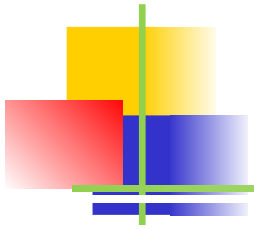

Fundamentos de Programação

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 come with Python, but we may also define new functions.
- A *function definition* specifies the name of a new function and a block of statements to execute when that function is called.

Syntax	Example

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

- 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 **when** the function is **invoked**.
- A function must be defined before being called.
- Define once, call as many times as needed.

Example


```
def print_hello():
    print(“Hello!”)
def repeat_hello():
    print_hello()
    print_hello()
#calling the function
repeat_hello()
```

- This example contains two function definitions: `print_hello` and `repeat_hello`.

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, but remember that statements inside the function are not executed until the function is called.

- 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.
- Inside the function, the arguments are assigned to variables called parameters.

```
def repeat_print(msg):
    print(msg)
    print(msg)
```

- This function assigns the argument to a parameter named `msg`. When the function is called, it prints the value of the parameter (whatever it is).

Return values

- When you create a variable inside a function, it is local, which means that it only exists inside the function. Parameters are also local.
- Some of the functions we are using, such as the `math` functions, produces results. Other functions, like `print`, perform an action but don't return a value. They are called void functions.
- The statement **`return`** `[expression]` exits a function, optionally passing back a result to the caller. A return statement with no argument is the same as `return None`.

Global vs. Local variables

deti universidade de aveiro
departamento de electrónica, telecomunicações e informática

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

```
total = 0  # This is a global variable

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

# Now you can call add function
print( add(10, 20))
print("Outside the function global total: ", total)
```

- Parameters are local variables, too.

Keyword and default arguments

- When you use **keyword** arguments in a function call, the caller identifies the arguments by the parameter name.

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

- A **default** argument value may be provided for a parameter, making it optional. if a value is not provided in the function call for that argument.

```
def printinfo( name, age=35 ):
    print("Name: ", name)
    print("Age ", age)
printinfo( "maria", 23 )
printinfo( "miki" )          # 35 is assumed
```

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 nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

Anonymous Functions

- Not declared in the standard manner by using the `def` keyword.
- You can use the **`lambda`** keyword to create small anonymous functions.

```
# Function definition is here
add = lambda a, b: a + b
# Now you can call add as a function
print(”Total: ", add( 10, 20 )) #Total: 30
```

- Lambda forms can take any number of arguments but return just one value in the form of an expression.
- They cannot contain statements, only a single expression.
- Most useful to pass as arguments to other functions.
- (Examples later in the course.)