



deti

universidade de aveiro
departamento de electrónica,
telecomunicações e informática

Fundamentos de Programação

António J. R. Neves
João Manuel Rodrigues

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro



Summary



deti

universidade de aveiro
departamento de electrónica,
telecomunicações e informática

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

- 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, 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 square(x) : y = x**2 return y</pre>

- 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.

- 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))
```

- 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



deti

universidade de aveiro
departamento de electrónica,
telecomunicações e informática

```
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`.
- There is one invocation to `repeat_hello`.
- When `repeat_hello` runs, it calls `print_hello` twice.

- 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.

- 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 print2times(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).

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

- 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
```

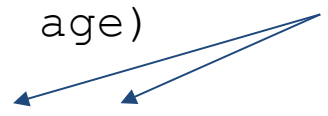
```
# Now you can call add function  
print( add(10, 20))  
print("Outside the function global total: ", total)
```

- Parameters are local variables, too.

- 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 )
```


positional arguments



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

```
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.

- 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
```

- (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.

- 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*.
- 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.)