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:

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

- But we may also <u>define new functions</u> of our own.

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

- After the definition, we can <u>call the function</u> just like any other function.

```python
print(  square(2)  +  square(3)  )
x = 3
print( 2 + square(1-square(x)) )
```

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

def functionName**(**parameters**):**
    statements
	def hms2sec(h, m, s):
	    sec = (h*60+m)*60+s
	    return sec

- The first line of the function definition is called the *header*, the rest 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

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

```python
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

universidade de aveiro
departamento de electrónica,
telecomunicações e informática

- 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 of the built-in functions we have seen 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 assigned to variables called parameters.

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


print2times("bye")
```

`msg="bye"`

Return values

- Some functions, such as the `math` functions, produce results.

- 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`.)*

- The statement
  return `expression`
  exits from a function and returns the result of the expresion.

- A return statement with no argument,
  return
  is the same as **return** `None`.

Global vs. local variables

- Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

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

total = 0                # This is a global variable
print( add(10, 20))      # Call add function
print("Outside the function global total: ", total)
```
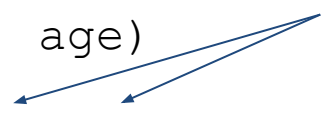
- Parameters are local variables, too.

Keyword arguments

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

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

printinfo( "miki", 50 )
```

positional arguments

- When you use **keyword arguments**, the caller identifies the arguments by the parameter name.

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

keyword arguments

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

Default argument values

universidade de aveiro
deti departamento de electrónica,
telecomunicações e informática

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

```python
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.

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

- This is useful for optional arguments.

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

Variable-length arguments

universidade de aveiro
deti departamento de electrónica,
telecomunicações e informática

- (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

deti · universidade de aveiro
departamento de electrónica,
telecomunicações e informática

- A *lambda expression* is an expression whose result is a function.

- You may store it in a variable and use it later, for example.

```python
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*.

- Lambda forms can take zero or more parameters, but return a single result.

- They cannot contain statements, only a single expression.

- They're most useful to pass as arguments to other functions.

- (Examples later in the course.)