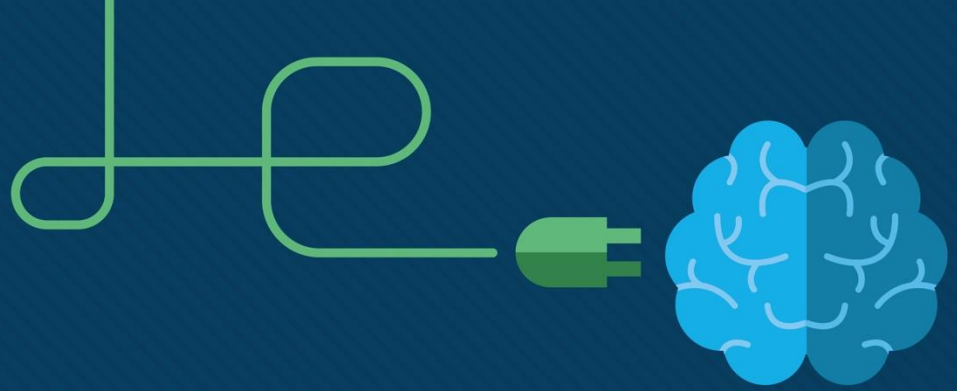


Programming Essentials in Python

Programming Essentials in Python

Module 4: Functions, tuples, dictionaries

Module Objectives

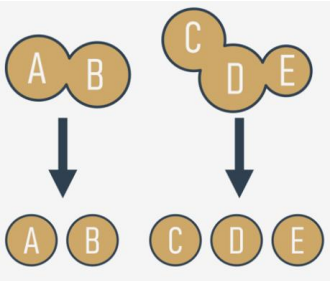
Module Title: Boolean values, conditional execution, loops, lists and list processing, logical and bitwise operations

Objectives
Defining and using functions;
Different ways of passing arguments;
Name scopes;
Tuples and dictionaries;
Data processing.

Why do we need functions?

Functions are tools to make life easier, and to simplify time-consuming and tedious tasks. A function often used when a particular piece of code is **repeated many times in your program.**

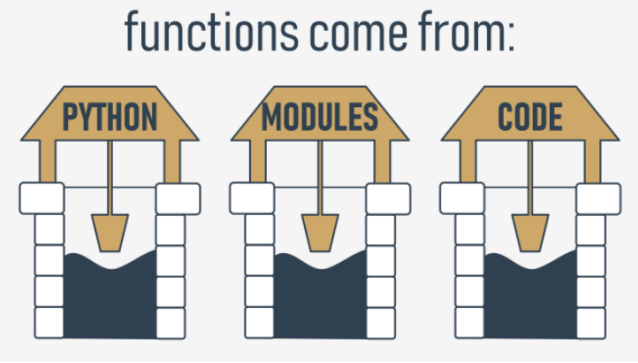
- **if a particular fragment of the code begins to appear in more than one place, consider the possibility of isolating it in the form of a function.**
- **if a piece of code becomes so large that reading and understating it may cause a problem, consider dividing it into separate, smaller problems, and implement each of them in the form of a separate function – decomposition.**



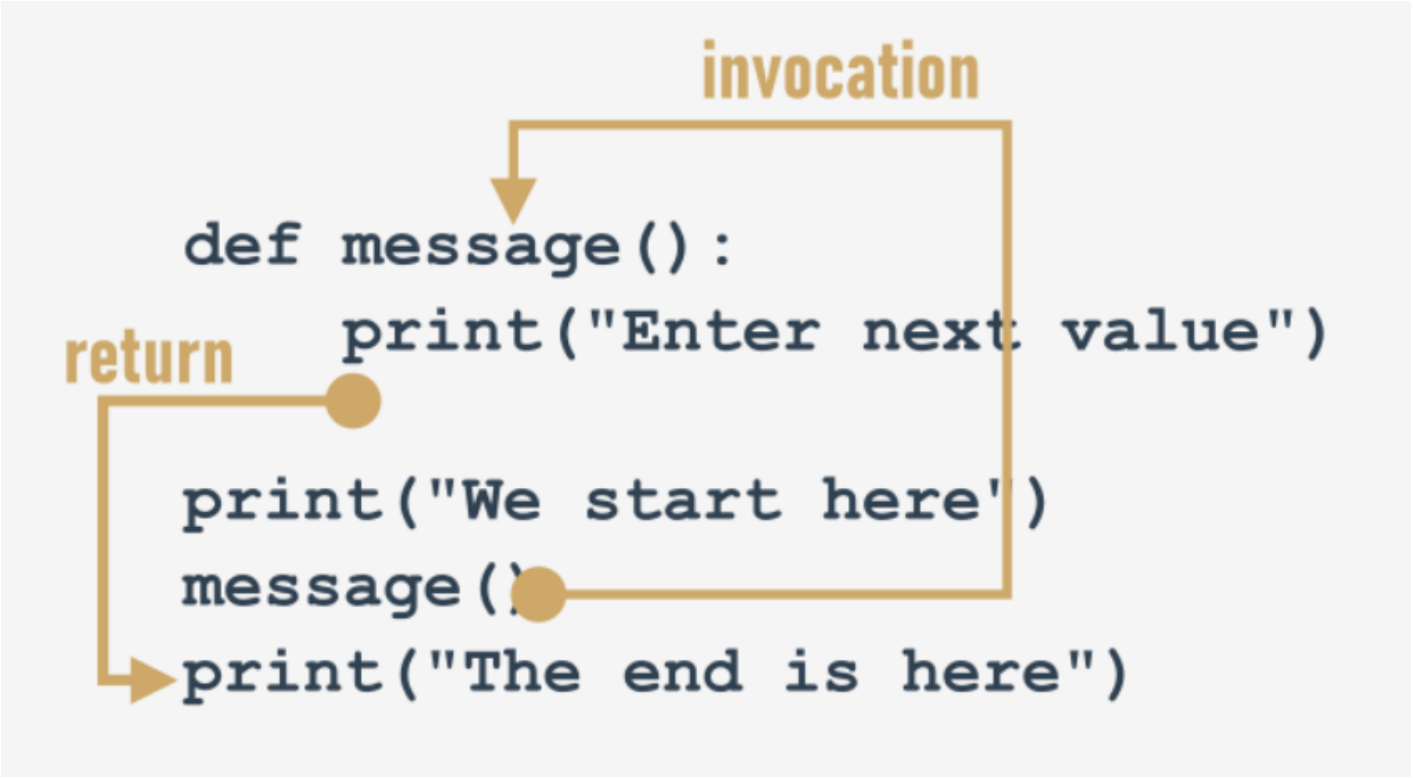
Where do the functions come from?

In general, functions come from at least three places:

- from Python itself - numerous functions (like `print()`) are an **integral part of Python**, and are always available without any additional effort on behalf of the programmer; we call these functions **built-in functions**;
- from Python's **preinstalled modules** - a lot of functions, very useful ones, but used significantly less often than built-in ones, are available in a number of modules installed together with Python; the use of these functions requires some additional steps from the programmer in order to make them fully accessible (we'll tell you about this in a while);
- **directly from your code** - you can write your own functions, place them inside your code, and use them freely;
- there is one other possibility, but it's connected with classes, so we'll omit it for now.



How functions work



Writing functions

How do you make such a function?

You need to **define** it. The word *define* is significant here.

This is what the simplest function definition looks like:

```
def functionName():
    functionBody
```

- It always starts with the **keyword** `def` (for *define*)
- next after `def` goes the **name of the function** (the rules for naming functions are exactly the same as for naming variables)
- after the function name, there's a place for a pair of **parentheses** (they contain nothing here, but that will change soon)
- the line has to be ended with a **colon**;
- the line directly after `def` begins the **function body** - a couple (at least one) of necessarily **nested instructions**, which will be executed every time the function is invoked; note: the **function ends where the nesting ends**, so you have to be careful.

Writing functions

Original code

```
print("Enter a value: ")
a = int(input())

print("Enter a value: ")
b = int(input())

print("Enter a value: ")
c = int(input())
```

Code using function

```
def message():
    print("Enter a value: ")

message()
a = int(input())
message()
b = int(input())
message()
c = int(input())
```

There are two, very important, catches. Here's the first of them:

You mustn't invoke a function which is not known at the moment of invocation.

The second catch sounds a little simpler:

You mustn't have a function and a variable of the same name.

Writing functions

Parametrized functions

A parameter is actually a variable, but there are two important factors that make parameters different and special:

- **parameters exist only inside functions in which they have been defined**, and the only place where the parameter can be defined is a space between a pair of parentheses in the `def` statement;
- **assigning a value to the parameter is done at the time of the function's invocation**, by specifying the corresponding argument.

```
def function(parameter):
    ###
```

Don't forget:

- **parameters live inside functions** (this is their natural environment)
- **arguments exist outside functions**, and are carriers of values passed to corresponding parameters.

Writing functions

Parametrized functions

```
def message(number):
    ###
```

The definition specifies that our function operates on just one parameter named `number`. You can use it as an ordinary variable, but **only inside the function** - it isn't visible anywhere else.

```
def message(number):
    print("Enter a number:", number)
```

We've made use of the parameter. Note: we haven't assigned the parameter with any value. Is it correct?

Writing functions

Parametrized functions

```
def message(number):
    print("Enter a number:", number)

message()
```

A value for the parameter will arrive from the function's environment.

Remember: **specifying one or more parameters in a function's definition** is also a requirement, and you have to fulfil it during invocation. You must **provide as many arguments as there are defined parameters**.

```
def message(number):
    print("Enter a number:", number)

message(1)
```

```
Enter a number: 1
```

Writing functions

Parametrized functions

We have to make you sensitive to one important circumstance.

It's legal, and possible, to have a **variable named the same as a function's parameter**.

```
def message(number):
    print("Enter a number:", number)

number = 1234
message(1)
print(number)
```

```
Enter a number: 1
1234
```

A situation like this activates a mechanism called **shadowing**:

- parameter `x` shadows any variable of the same name, but...
- ... only inside the function defining the parameter.

The parameter named `number` is a completely different entity from the variable named `number`.

Writing functions

Positional parameter passing

A technique which assigns the i^{th} (first, second, and so on) argument to the i^{th} (first, second, and so on) function parameter is called **positional parameter passing**, while arguments passed in this way are named **positional arguments**.

```
def myFunction(a, b, c):
    print(a, b, c)

myFunction(1, 2, 3)
```

```
1 2 3
```

```
def introduction(firstName, lastName):
    print('Hello, my name is {} {}.'.format(firstName, lastName))

introduction("Luke", "Skywalker")
introduction("Jesse", "Quick")
introduction("Clark", "Kent")
```

```
Hello, my name is Luke Skywalker.
Hello, my name is Jesse Quick.
Hello, my name is Clark Kent.
```

Writing functions

Keyword argument passing

Python offers another convention for passing arguments, where **the meaning of the argument is dictated by its name**, not by its position - it's called **keyword argument passing**.

```
def introduction(firstName, lastName):
    print("Hello, my name is", firstName, lastName)

introduction(firstName = "James", lastName = "Bond")
introduction(lastName = "Skywalker", firstName = "Luke")
```

```
Hello, my name is James Bond
Hello, my name is Luke Skywalker
```

It happens at times that a particular parameter's values are in use more often than others. Such arguments may have their **default (predefined) values** taken into consideration when their corresponding arguments have been omitted.

Effects and results: the return instruction

Effects and results: the `return` instruction

All the previously presented functions have some kind of effect - they produce some text and send it to the console.

To get **functions to return a value** (but not only for this purpose) you use the `return` instruction.

The `return` instruction has **two different variants** - let's consider them separately.

`return` without an expression

The first consists of the keyword itself, without anything following it.

When used inside a function, it causes the **immediate termination of the function's execution, and an instant return (hence the name) to the point of invocation**.

Effects and results: the return instruction

return without an expression

Examples:

```
def happyNewYear(wishes = True):
    print("Three...")
    print("Two...")
    print("One...")
    if not wishes:
        return

    print("Happy New Year!")

happyNewYear(True)
```

```
Three...
Two...
One...
Happy New Year!
```

```
def happyNewYear(wishes = True):
    print("Three...")
    print("Two...")
    print("One...")
    if not wishes:
        return

    print("Happy New Year!")

happyNewYear(False)
```

```
Three...
Two...
One...
```

Effects and results: the return instruction

`return` with an expression

The second `return` variant is **extended with an expression**:

```
function():
    return expression
```

There are two consequences of using it:

- it causes the **immediate termination of the function's execution** (nothing new compared to the first variant)
- moreover, the function will **evaluate the expression's value and will return (hence the name once again) it as the function's result**.

Effects and results: the return instruction

return with an expression

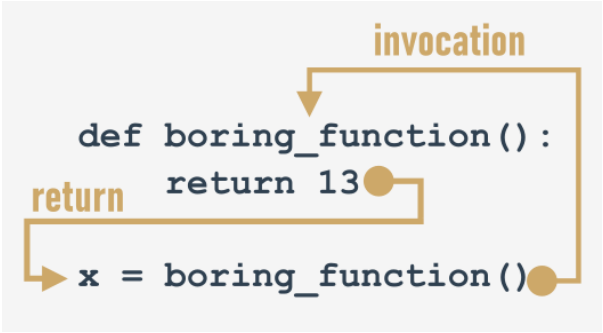
Examples:

```
def boringFunction():
    return 123

x = boringFunction()

print("The boringFunction has returned its result. It's:", x)
```

```
The boringFunction has returned its result. It's: 123
```



Effects and results: lists and functions

There are two additional questions that should be answered here.

The first is: **may a list be sent to a function as an argument?**

Examples:

```
def sumOfList(lst):
    sum = 0

    for elem in lst:
        sum += elem

    return sum

print(sumOfList([5, 4, 3]))
```

```
12
```

Effects and results: lists and functions

There are two additional questions that should be answered here.

The first is: **may a list be sent to a function as an argument?**

The second question is: **may a list be a function result?**

Examples:

```
def sumOfList(lst):
    sum = 0

    for elem in lst:
        sum += elem

    return sum

print(sumOfList([5, 4, 3]))
```

```
12
```

```
def strangeListFunction(n):
    strangeList = []

    for i in range(0, n):
        strangeList.insert(0, i)

    return strangeList

print(strangeListFunction(5))
```

```
[4, 3, 2, 1, 0]
```

Function and scopes

Functions and scopes

The **scope of a name** (e.g., a variable name) is the part of a code where the name is properly recognizable.

For example, the scope of a function's parameter is the function itself. The parameter is inaccessible outside the function.

Examples:

```
def scopeTest():
    a = 123

scopeTest()
print(a)
```

```
def myFunction():
    print("Do I know that variable?", var)

var = 1
myFunction()
print(var)
```

```
Do I know that variable? 1
1
```

```
# A variable existing outside a function has a scope inside the functions' bodies.
```

Function and scopes

Functions and scopes

More examples:

```
def myFunction():
    var = 2
    print("Do I know that variable?", var)

var = 1
myFunction()
print(var)
```

```
Do I know that variable? 2
1
```

```
def myFunction():
    global var
    var = 2
    print("Do I know that variable?", var)

var = 1
myFunction()
print(var)
```

```
Do I know that variable? 2
2
```

Function and scopes

How the function interacts with its arguments

The conclusion is obvious - **changing the parameter's value doesn't propagate outside the function** (in any case, not when the variable is a scalar, like in the example).

This also means that a function receives the **argument's value**, not the argument itself. This is true for scalars.

```
def myFunction(myList1):
    print(myList1)
    del myList1[0:2]

myList2 = [2, 3, 4]
myFunction(myList2)
print(myList2)
```

```
[2, 3, 4]
[4]
```

- if the argument is a list, then changing the value of the corresponding parameter doesn't affect the list (remember: variables containing lists are stored in a different way than scalars)
- but if you change a list identified by the parameter (note: the list, not the parameter!), the list will reflect the change.

Two-parameters functions

Examples:

```
def bmi(weight, height):
    return weight / height ** 2

print(bmi(80, 1.65))
```

```
29.384756657483933
```

```
def bmi(weight, height):
    if height < 1.0 or height > 2.5 or \
    weight < 20 or weight > 200:
        return None

    return weight / height ** 2

print(bmi(195.5, 1.65))
```

```
71.80899908172636
```

Sequence types and mutability

A **sequence type is a type of data in Python which is able to store more than one value (or less than one, as a sequence may be empty), and these values can be sequentially (hence the name) browsed**, element by element.

As the `for` loop is a tool especially designed to iterate through sequences, we can express the definition as: **a sequence is data which can be scanned by the `for` loop**.

The second notion - **mutability** - is a property of any of Python's data that describes its readiness to be freely changed during program execution. There are two kinds of Python data: **mutable** and **immutable**.

Mutable data can be freely updated at any time - we call such an operation in situ.

Sequence types and mutability

In situ is a Latin phrase that translates as literally *in position*. For example, the following instruction modifies the data in situ:

```
list.append(1)
```

Immutable data cannot be modified in this way.

Imagine that a list can only be assigned and read over. You would be able neither to append an element to it, nor remove any element from it. This means that appending an element to the end of the list would require the recreation of the list from scratch.

You would have to build a completely new list, consisting of the all elements of the already existing list, plus the new element.

The data type we want to tell you about now is a **tuple**. **A tuple is an immutable sequence type**. It can behave like a list, but it mustn't be modified in situ.

What is tuple

What is a tuple?

The first and the clearest distinction between lists and tuples is the syntax used to create them - **tuples prefer to use parenthesis**, whereas lists like to see brackets, although it's also **possible to create a tuple just from a set of values separated by commas**.

Example:

```
tuple1 = (1, 2, 4, 8)
tuple2 = 1., .5, .25, .125, 8

print(tuple1)
print(tuple2)
```

```
(1, 2, 4, 8)
(1.0, 0.5, 0.25, 0.125, 8)
```

Note: **each tuple element may be of a different type** (floating-point, integer, or any other not-as-yet-introduced kind of data).

What is tuple

How to create a tuple?

It is possible to create an empty tuple - parentheses are required then:

```
emptyTuple = ()
```

If you want to create a **one-element tuple**, you have to take into consideration the fact that, due to syntax reasons (a tuple has to be distinguishable from an ordinary, single value), you must end the value with a comma:

```
oneElementTuple1 = (1, )
oneElementTuple2 = 1.,
```

Removing the commas won't spoil the program in any syntactical sense, but you will instead get two single variables, not tuples.

How to use a tuple

Examples:

```
myTuple = (1, 10, 100, 1000)

print(myTuple[0])
print(myTuple[-1])
print(myTuple[1:])
print(myTuple[:-2])

for elem in myTuple:
    print(elem)
```

```
1
1000
(10, 100, 1000)
(1, 10)
1
10
100
1000
```

How to use a tuple

```
myTuple = (1, 10, 100)

t1 = myTuple + (1000, 10000)
t2 = myTuple * 3

print(len(t2))
print(t1)
print(t2)
print(10 in myTuple)
print(-10 not in myTuple)
```

```
9
(1, 10, 100, 1000, 10000)
(1, 10, 100, 1, 10, 100, 1, 10, 100)
True
True
```

What else can tuples do for you?

- the `len()` function accepts tuples, and returns the number of elements contained inside;
- the `+` operator can join tuples together (we've shown you this already)
- the `*` operator can multiply tuples, just like lists;
- the `in` and `not in` operators work in the same way as in lists.

What is a dictionary

What is a dictionary?

The **dictionary** is another Python data structure. It's **not a sequence** type (but can be easily adapted to sequence processing) and it is **mutable**.

To explain what the Python dictionary actually is, it is important to understand that it is literally a dictionary.

The Python dictionary works in the same way as **a bilingual dictionary**. For example, you have an English word (e.g., cat) and need its French equivalent. You browse the dictionary in order to find the word (you may use different techniques to do that - it doesn't matter) and eventually you get it. Next, you check the French counterpart and it is (most probably) the word "chat".

What is a dictionary

In Python's world, the word you look for is named a `key`. The word you get from the dictionary is called a `value`.

This means that a dictionary is a set of **key-value** pairs. Note:

- each key must be **unique** - it's not possible to have more than one key of the same value;
- a key may be **data of any type**: it may be a number (integer or float), or even a string;
- a dictionary is not a list - a list contains a set of numbered values, while a **dictionary holds pairs of values**;
- the `len()` function works for dictionaries, too - it returns the numbers of key-value elements in the dictionary;
- a dictionary is a **one-way tool** - if you have an English-French dictionary, you can look for French equivalents of English terms, but not vice versa.

How to make a dictionary

Example:

```
dict = {"cat" : "chat", "dog" : "chien", "horse" : "cheval"}
phoneNumbers = {'boss' : 5551234567, 'Suzy' : 22657854310}
emptyDictionary = {}

print(dict)
print(phoneNumbers)
print(emptyDictionary)
```

```
{'cat': 'chat', 'dog': 'chien', 'horse': 'cheval'}
{'boss': 5551234567, 'Suzy': 22657854310}
{}
```

The list of pairs is **surrounded by curly braces**, while the pairs themselves are **separated by commas**, and the **keys and values by colons**.

How to make a dictionary

More examples:

```
dict = {"cat" : "chat", "dog" : "chien", "horse" : "cheval"}
phoneNumbers = {'boss' : 5551234567, 'Suzy' : 22657854310}
emptyDictionary = {}

print(dict["cat"])
print(phoneNumbers["Suzy"])
```

```
chat
22657854310
```

How to make a dictionary

More examples:

```
1  dict = {"cat" : "chat", "dog" : "chien", "horse" : "cheval"}
2
3  words = ['cat', 'lion', 'horse']
4
5  for key in words:
6      if key in dict:
7          print(key, "->", dict[key])
8      else:
9          print(key, "is not in dictionary")
```

```
cat -> chat
lion is not in dictionary
horse -> cheval
```

How to use a dictionary: the keys()

How to use a dictionary: the `keys()`

Can dictionaries be **browsed** using the `for` loop, like lists or tuples?

No, because a dictionary is **not a sequence type** - the `for` loop is useless with it.

Yes, because there are simple and very effective tools that can **adapt any dictionary to the** `for` **loop requirements** (in other words, building an intermediate link between the dictionary and a temporary sequence entity).

The first of them is a method named `keys()`, possessed by each dictionary. The method **returns a list built of all the keys gathered within the dictionary**. Having a list of keys enables you to access the whole dictionary in an easy and handy way.

How to use a dictionary: the keys()

Example:

```
dict = {"cat" : "chat", "dog" : "chien", "horse" : "cheval"}

for key in dict.keys():
    print(key, "->", dict[key])
```

```
cat -> chat
dog -> chien
horse -> cheval
```

The sorted() function

```
dict = {"cat" : "chat", "dog" : "chien", "horse" : "cheval"}

for key in sorted(dict.keys()):
    print(key, "->", dict[key])
```

The items() and values() methods

Another way is based on using a dictionary's method named `items()`. The method **returns a list of tuples** (this is the first example where tuples are something more than just an example of themselves) **where each tuple is a key-value pair**.

Example:

```
dict = {"cat" : "chat", "dog" : "chien", "horse" : "cheval"}

for english, french in dict.items():
    print(english, "->", french)
```

```
cat -> chat
dog -> chien
horse -> cheval
```

The items() and values() methods

Example:

```
dict = {"cat" : "chat", "dog" : "chien", "horse" : "cheval"}

for french in dict.values():
    print(french)
```

```
chat
chien
cheval
```

Modifying and adding values

modifying and adding values

```
dict = {"cat" : "chat", "dog" : "chien", "horse" : "cheval"}

dict['cat'] = 'minou'
print(dict)
```

```
{'cat': 'minou', 'dog': 'chien', 'horse': 'cheval'}
```

Adding a new key

```
dict = {"cat" : "chat", "dog" : "chien", "horse" : "cheval"}

dict['swan'] = 'cygne'
print(dict)
```

```
{'cat': 'chat', 'dog': 'chien', 'horse': 'cheval', 'swan': 'cygne'}
```

Modifying and adding values

EXTRA

You can also insert an item to a dictionary by using the `update()` method, e.g.:

Example:

```
dict = {"cat" : "chat", "dog" : "chien", "horse" : "cheval"}

dict.update({"duck" : "canard"})
print(dict)
```

```
{'cat': 'chat', 'dog': 'chien', 'horse': 'cheval', 'duck': 'canard'}
```

Modifying and adding values

Removing a key

Note: removing a key will always cause the **removal of the associated value**. **Values cannot exist without their keys**.

Example:

```
dict = {"cat" : "chat", "dog" : "chien", "horse" : "cheval"}

del dict['dog']
print(dict)
```

```
{'cat': 'chat', 'horse': 'cheval'}
```


CISCO