

Fundamentos de Programação

António J. R. Neves
João Manuel Rodrigues

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

Summary

- Functions: definition and invocation
- Parameters and local variables
- Lambda expressions

Functions

- So far, we have only been using the functions that are predefined in Python, such as:

```
name = input("Name? ")
print("Hello", name, "!")
root2 = math.sqrt(2)
```

- But we may also define new functions of our own.

```
def square(x):
    y = x**2
    return y
```

- After definition, we may call our function just like any other.

```
a = 10 + square(2)
b = square(a - 8)
x = 3
print(x, square(1 - square(x-1)) + 1)
```

[Play](#) 

Function definition

- A ***function definition*** specifies the name of a new function, a list of parameters, and a block of statements to execute when that function is called.

Syntax	Example
<pre>def functionName(parameters): statements</pre>	<pre>def hms2sec(h, m, s): sec = (h*60+m)*60+s return sec</pre>

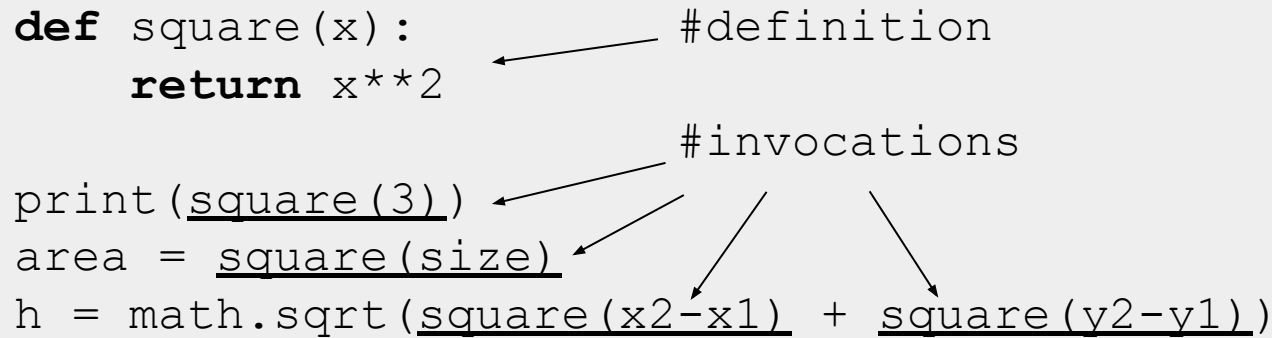
- The first line of the function definition is called the *header*, the indented block is called the *body*.
- The header starts with the **def** keyword and ends with a colon. The body has to be **indented**.
- Function names follow the same rules as variable names.

Definition vs. invocation

- Do not confuse *function **definition*** with *function **invocation*** (aka *function call*)!

```
def square(x):           #definition
    return x**2

print(square(3))         #invocations
area = square(size)
h = math.sqrt(square(x2-x1) + square(y2-y1))
```

A diagram illustrating the difference between function definition and invocation. The code is shown in a light gray box. The first line is a function definition: 'def square(x):' followed by an indented 'return x**2'. An arrow points from the text '#definition' to the 'def' keyword. Below this are three lines of code: 'print(square(3))', 'area = square(size)', and 'h = math.sqrt(square(x2-x1) + square(y2-y1))'. The function calls 'square(3)', 'square(size)', 'square(x2-x1)', and 'square(y2-y1)' are underlined. An arrow points from the text '#invocations' to the first underlined call 'square(3)'. Another arrow points from the same text to the second underlined call 'square(size)'. Two more arrows point from the text '#invocations' to the two underlined calls within the 'math.sqrt' function: 'square(x2-x1)' and 'square(y2-y1)'.

- In a function **definition**, the statements are **not executed**. They are just **stored** for later use.
- They are **executed** only if and **when** the function is **invoked**.
- A function must be defined before being called.
- Define once, call as many times as needed.

Example

```
def hello():  
    print("Hello!")
```

```
def helloTwice():  
    hello()  
    hello()
```

```
#calling the function  
helloTwice()
```

[Play](#) 

- This example contains two function definitions: `hello` and `helloTwice`.
- Then, `helloTwice` is called (invoked).
- When `helloTwice` runs, it calls `hello` twice.

Flow of execution

- Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.
- Function definitions do not alter the flow of execution of the program. They simply store the statements in the function body for later use. The body is not executed at this time.
- A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.

Parameters and arguments

- Some functions require arguments. For example, when you call `math.sin` you pass a number as an argument.
- Some functions take more than one argument: `math.pow` takes two, the base and the exponent.
- When the function is called, the arguments are *values* assigned to *variables* called parameters.

```
def print2times(msg):  
    print(msg)  
    print(msg)
```

`msg="bye"` (*implicit assignment*)

```
print2times("bye")
```



[Play](#) 

Return values

- Some functions, such as `abs` or `math.sin`, produce results, which may be used in expressions or stored in variables.
- Other functions, like `print`, perform an action but don't return a value. They are called void functions. (*Actually, they return the special value `None`.*)
- When a function is being executed and reaches a statement **`return`** `expression` this exits the function and returns the value of the expression to wherever the function was called from.
- A return statement with no expression, **`return`** is the same as **`return`** `None`.

Global vs. local variables

- Variables defined inside a function have a *local scope*.
- Variables defined outside have a *global scope*.
- When you assign to a name inside a function, you create a new local variable even if an identical global name exists.

```
def add(a, b):  
    total = a + b    # Here total is local variable  
    print("Inside: ", total)  
    return total
```

```
total = 0                # This is a global variable  
print(add(10, 20))        # Call add function  
print("Outside: ", total)  
print(a, b)               # ERROR!
```



Parameters are local variables

- Parameters are local variables, too.
- You may modify parameters, but the effect is local!

```
def double(x):  
    x *= 2      # you may modify parameters  
    return x
```



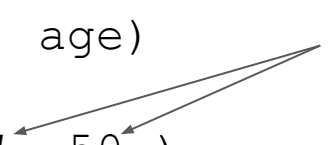
```
x = 3  
y = double(x)  # <=> double(3)  
print(x, y)    # What's the value of x and y?
```

- When the function is called, the parameter receives (just) the value of the argument.
- This form of argument passing is called *pass by value*.

Positional and keyword arguments

- In a function call, **positional arguments** are assigned to parameters according to their position.

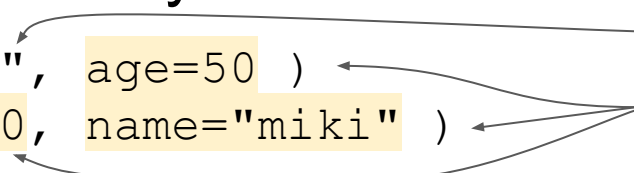
```
def printinfo( name, age ):  
    print("Name:", name)  
    print("Age:", age)  
  
printinfo( "miki", 50 )
```



A diagram with the text "positional arguments" on the right. Two arrows point from this text to the arguments "miki" and "50" in the function call `printinfo("miki", 50)`.

- With **keyword arguments**, the values are assigned to parameters identified by name.

```
printinfo( "miki", age=50 )  
printinfo( age=50, name="miki" )
```



A diagram with the text "positional argument" and "keyword arguments" on the right. An arrow points from "positional argument" to the string "miki" in the first function call. Two arrows point from "keyword arguments" to the `age=50` and `name="miki"` parts in both function calls.

- With keyword arguments you don't have to remember the order of parameters, just their names.
- When mixed, positional must precede keyword arguments.

Default argument values

- A function definition may specify **default argument values** for some of its parameters.

```
def printinfo( name, age=35 ):
    print("Name: ", name)
    print("Age ", age)
```

- When calling the function, if a value is not provided for that argument, it takes the default value.

```
printinfo( "miki", 50 )
printinfo( "miki" )      # here, age is 35!
printinfo( name="miki" ) # same here
```

- This is useful for optional arguments.

```
print(1, 2, 3)
print(1, 2, 3, sep='->')
print(1, 2, 3, sep='->', end='\n-FIM-\n')
```

Variable-length arguments

- (Advanced topic. Not required.)
- You may need to process a function for more arguments than you specified while defining the function.
- These arguments are called variable-length arguments and are not named in the function definition.

```
def printinfo( arg1, *vartuple ):  
    print(arg1)  
    for var in vartuple:  
        print(var)  
printinfo( 10 )  
printinfo( 70, 60, 50 ) #the last two are passed as a tuple
```

- An asterisk (*) is placed before the variable name that holds the values of all non-keyword variable arguments.

Lambda expressions

- A *lambda expression* is an expression whose result is a function.
- You may store it in a variable and use it later, for example.

```
add = lambda a, b: a + b ← #lambda expression  
# Now you can call add as a function  
print("Total: ", add(10, 20))    #Total: 30
```

- They're also known as *anonymous functions*.
- They cannot contain statements, only a single expression.
- They're most useful to pass as arguments to other functions.
- (Examples later in the course.)

Why use functions?

- Defining a function gives a name to a group of statements. This makes the program easier to understand and debug.
- Dividing a long program into functions allows you to develop and debug the parts one at a time and then assemble them into a working whole.
- Functions can be called many times. This eliminates redundant, repetitive code, and makes programs smaller and easier to maintain (if you make a change, you only have to make it in one place).
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Exercises

- Do these [codecheck exercises](#).
- Answer this [review quiz](#).
- What was the [muddiest point](#) in class?

