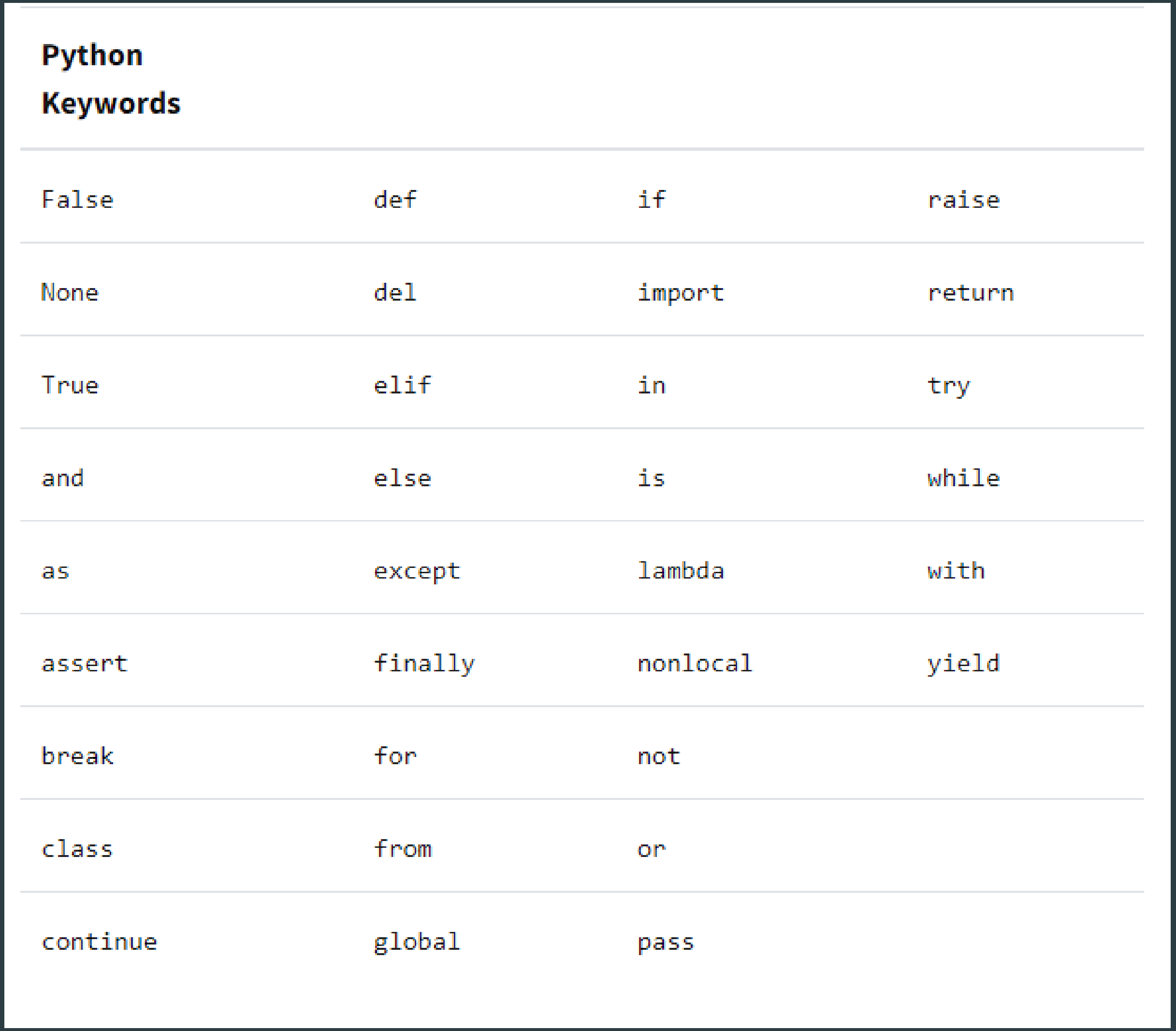
- Use one leading underscore only for non-public methods and instance variables.

- To avoid name clashes with subclasses, use two leading underscores to invoke Python’s name mangling rules.

- Python mangles these names with the class name: if class Foo has an attribute named __a, it cannot be accessed by Foo.__a. (An insistent user could still gain access by calling Foo._Foo__a.) Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed.

- Note: there is some controversy about the use of __names

Names that can not be used



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

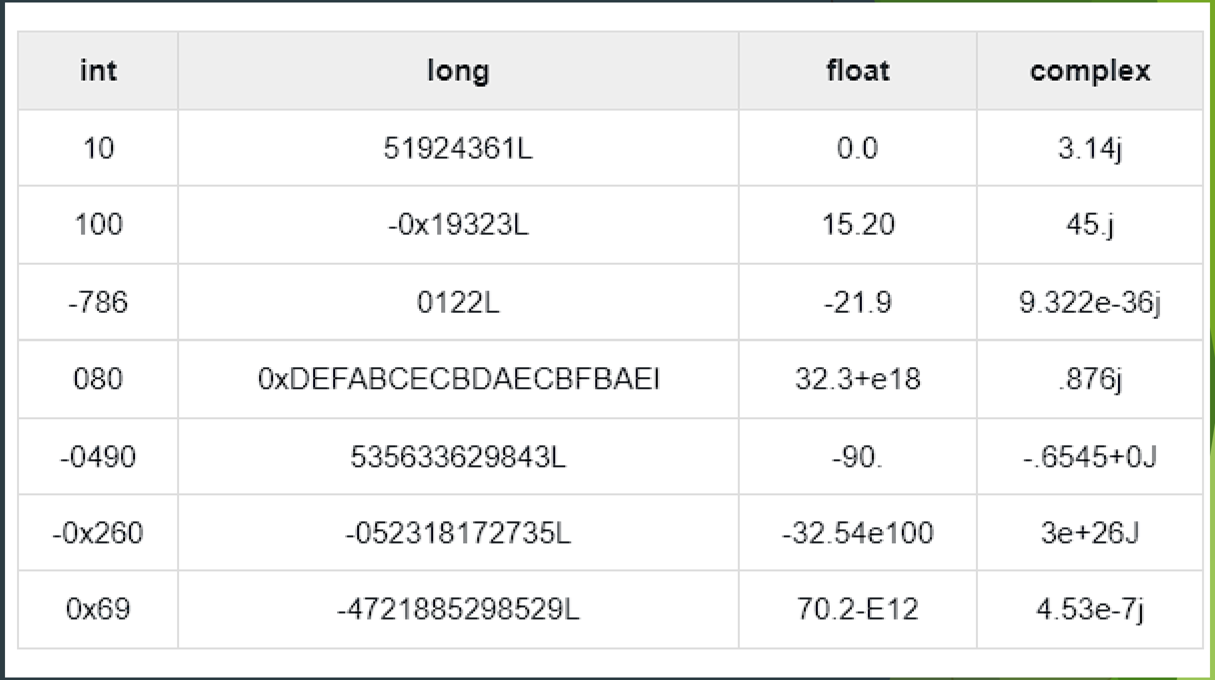
Agenda

- **Variable naming**
- **Variable types**
- Functions aka methods

Python Numeric Types

Python supports four different numerical types –

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
- complex (complex numbers)



int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEl	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

Python - Booleans

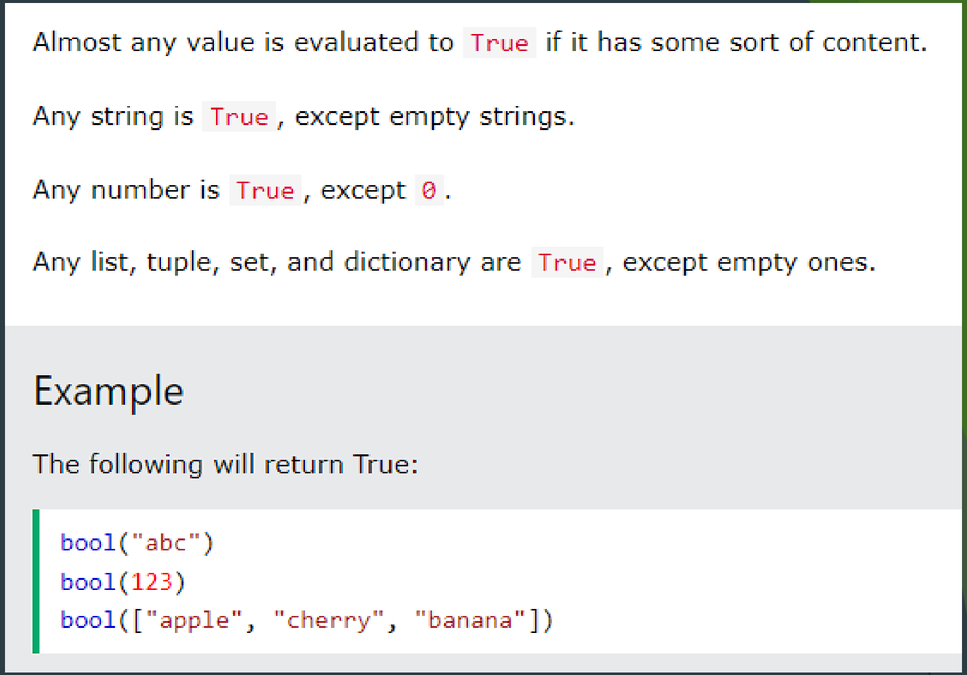
In programming you often need to know if an expression is True or False.

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

```
print(10 > 9)

print(10 == 9)

print(10 < 9)
```

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

Python Booleans



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

Any string is `True`, except empty strings.

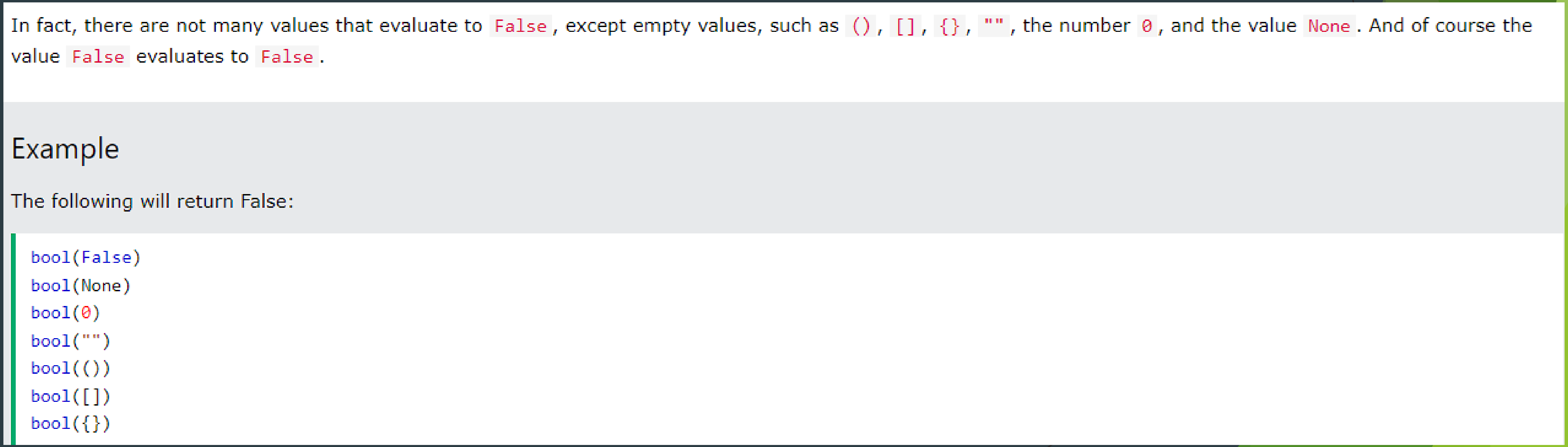
Any number is `True`, except `0`.

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

Example

The following will return True:

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



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

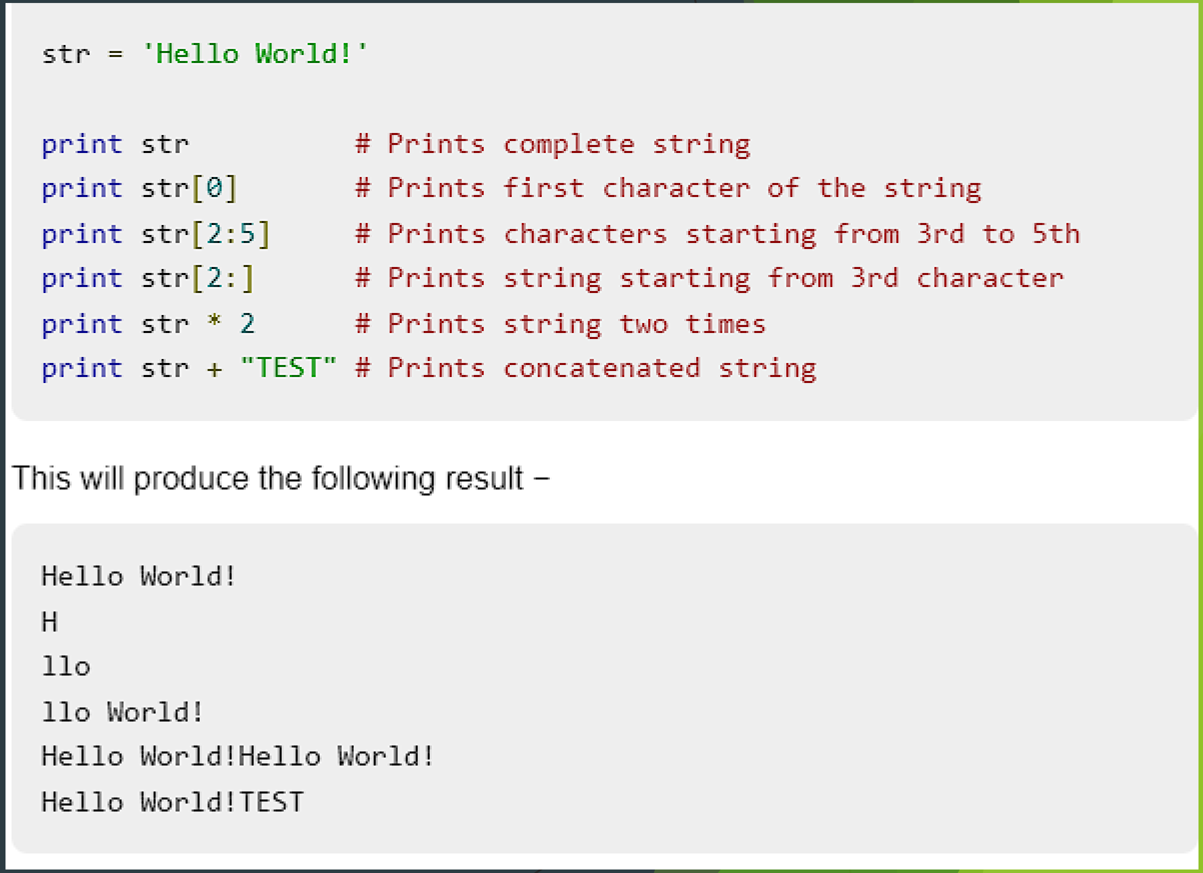
Example

The following will return False:

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

Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end. The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator.



```
str = 'Hello World!'

print str          # Prints complete string
print str[0]       # Prints first character of the string
print str[2:5]     # Prints characters starting from 3rd to 5th
print str[2:]      # Prints string starting from 3rd character
print str * 2      # Prints string two times
print str + "TEST" # Prints concatenated string
```

This will produce the following result –

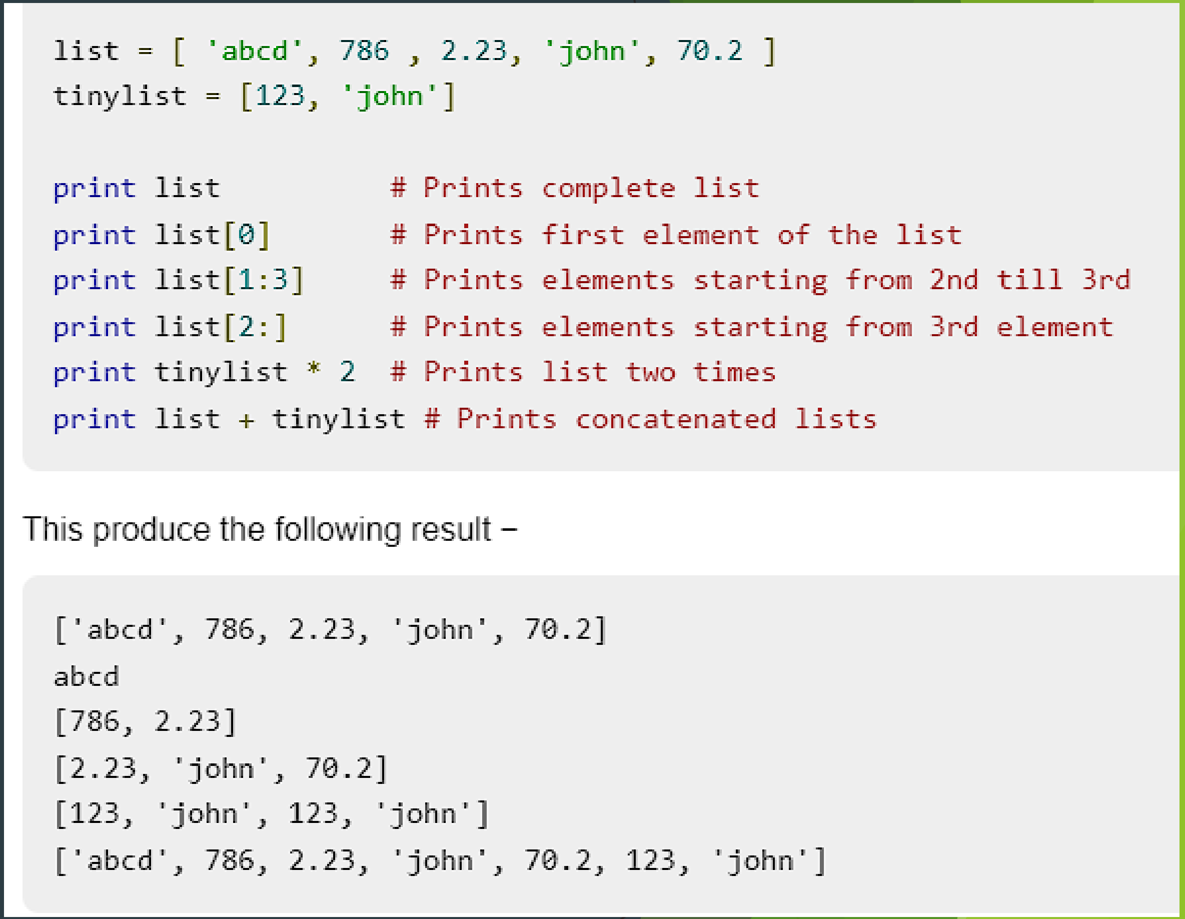
```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Python lists - Not ARRAY

Note: Python does not have built-in support for Arrays, but Python Lists can be used instead.

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator



```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print list          # Prints complete list
print list[0]       # Prints first element of the list
print list[1:3]     # Prints elements starting from 2nd till 3rd
print list[2:]      # Prints elements starting from 3rd element
print tinylist * 2  # Prints list two times
print list + tinylist # Prints concatenated lists
```

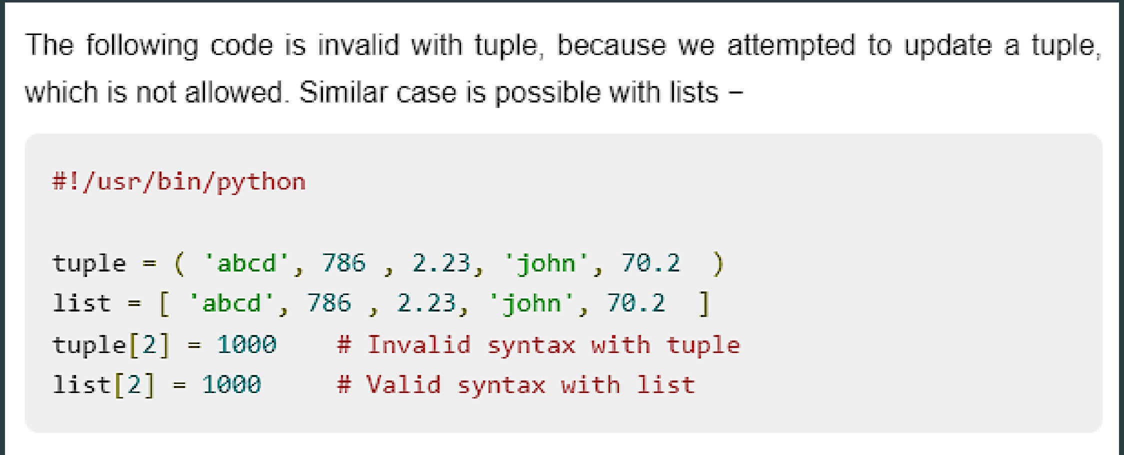
This produce the following result –

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

Python Tuples

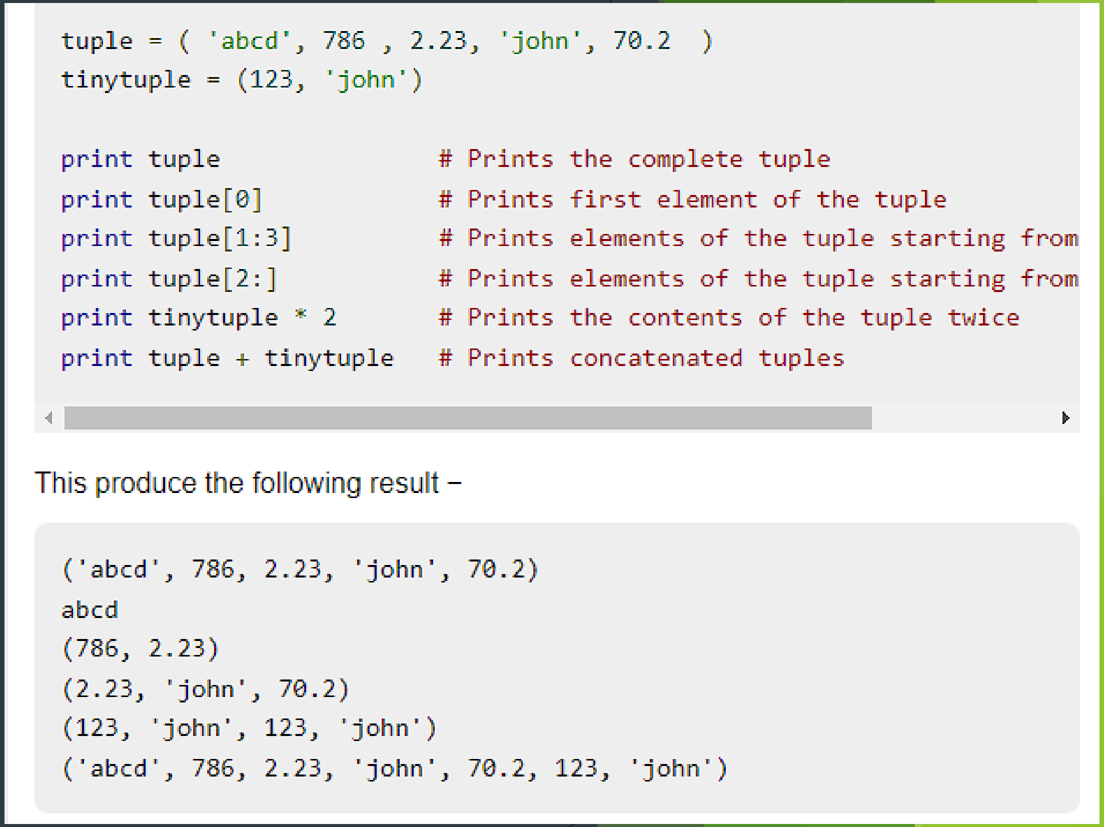
A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. ***Tuples can be thought of as read-only lists.***



The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists –

```
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```



```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

print tuple               # Prints the complete tuple
print tuple[0]            # Prints first element of the tuple
print tuple[1:3]          # Prints elements of the tuple starting from
print tuple[2:]           # Prints elements of the tuple starting from
print tinytuple * 2       # Prints the contents of the tuple twice
print tuple + tinytuple   # Prints concatenated tuples
```

This produce the following result –

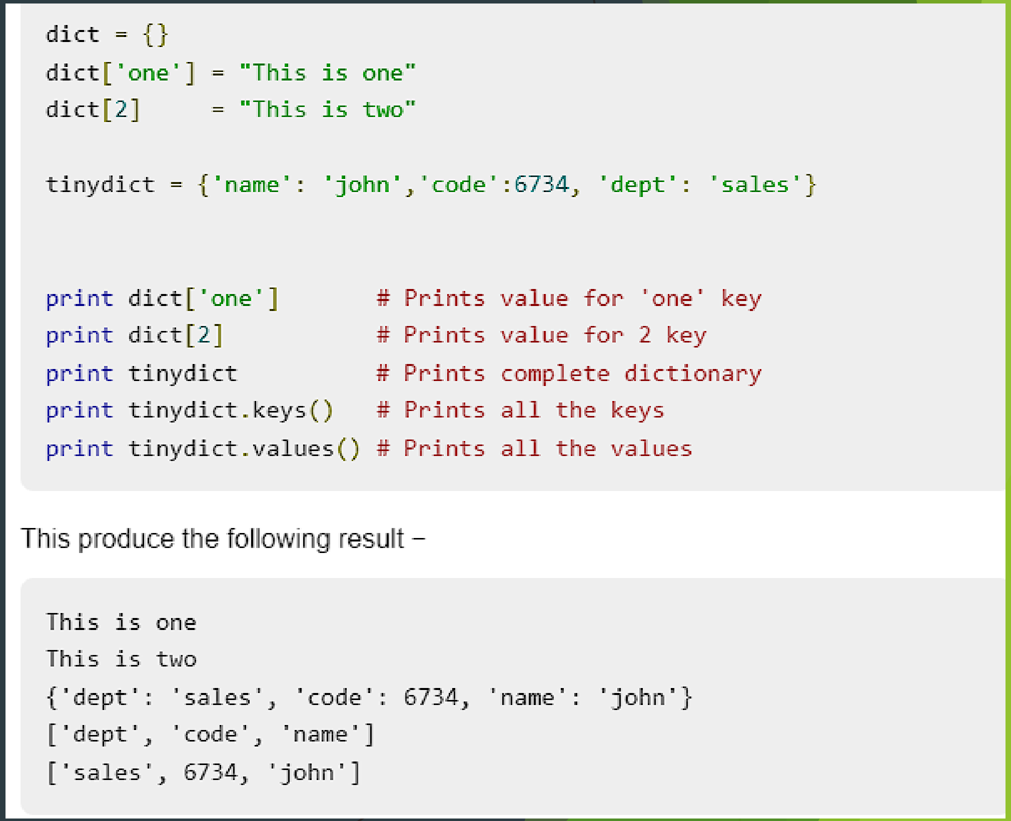
```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

Python Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

<u>QUESTION: </u>what will print code: *print(dict) ??????*



```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}


print dict['one']       # Prints value for 'one' key
print dict[2]           # Prints value for 2 key
print tinydict          # Prints complete dictionary
print tinydict.keys()   # Prints all the keys
print tinydict.values() # Prints all the values
```

This produce the following result –

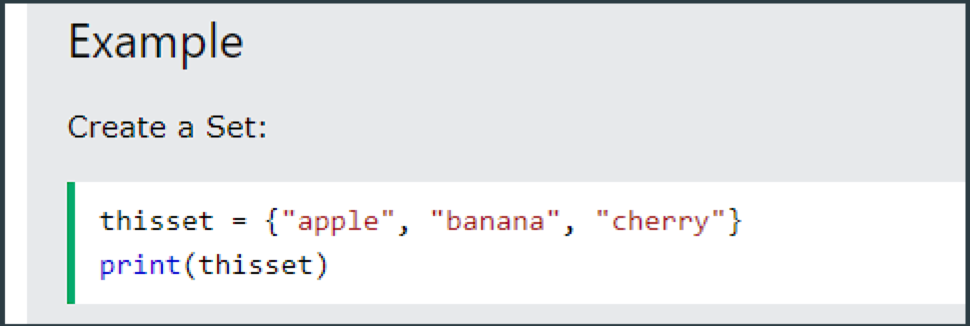
```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

Python sets

A set is a collection which is unordered, unchangeable*, and unindexed.

* Note: Set items are unchangeable, but you can remove items and add new items. Sets are unordered, so you cannot be sure in which order the items will appear.

No duplicates - only unique items



Example

Create a Set:

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

Differences list vs tuples vs sets vs dictionaries

List is a collection which is ordered and changeable. Allows duplicate members.

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

Tuple is a collection which is ordered and unchangeable. Allows duplicate members.

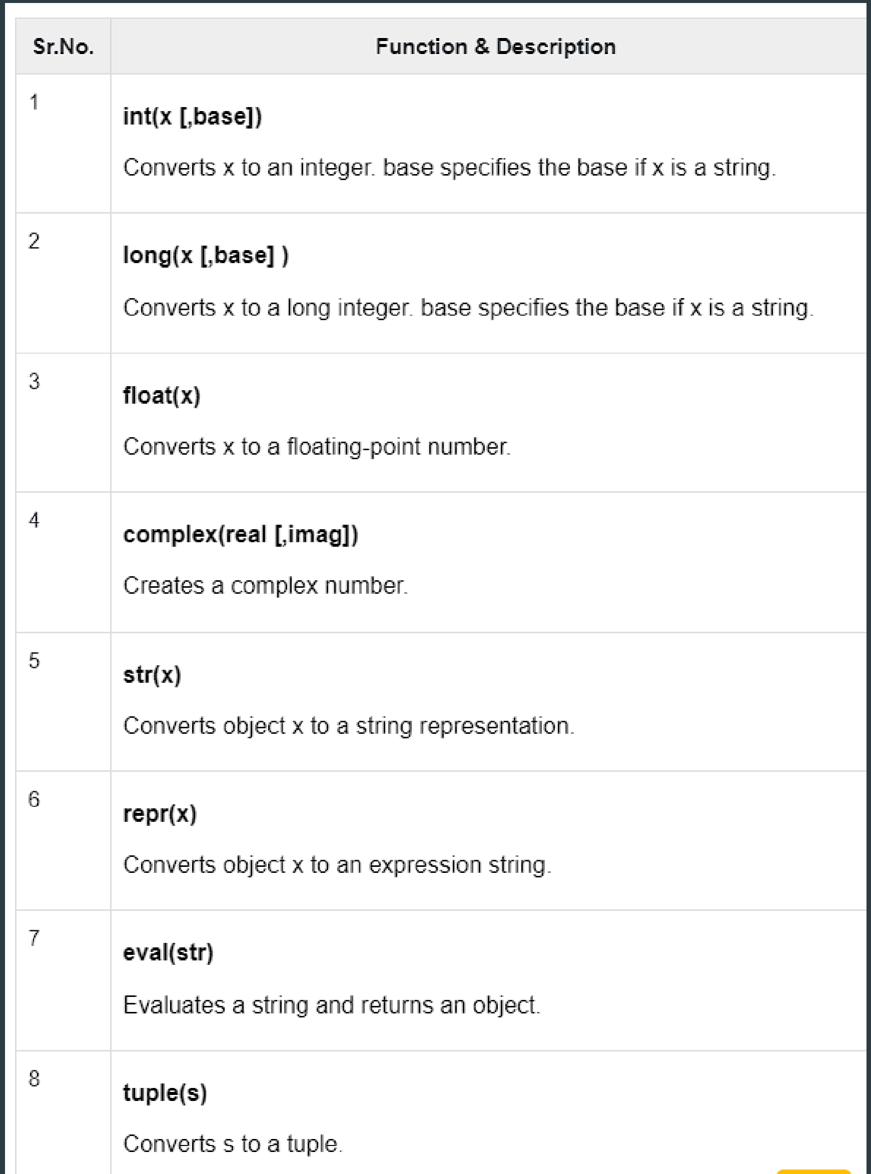
mytuple = ("apple", "banana", "cherry")

Set is a collection which is unordered, unchangeable (but you can remove and/or add items whenever you like), and unindexed. No duplicate members.

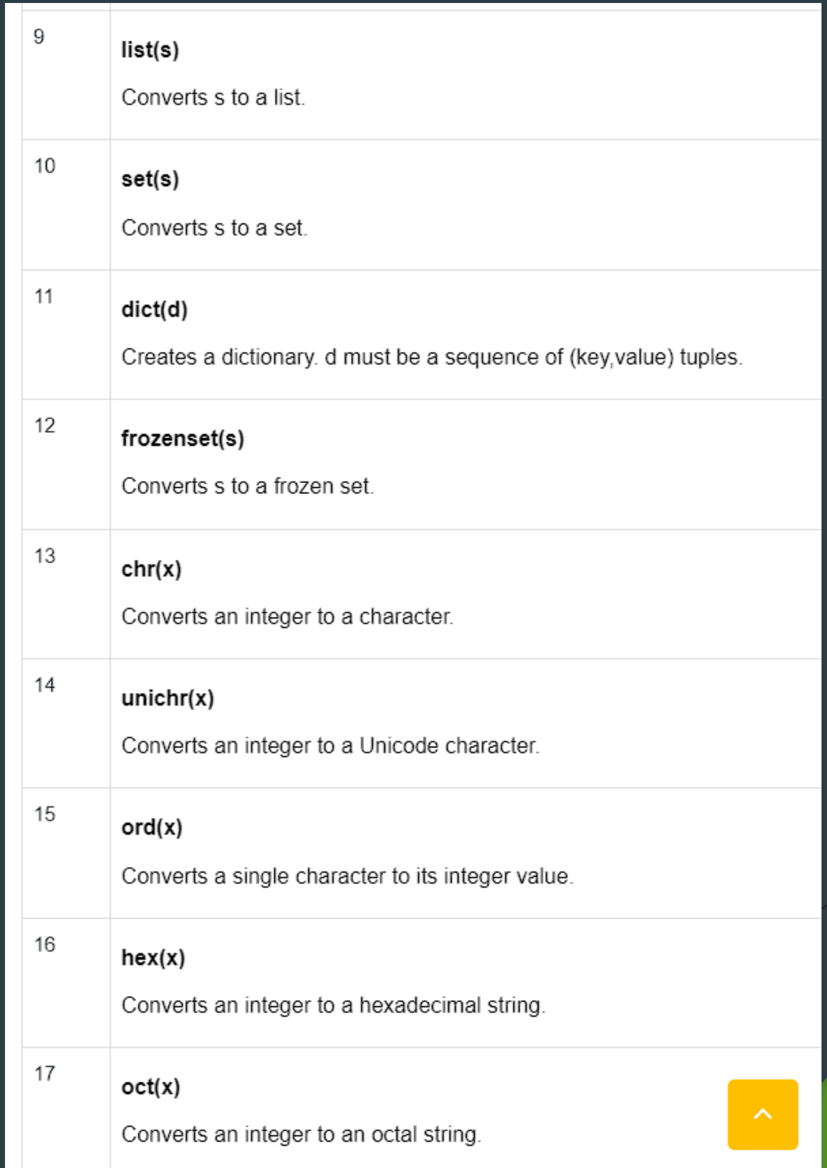
myset = {"apple", "banana", "cherry"}

Dictionary is a collection which is ordered (from Python 3.7) and changeable. No duplicate members. ***thisdict = { "brand": "Ford", "model": "Mustang", "year": 1964 }***

Python casting



Sr.No.	Function & Description
1	**int(x [,base])** Converts x to an integer. base specifies the base if x is a string.
2	**long(x [,base])** Converts x to a long integer. base specifies the base if x is a string.
3	**float(x)** Converts x to a floating-point number.
4	**complex(real [,imag])** Creates a complex number.
5	**str(x)** Converts object x to a string representation.
6	**repr(x)** Converts object x to an expression string.
7	**eval(str)** Evaluates a string and returns an object.
8	**tuple(s)** Converts s to a tuple.



9	**list(s)** Converts s to a list.
10	**set(s)** Converts s to a set.
11	**dict(d)** Creates a dictionary. d must be a sequence of (key,value) tuples.
12	**frozenset(s)** Converts s to a frozen set.
13	**chr(x)** Converts an integer to a character.
14	**unichr(x)** Converts an integer to a Unicode character.
15	**ord(x)** Converts a single character to its integer value.
16	**hex(x)** Converts an integer to a hexadecimal string.
17	**oct(x)** Converts an integer to an octal string.

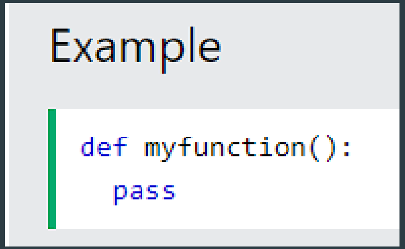
Agenda

- Variable naming
- Variable types
- Functions aka methods

Python methods/function

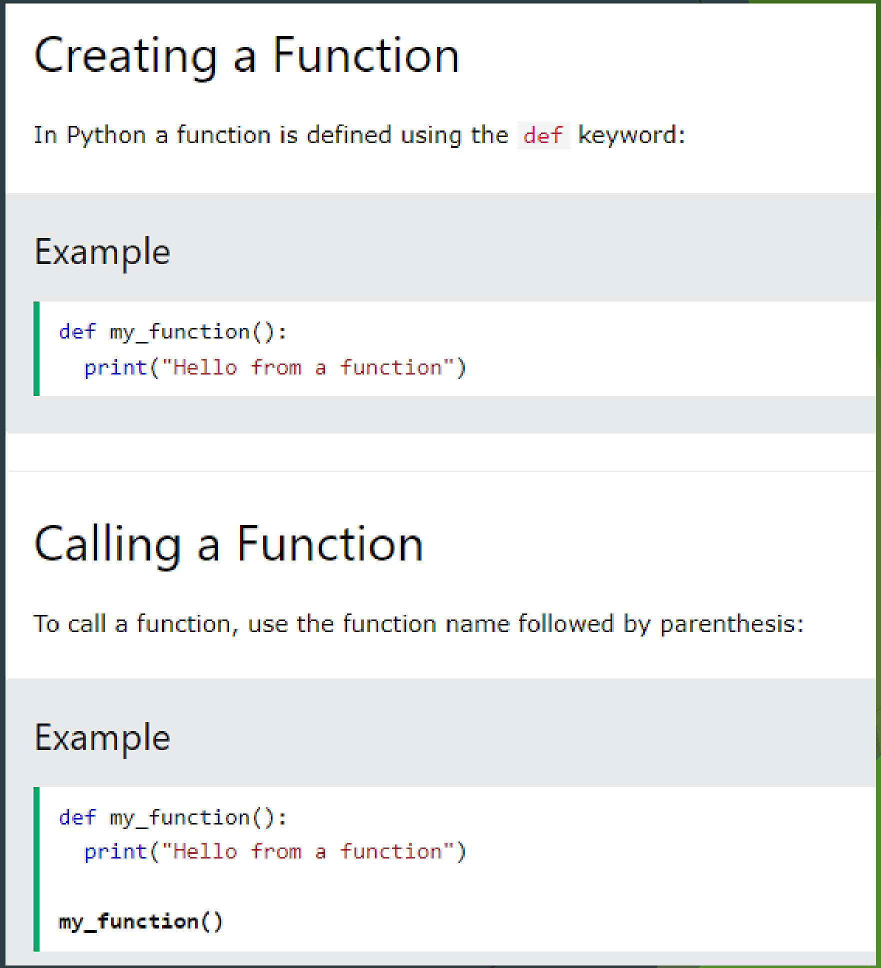
A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

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



Example

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



Creating a Function

In Python a function is defined using the `def` keyword:

Example

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

Calling a Function

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

Example

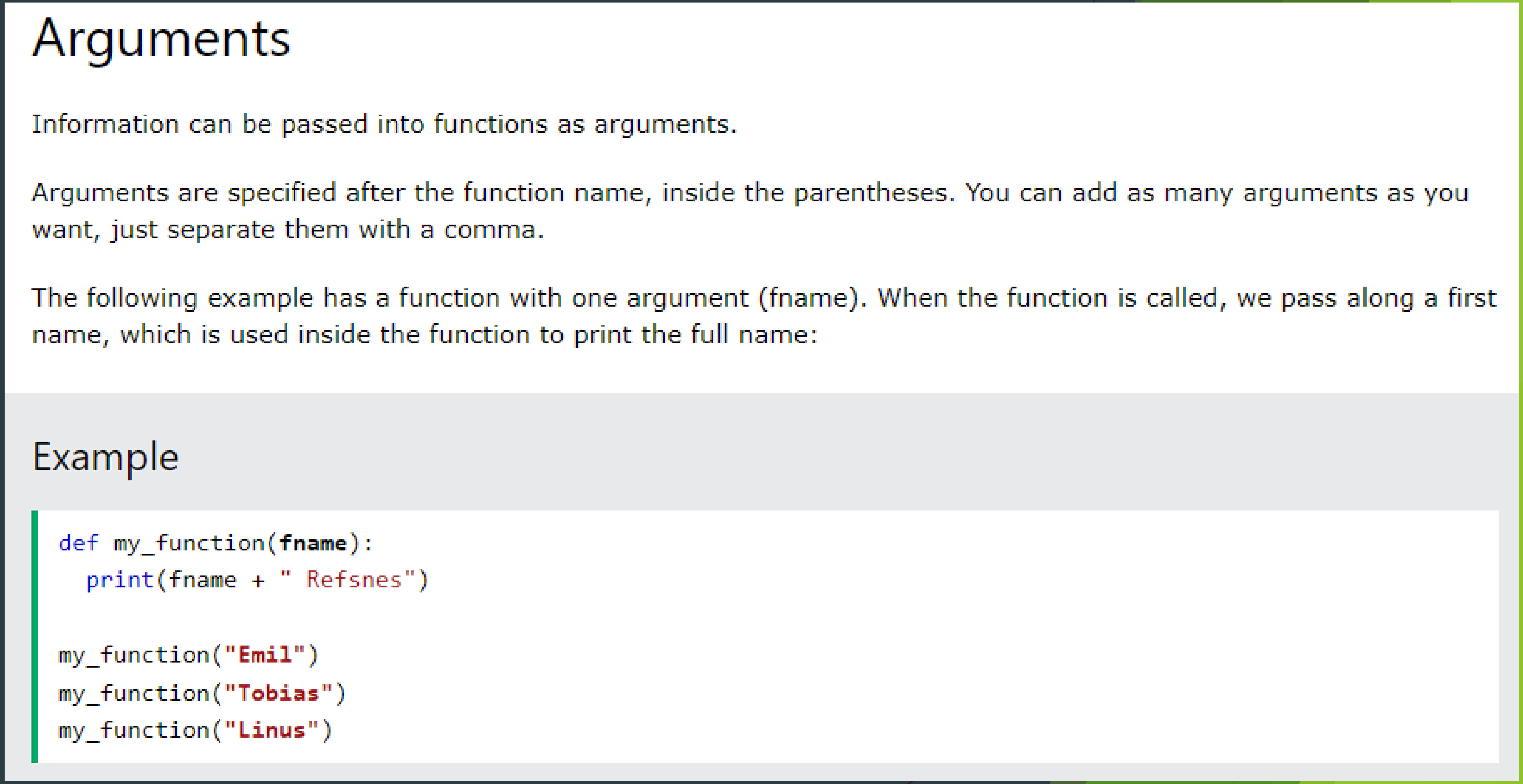
```
def my_function():
  print("Hello from a function")

my_function()
```

Python methods/function

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

From a function's perspective: A parameter is the variable listed inside the parentheses in the function definition. An argument is the value that is sent to the function when it is called.

Note:if function expects 2 arguments and you try to call the function with 1 or 3 arguments, you will get an error



Arguments

Information can be passed into functions as arguments.

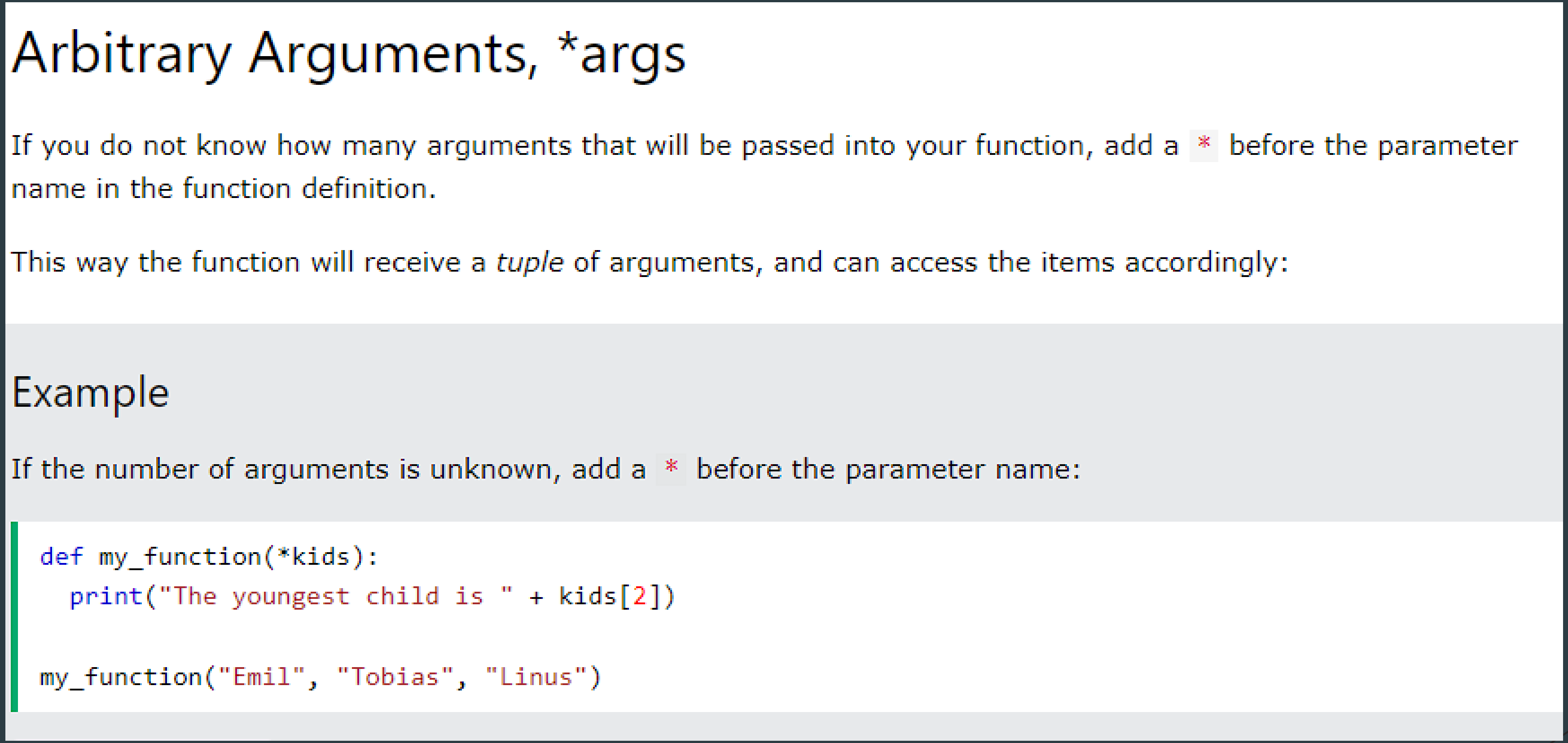
Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

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

Example

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

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

*args



Arbitrary Arguments, *args

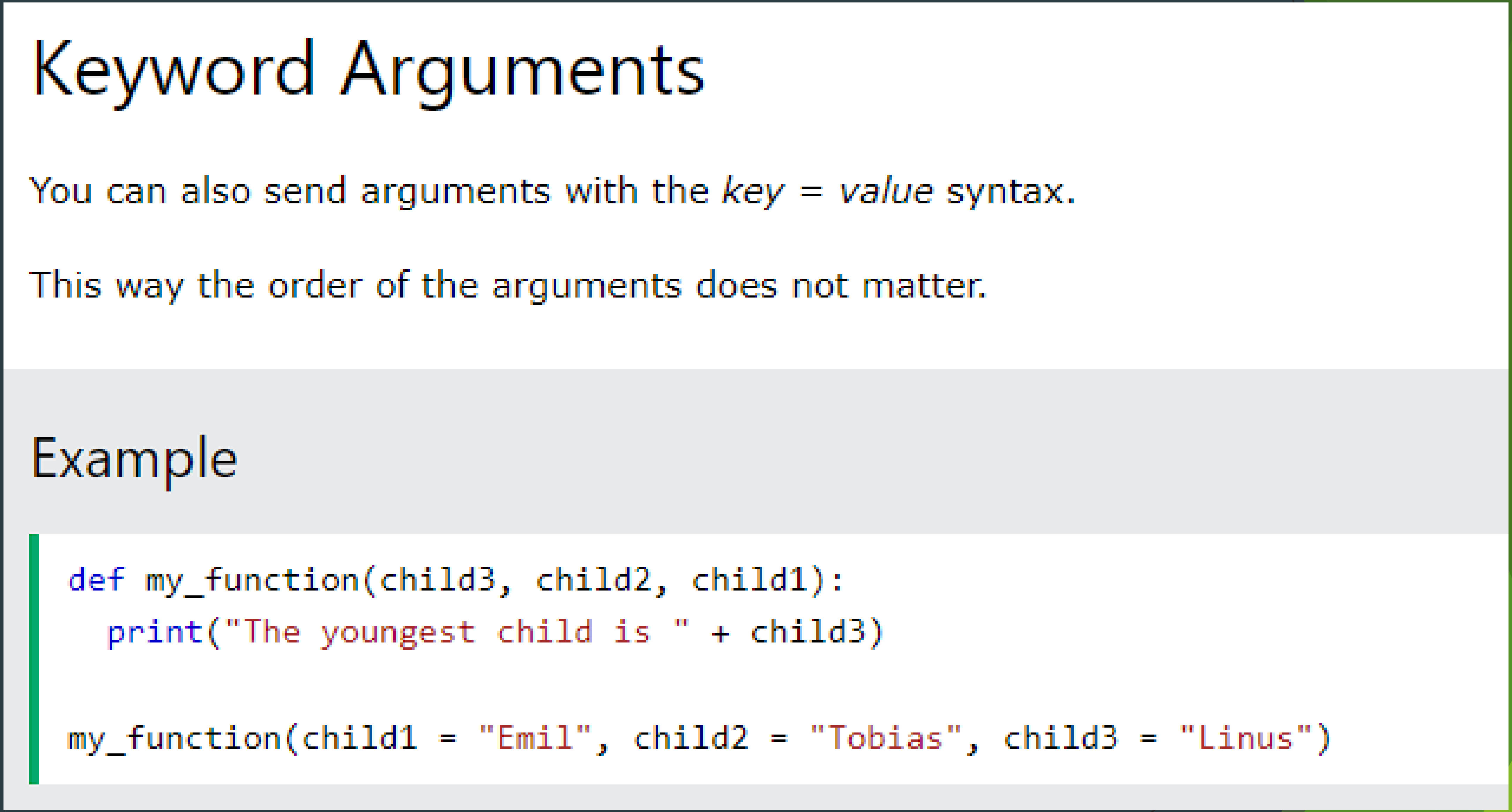
If you do not know how many arguments that will be passed into your function, add a `*` before the parameter name in the function definition.

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

Example

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

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

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



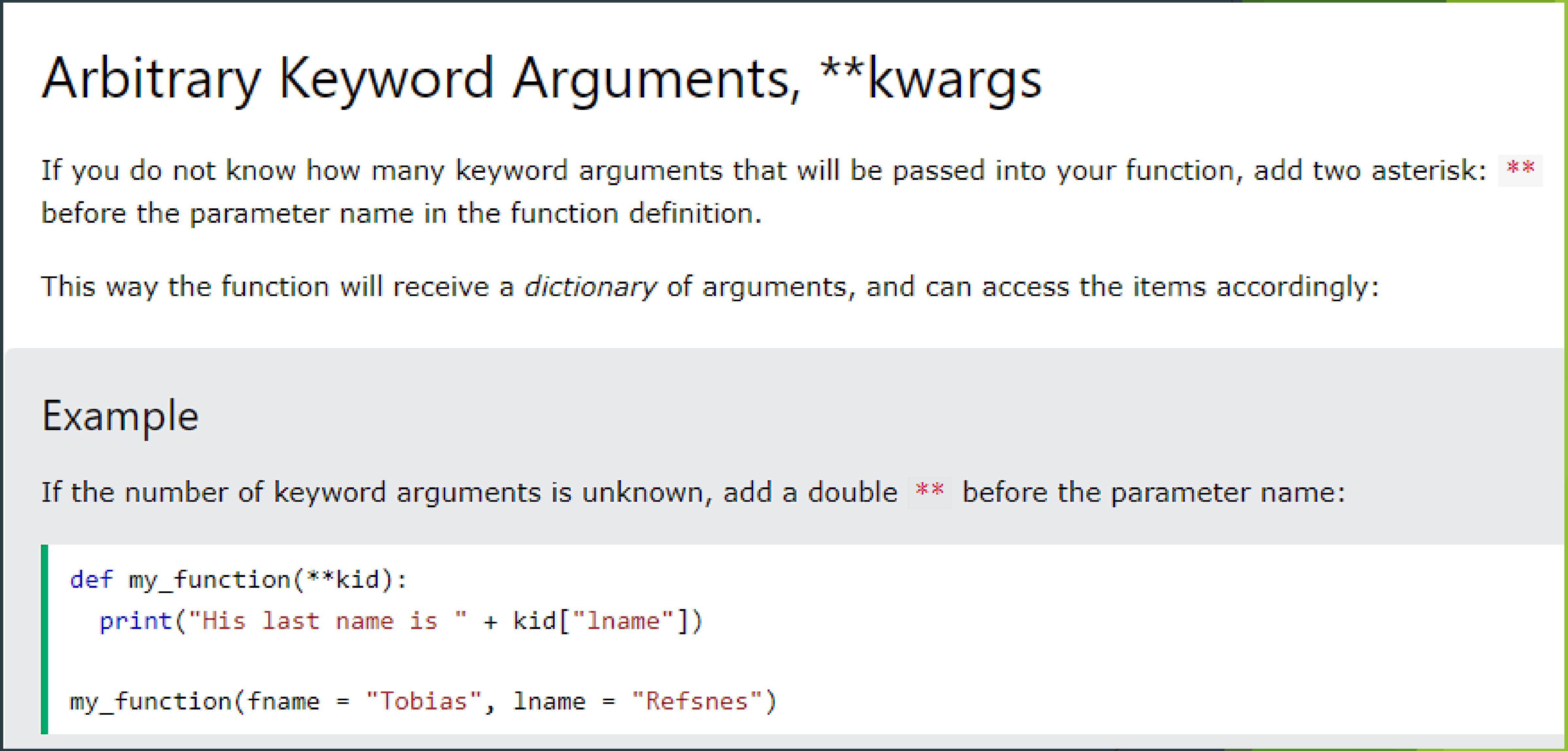
Keyword Arguments

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

This way the order of the arguments does not matter.

Example

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

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



Arbitrary Keyword Arguments, **kwargs

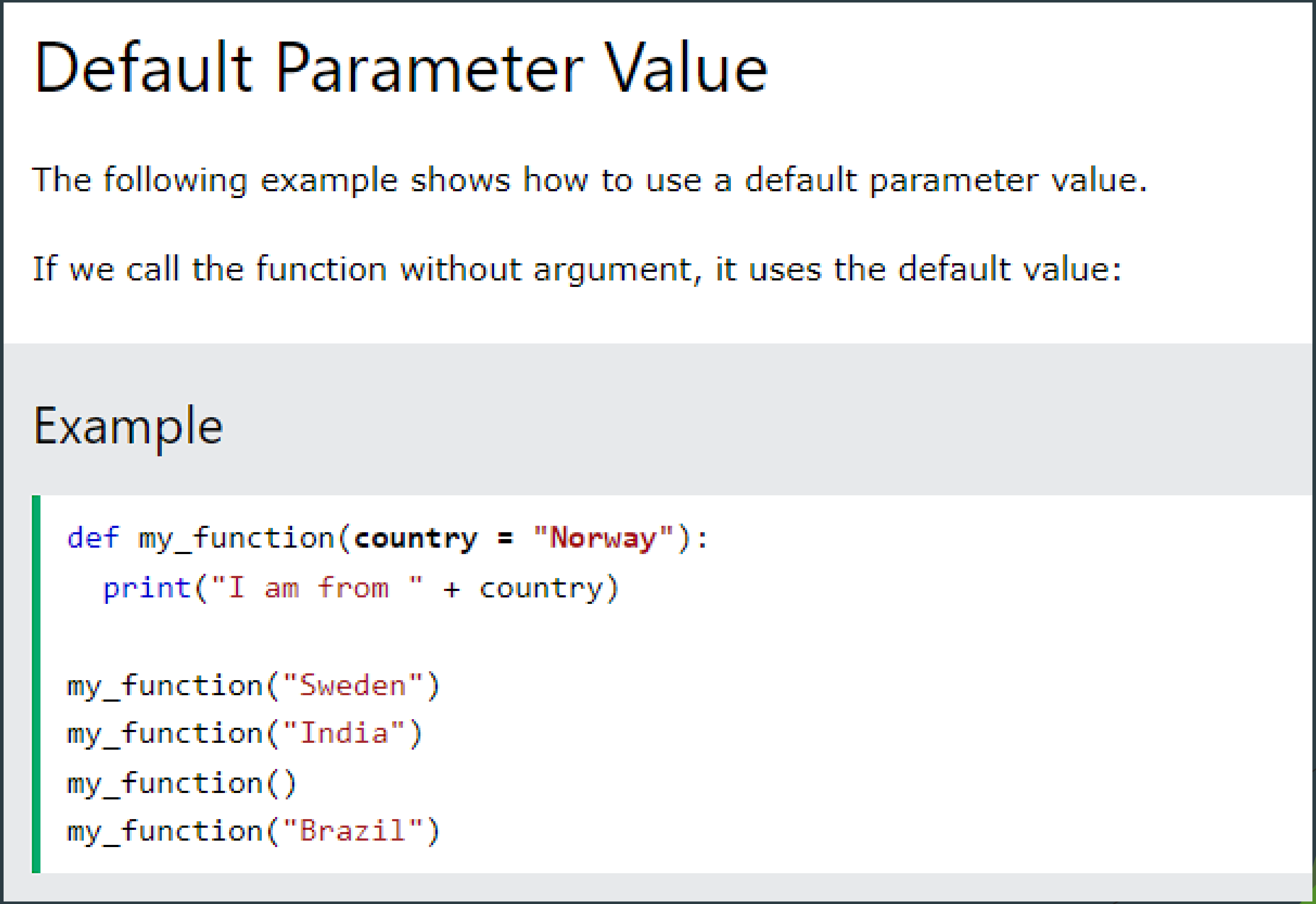
If you do not know how many keyword arguments that will be passed into your function, add two asterisk: `**` before the parameter name in the function definition.

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

Example

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")
```



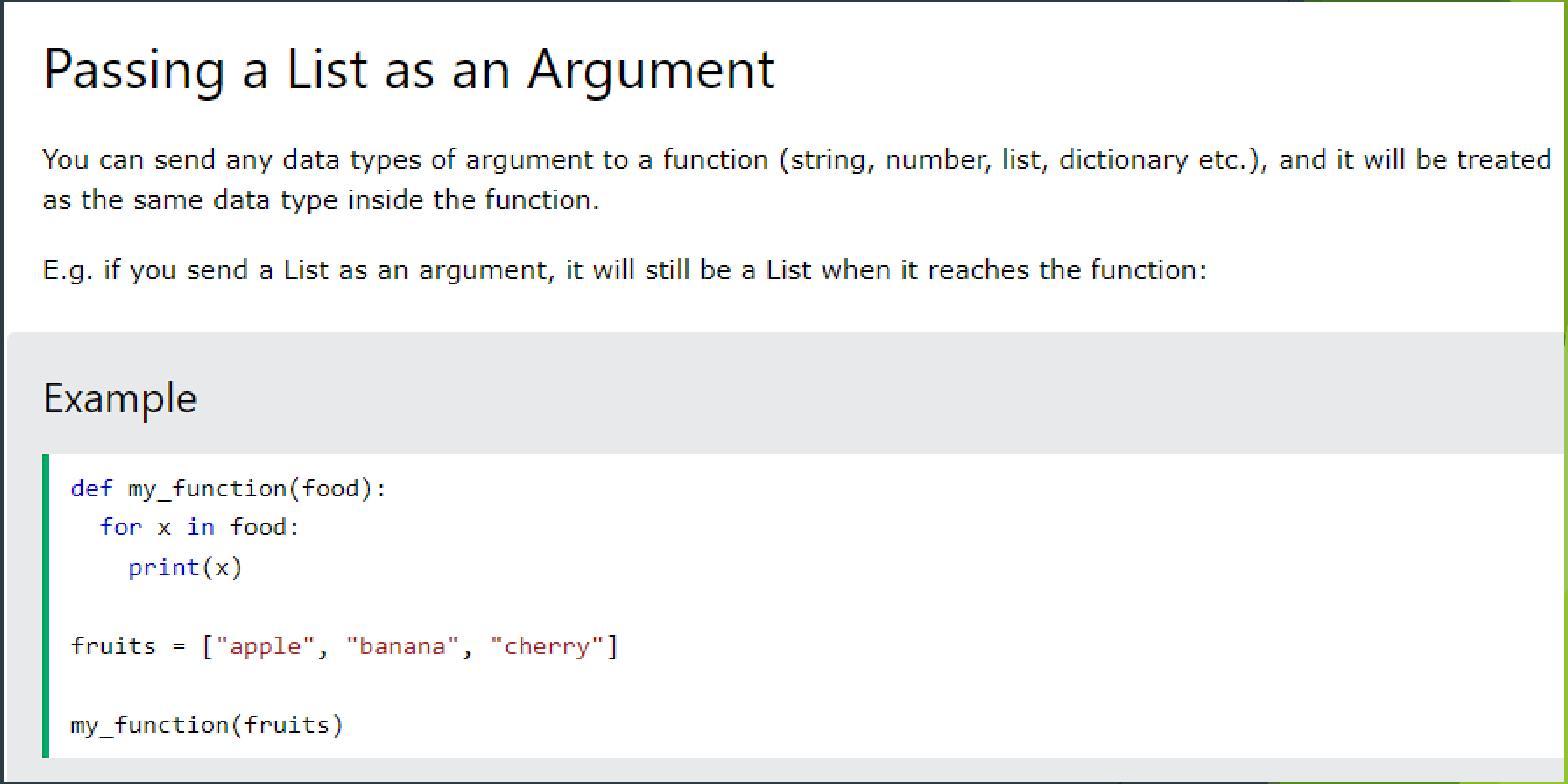
Default Parameter Value

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

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

Example

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

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



Passing a List as an Argument

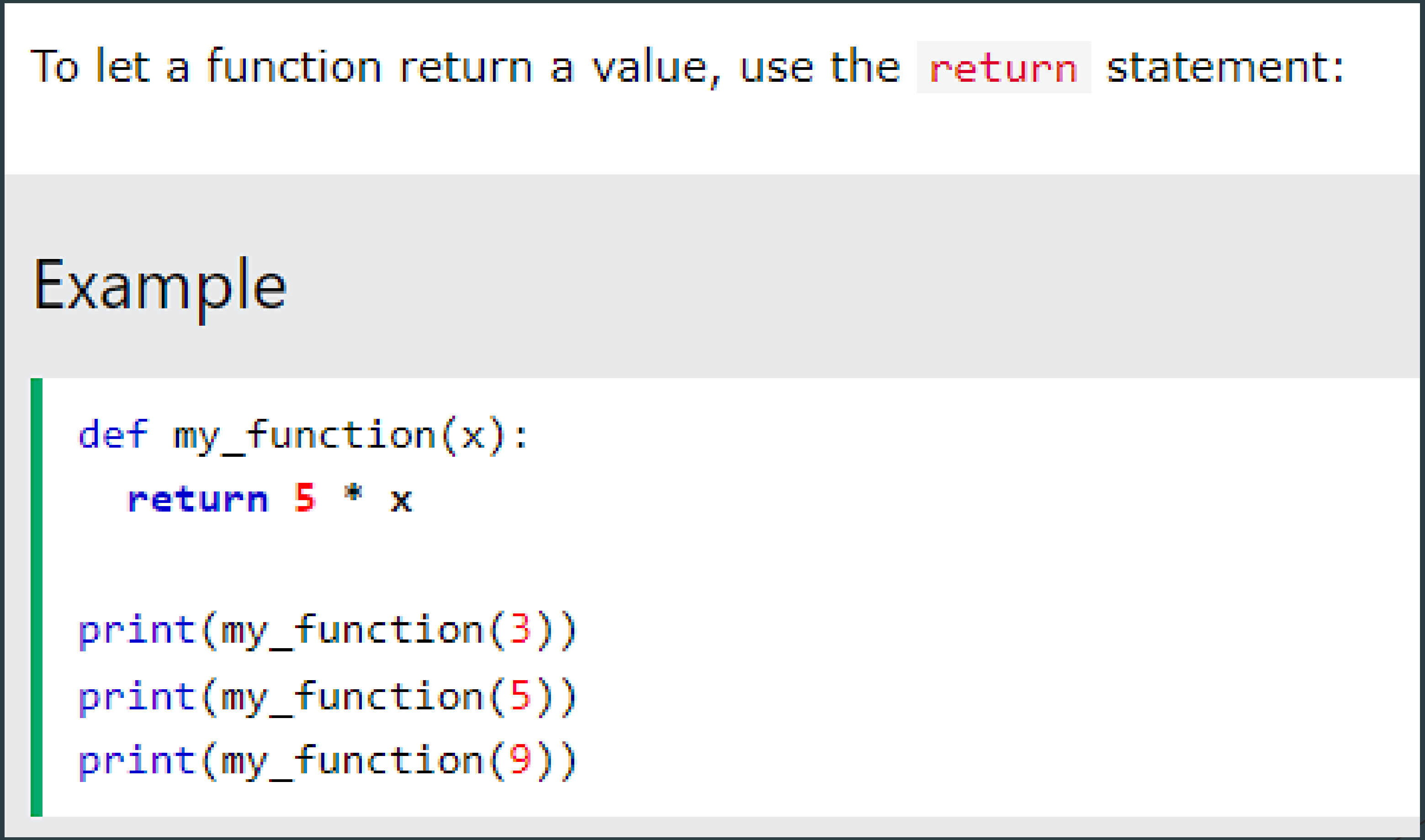
You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

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

Example

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

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

my_function(fruits)
```

Return Values



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

Example

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

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

Python - writing code

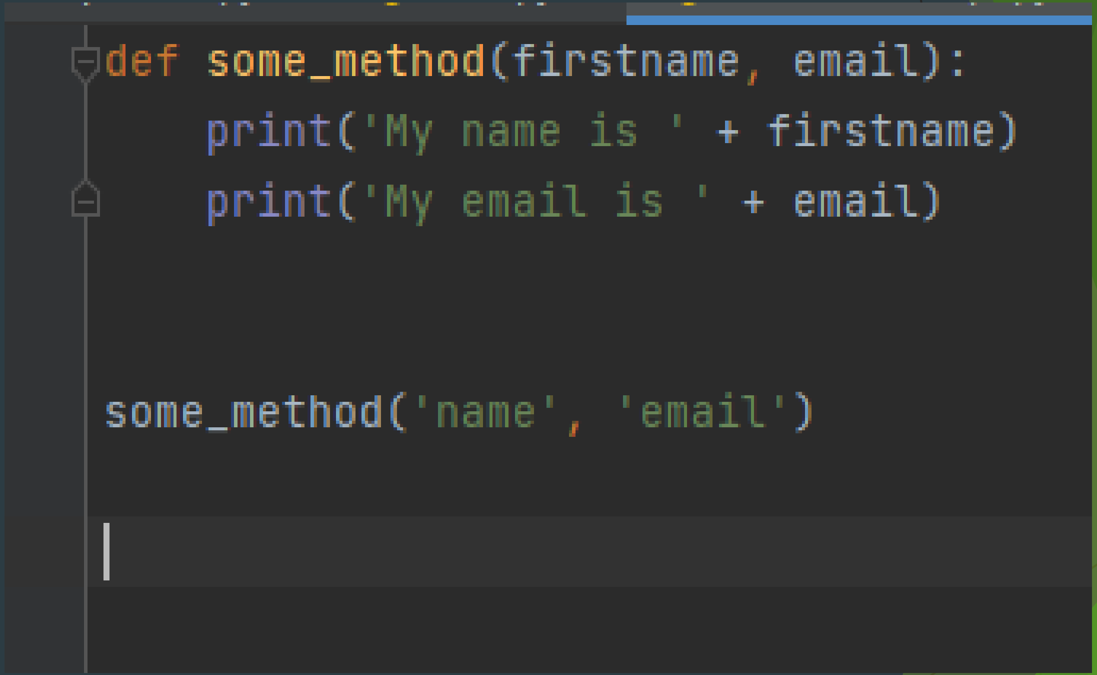
function:

```
def some_method(first_name, email):
    print('My name is ' + first_name)
    print('My email is ' + email)
```

some_method('name', 'email') - launching function

note!: spacing

note!: launch script through command line - e.g. *python script.py*



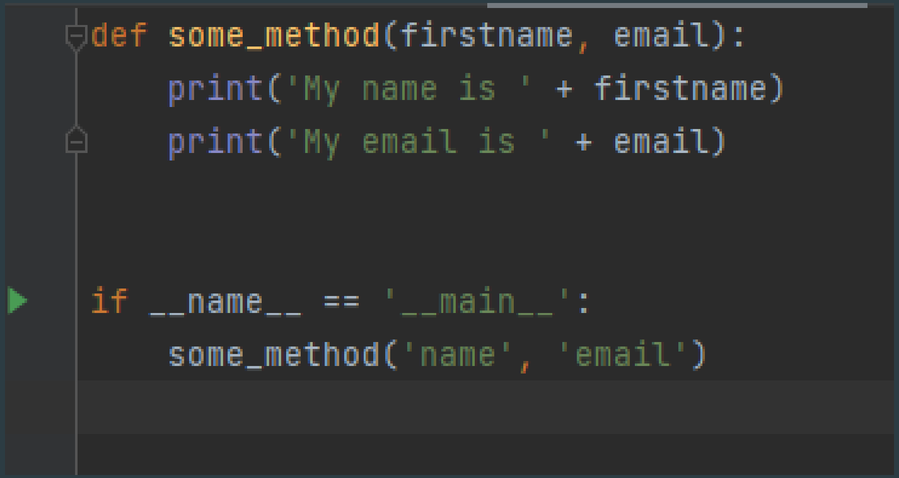
```
def some_method(firstname, email):
    print('My name is ' + firstname)
    print('My email is ' + email)


some_method('name', 'email')
```

Python - writing code

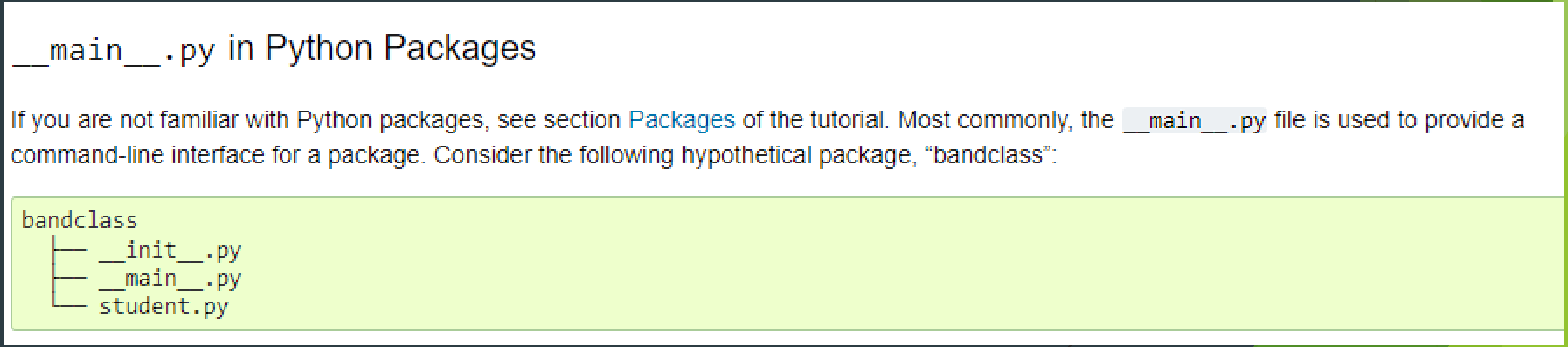
To Launch code from PyCharm write:

if __name__ == '__main__':

        some_method('name', 'email')



```python
def some_method(firstname, email):
    print('My name is ' + firstname)
    print('My email is ' + email)


if __name__ == '__main__':
    some_method('name', 'email')
```



`__main__.py` in Python Packages

If you are not familiar with Python packages, see section Packages of the tutorial. Most commonly, the `__main__.py` file is used to provide a command-line interface for a package. Consider the following hypothetical package, “bandclass”:

```
bandclass
  ├── __init__.py
  ├── __main__.py
  └── student.py
```

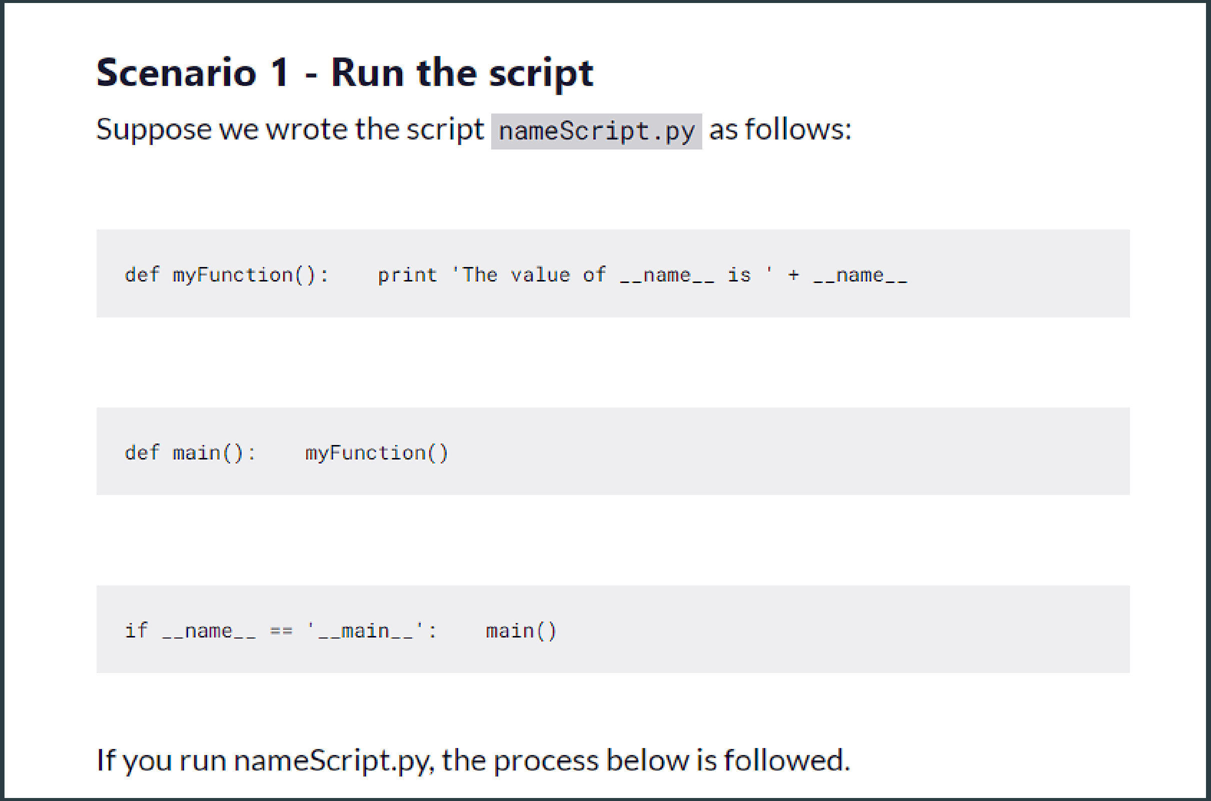
Python - writing code

Why is the __name__ variable used?

The __name__ variable (two underscores before and after) is a special Python variable. It gets its value depending on how we execute the containing script. Sometimes you write a script with functions that might be useful in other scripts as well. In Python, you can import that script as a module in another script. Thanks to this special variable, you can decide whether you want to run the script. Or that you want to import the functions defined in the script.

What values can the __name__ variable contain? When you run your script, the __name__ variable equals __main__. When you import the containing script, it will contain the name of the script.

Python - writing code



Scenario 1 - Run the script

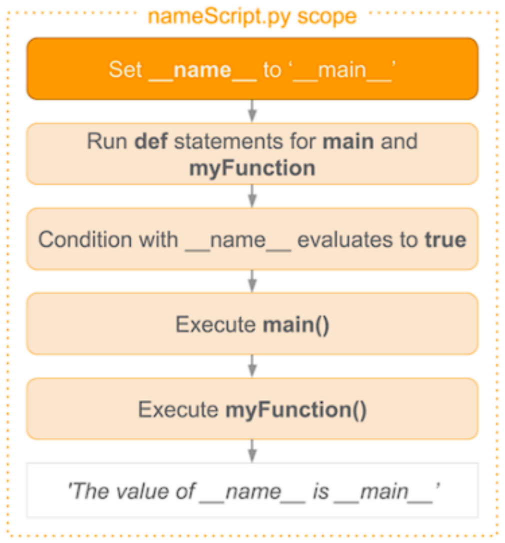
Suppose we wrote the script `nameScript.py` as follows:

```
def myFunction():    print 'The value of __name__ is ' + __name__
```

```
def main():    myFunction()
```

```
if __name__ == '__main__':    main()
```

If you run nameScript.py, the process below is followed.


nameScript.py scope
Set __name__ to '__main__'
Run def statements for main and myFunction
Condition with __name__ evaluates to true
Execute main()
Execute myFunction()
'The value of __name__ is __main__'

Before all other code is run, the `__name__` variable is set to __main__. After that, the `main` and `myFunction` def statements are run. Because the condition evaluates to true, the main function is called. This, in turn, calls myFunction. This prints out the value of `__main__`.

Python - writing code

Scenario 2 - Import the script in another script

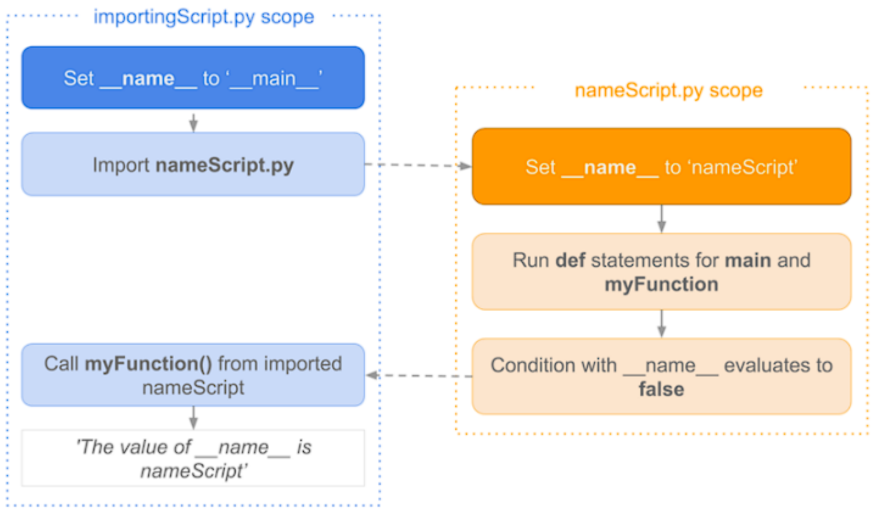
If we want to re-use myFunction in another script, for example `importingScript.py`, we can import `nameScript.py` as a module.

The code in `importingScript.py` could be as follows:

```
import nameScript as ns
```

```
ns.myFunction()
```

We then have two scopes: one of `importingScript` and the second scope of `nameScript`. In the illustration, you'll see how it differs from the first use case.



importingScript.py scope
Set __name__ to '__main__'
Import nameScript.py
Call myFunction() from imported nameScript
'The value of __name__ is nameScript'
nameScript.py scope
Set __name__ to 'nameScript'
Run def statements for main and myFunction
Condition with __name__ evaluates to false

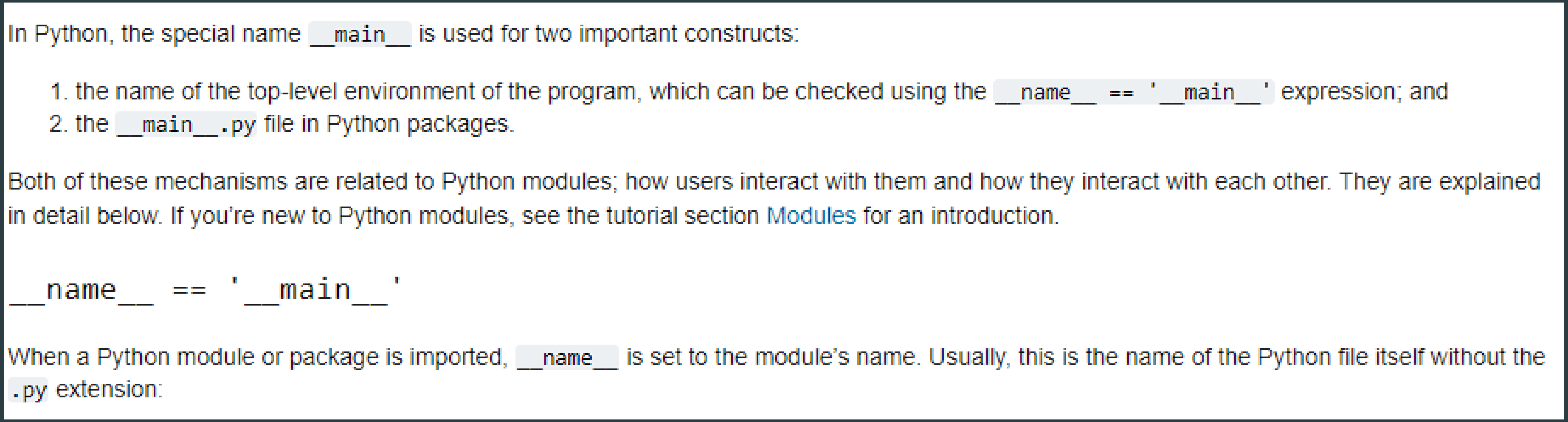
In importingScript.py the `__name__` variable is set to _main_. By importing nameScript, Python starts looking for a file by adding `.py` to the module name. It then runs the code contained in the imported file.

But this time it is set to nameScript. Again the def statements for main and myFunction are run. But, now the condition evaluates to false and main is not called.

In importingScript.py we call myFunction which outputs nameScript. NameScript is known to myFunction when that function was defined.

If you would print `__name__` in the importingScript, this would output `__main__`. The reason for this is that Python uses the value known in the scope of importingScript.

Python - writing code



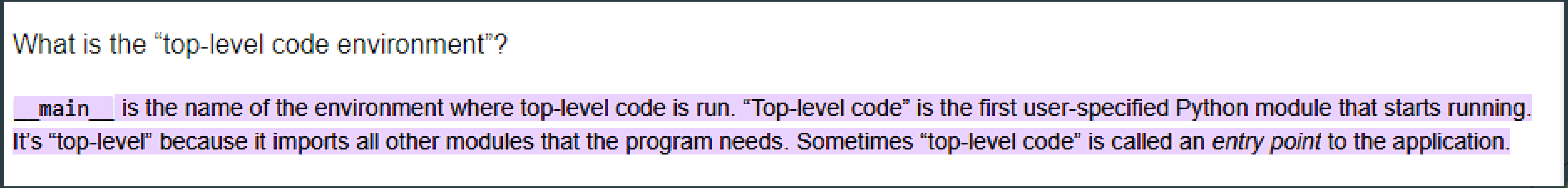
In Python, the special name `__main__` is used for two important constructs:

1. the name of the top-level environment of the program, which can be checked using the `__name__ == '__main__'` expression; and
2. the `__main__.py` file in Python packages.

Both of these mechanisms are related to Python modules; how users interact with them and how they interact with each other. They are explained in detail below. If you're new to Python modules, see the tutorial section Modules for an introduction.

`__name__ == '__main__'`

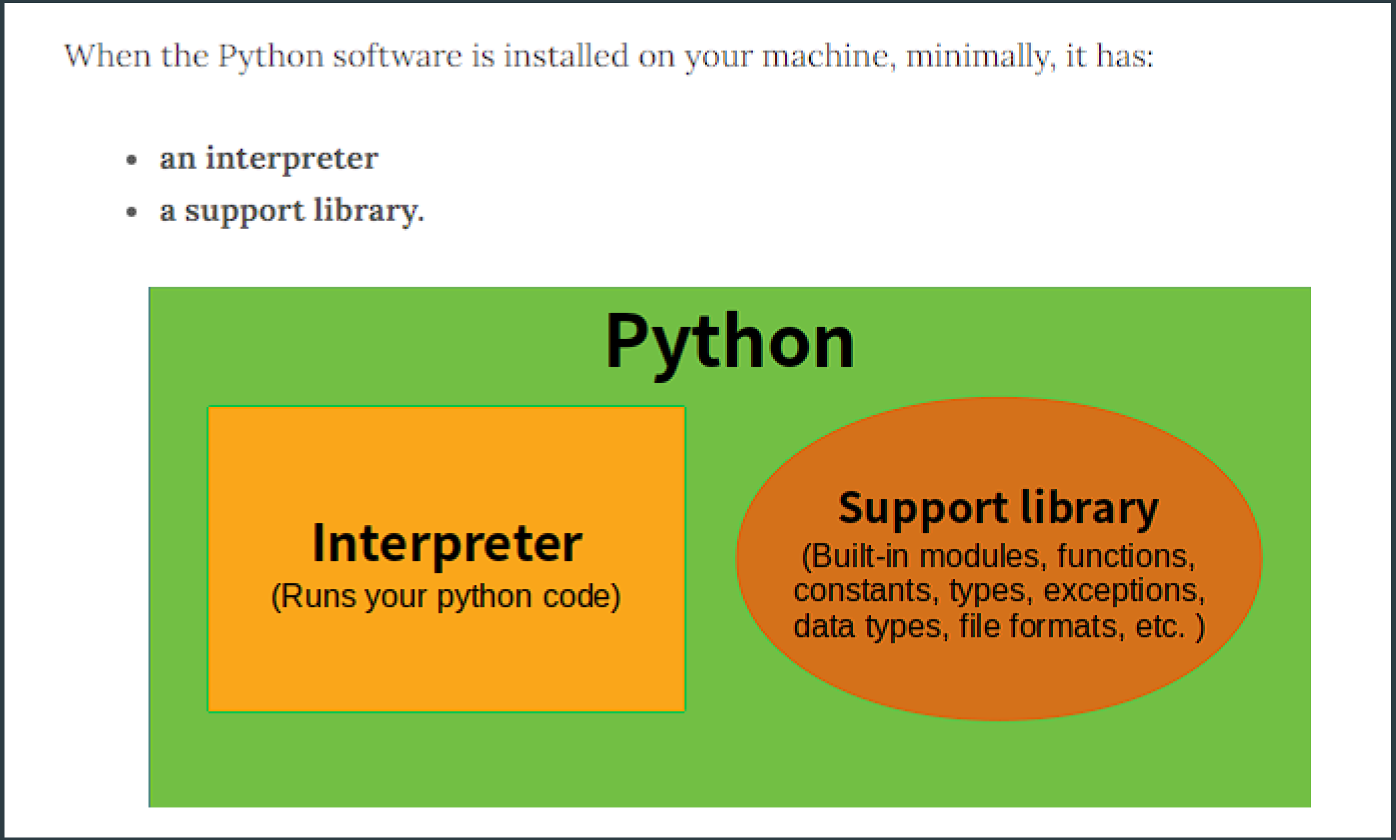
When a Python module or package is imported, `__name__` is set to the module's name. Usually, this is the name of the Python file itself without the `.py` extension:



What is the "top-level code environment"?

`__main__` is the name of the environment where top-level code is run. "Top-level code" is the first user-specified Python module that starts running. It's "top-level" because it imports all other modules that the program needs. Sometimes "top-level code" is called an *entry point* to the application.

How code is launched in Python

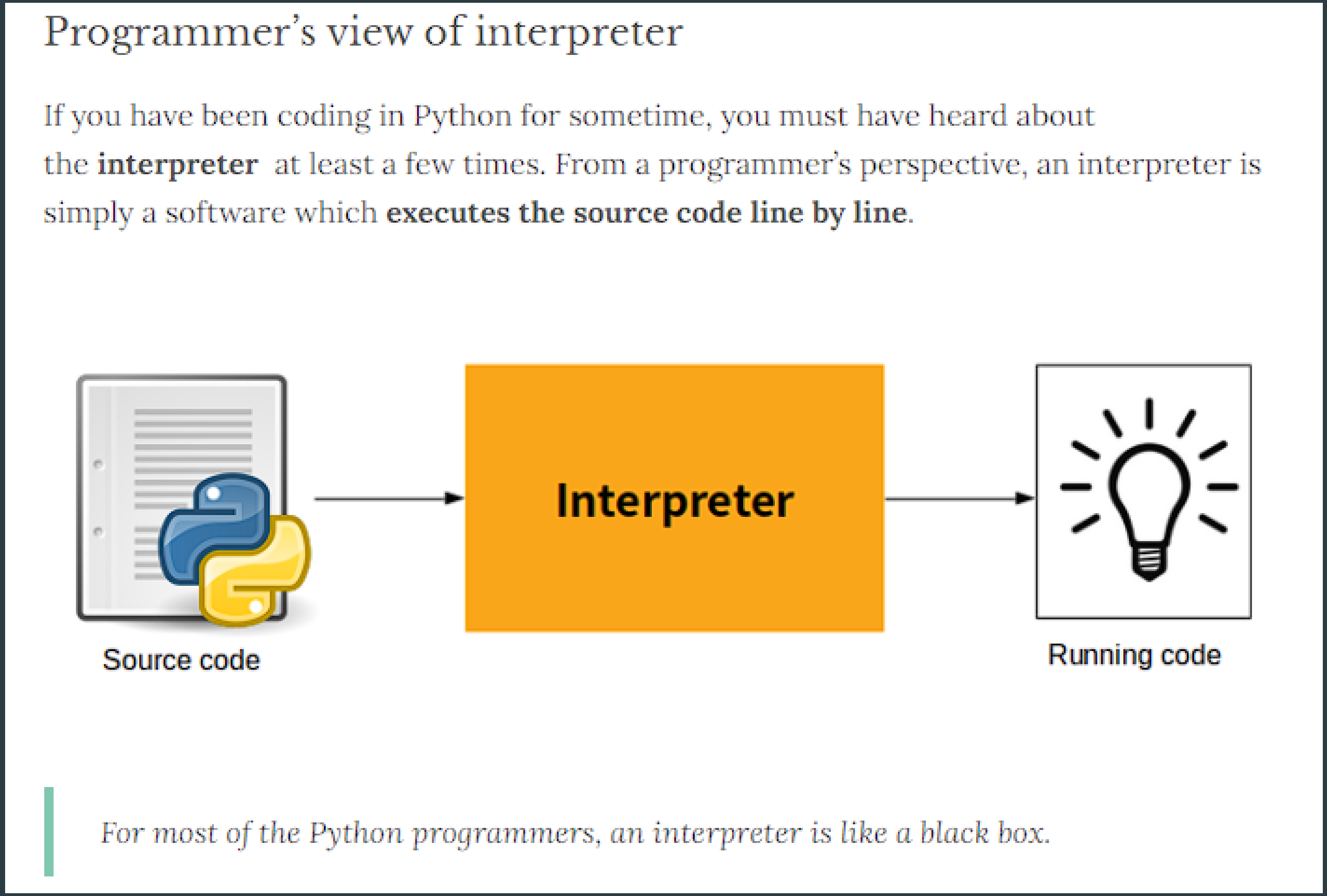


When the Python software is installed on your machine, minimally, it has:

- **an interpreter**
- **a support library.**

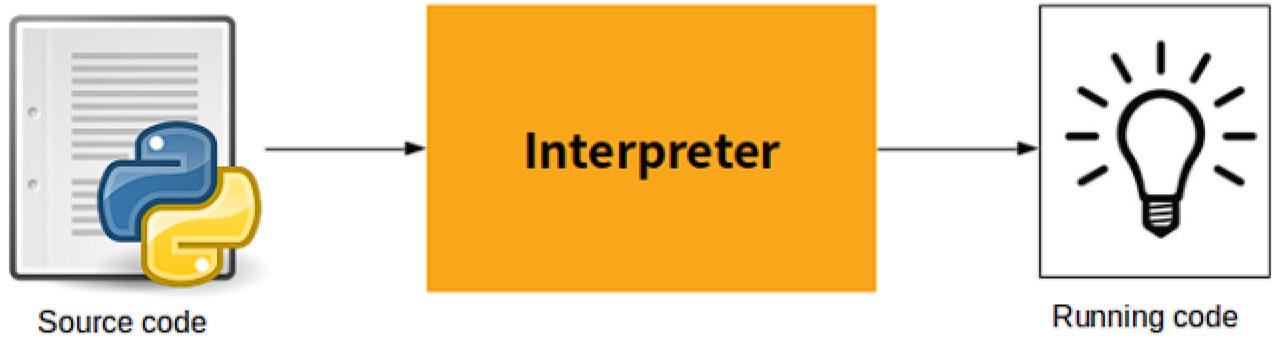
Python
Interpreter
(Runs your python code)
Support library
(Built-in modules, functions, constants, types, exceptions, data types, file formats, etc.)

How code is launched in Python



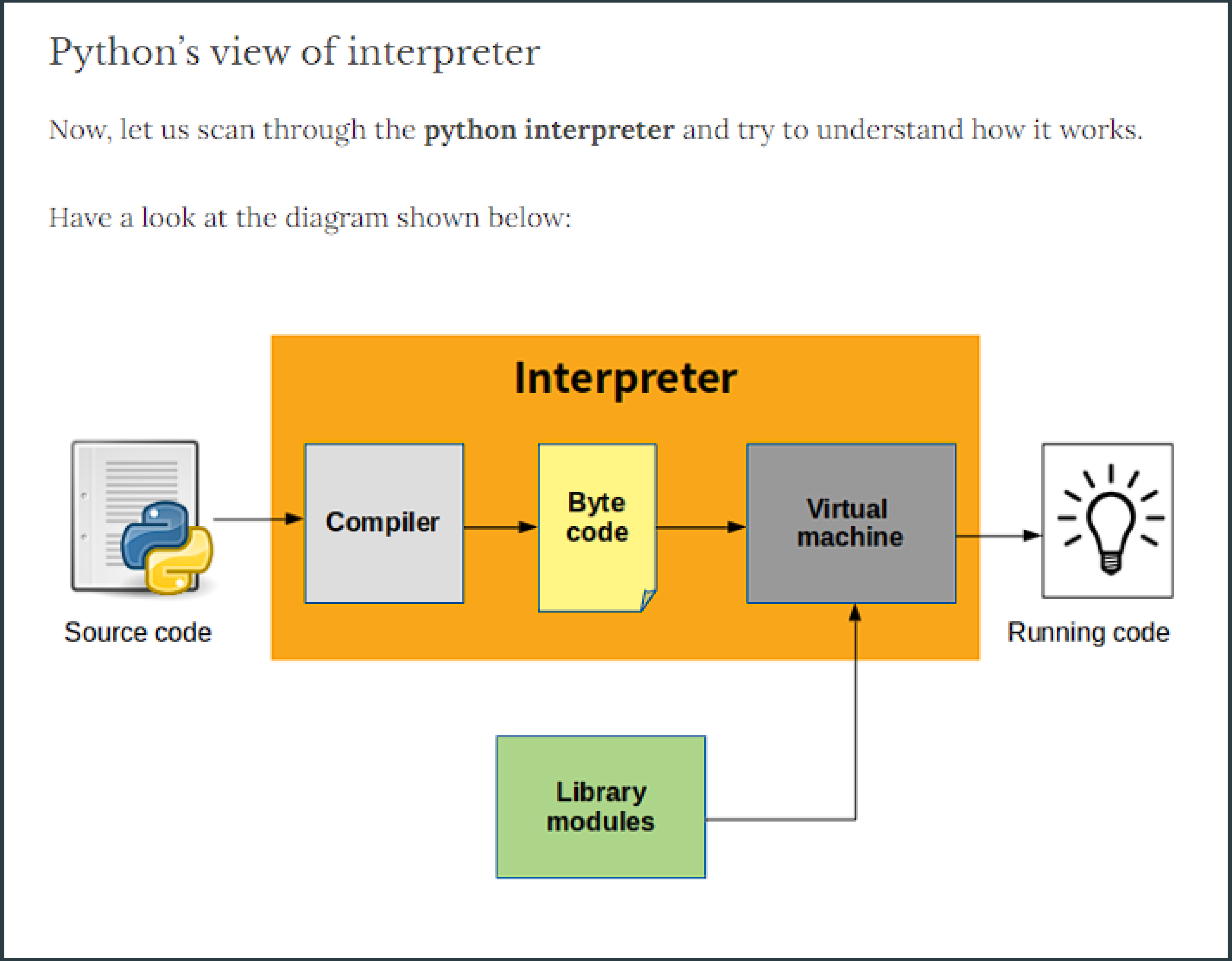
Programmer's view of interpreter

If you have been coding in Python for sometime, you must have heard about the **interpreter** at least a few times. From a programmer's perspective, an interpreter is simply a software which **executes the source code line by line**.



> *For most of the Python programmers, an interpreter is like a black box.*

How code is launched in Python

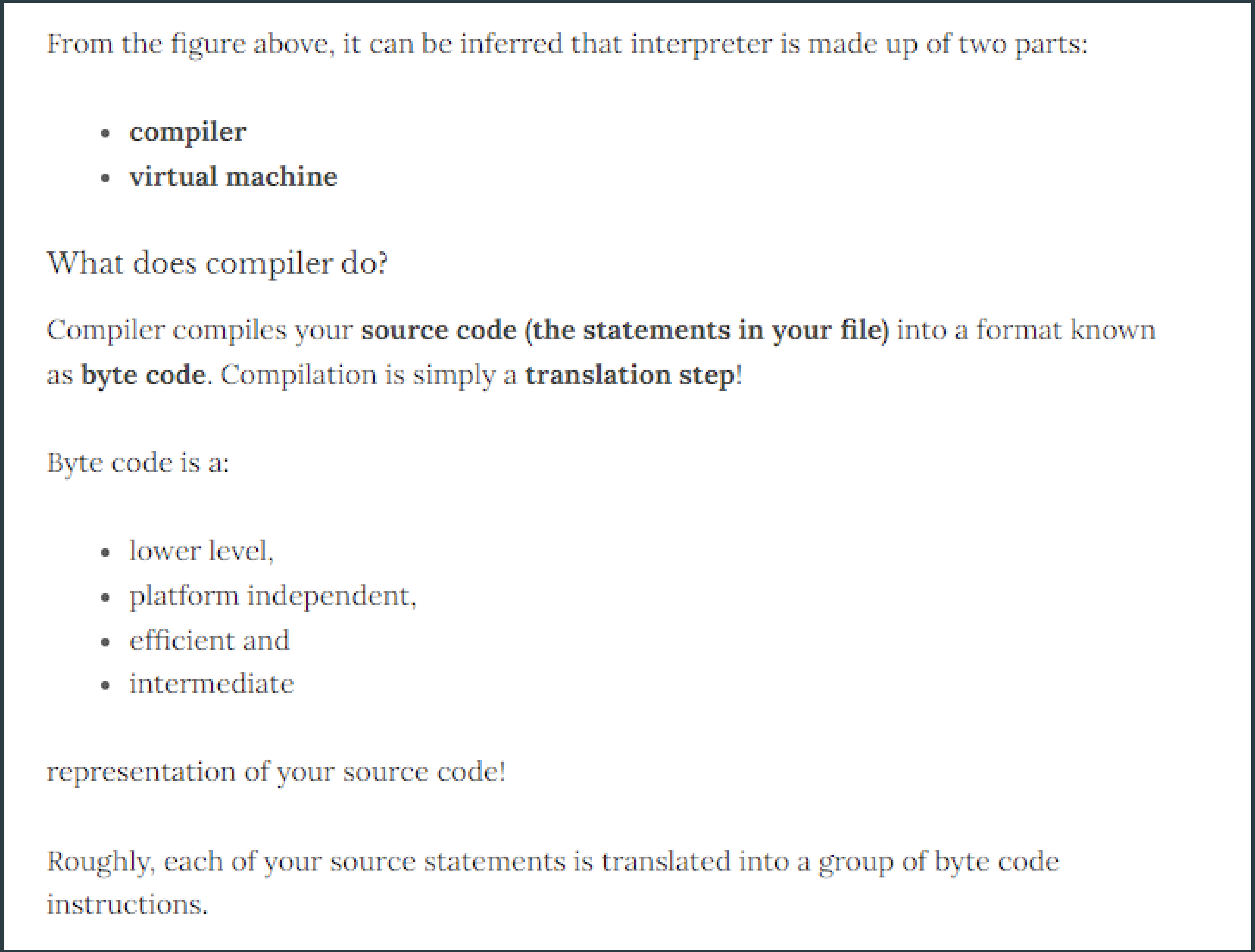


Python's view of interpreter

Now, let us scan through the **python interpreter** and try to understand how it works.

Have a look at the diagram shown below:

How code is launched in Python



From the figure above, it can be inferred that interpreter is made up of two parts:

- **compiler**
- **virtual machine**

What does compiler do?

Compiler compiles your **source code (the statements in your file)** into a format known as **byte code**. Compilation is simply a **translation step**!

Byte code is a:

- lower level,
- platform independent,
- efficient and
- intermediate

representation of your source code!

Roughly, each of your source statements is translated into a group of byte code instructions.

Python naming

1.General

Avoid using names that are too general or too wordy. Strike a good balance between the two.

Bad: data_structure, my_list, info_map, dictionary_for_the_purpose_of_storing_data_representing_word_definitions

Good: user_profile, menu_options, word_definitions

Don’t be a jackass and name things “O”, “l”, or “I”

When using CamelCase names, capitalize all letters of an abbreviation (e.g. HTTPServer)

2. Packages - folders, Modules - file name

Package names should be all lower case

When multiple words are needed, an underscore should separate them

It is usually preferable to stick to 1 word names

3. Classes

Class names should follow the UpperCaseCamelCase convention

Python’s built-in classes, however are typically lowercase words

Exception classes should end in “Error”

Python naming

4. Global (module-level) Variables

Global variables should be all lowercase

Words in a global variable name should be separated by an underscore

5. Instance Variables

Instance variable names should be all lower case

Words in an instance variable name should be separated by an underscore

Non-public instance variables should begin with a single underscore

If an instance name needs to be mangled, two underscores may begin its name

6. Methods

Method names should be all lower case

Words in an method name should be separated by an underscore

Non-public method should begin with a single underscore

If a method name needs to be mangled, two underscores may begin its name

Python naming

7. Method Arguments

Instance methods should have their first argument named ‘self’.

Class methods should have their first argument named ‘cls’

8. Functions

Function names should be all lower case

Words in a function name should be separated by an underscore

9. Constants

Constant names must be fully capitalized

Words in a constant name should be separated by an underscore

Tabs or Spaces? - PEP8

- **Spaces are the preferred indentation method.**
- *Tabs should be used solely to remain consistent with code that is already indented with tabs.*
- *Python disallows mixing tabs and spaces for indentation.*

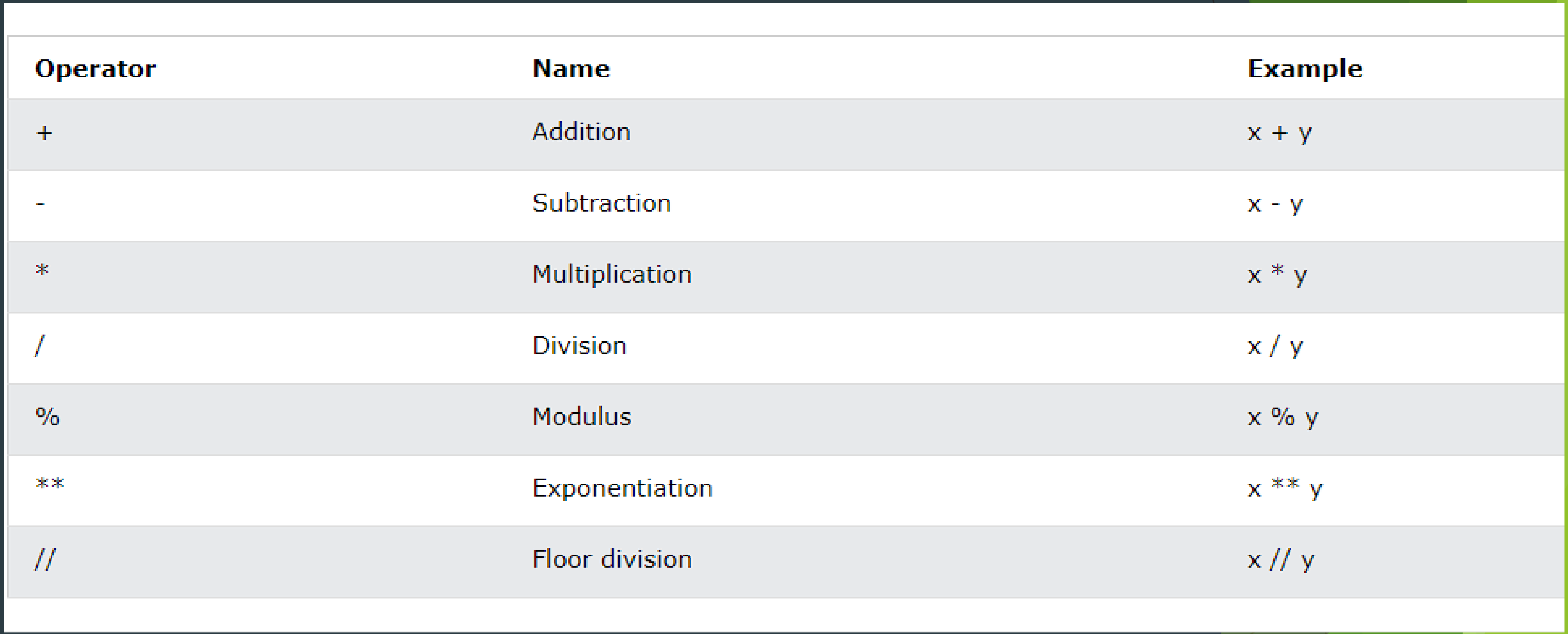
KISS - Keep it simple stupid

K KEEP
i iT
S SUPER
S SiMPLE

Practical part

```
x = 2
y = 5

print(x ** y)
#same as
2*2*2*2*2
```

```
x = 15
y = 2
print(x // y)
#the floor
division // rounds
the result down
to the nearest
whole number
```



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

Task 1 - Hello world

Lets create our first HELLO world app

1. Hello World - in one line
2. Hello world + name - setup variable

We will launch from the script and from Pycharm - if __name='__main__'

Task 2 - Calculator

Create calculator app

1. function for +
2. function for -
3. function for *
4. function for /
5. main function where you call previous functions and provide a and b variables

Reference

Variables: https://www.tutorialspoint.com/python/python_variable_types.htm#

Lists: https://www.w3schools.com/python/python_arrays.asp

Pep8 - Style Guide: https://peps.python.org/pep-0008/#method-names-and-instance-variables

__main__ - https://docs.python.org/3/library/__main__.html#:~:text=__main__%20is%20the,entry%20point%20to%20the%20application.

__name__ - https://www.freecodecamp.org/news/whats-in-a-python-s-name-506262fe61e8/#:~:text=The%20__name__%20variable%20(two%20underscores%20before%20and%20after,a%20module%20in%20another%20script.

Compiler - https://indianpythonista.wordpress.com/2018/01/04/how-python-runs/

Methods types - https://www.analyticsvidhya.com/blog/2020/11/basic-concepts-object-oriented-programming-types-methods-python/#:~:text=There%20are%20basically%20three%20types,Static%20Method

THANK YOU FOR YOUR ATTENTION!
...YOU ALWAYS HAVE MY ATTENTION.
makeameme.org