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.

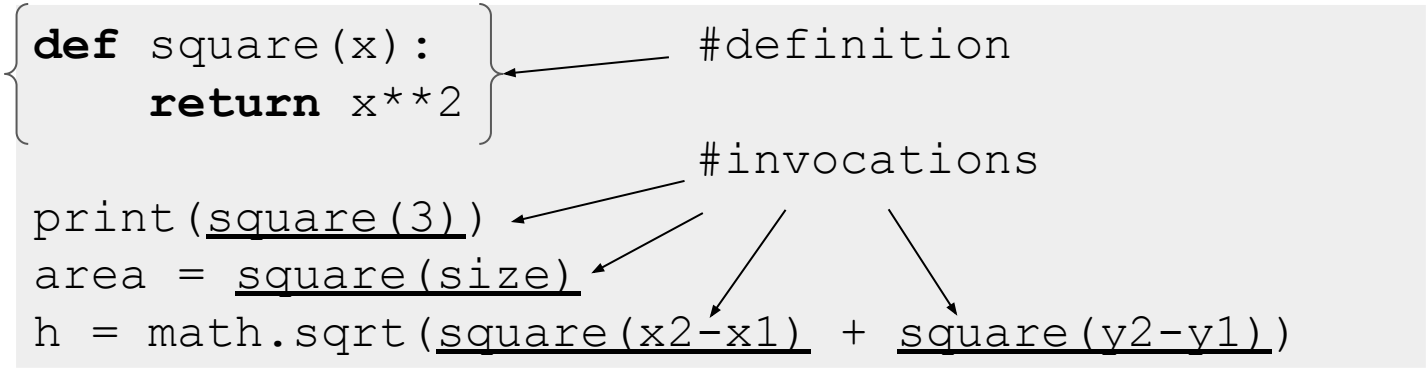
Syntax	Example

`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 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*)!



- 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.
- A **function definition** simply **stores 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 **execute the body** of the function, and **then returns** to pick up where it left off.
- Before executing the body, argument values are **assigned** to function parameters. (See next slide.)

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 in the definition.

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

print2times("bye")
```

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

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`.)*
- The *return statement* can only be used inside a function.

  `return` `expression`
- When executed, it exits the function and returns the value of the expression to wherever the function was called from.
- A return statement with no expression ⇔ **`return`** `None`
- If execution reaches the end of the body ⇔ **`return`** `None`

Global vs. local variables

- Variables defined inside a function have a *local scope*. *Local variables are accessible and changeable only inside their function.*
- Variables defined outside functions have a *global scope*. *Global variables are accessible everywhere.*
- But when you *assign* to a name inside a function, you create a new local variable even if an identical global name exists. *In summary: local names mask global names.*

```
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!
```

Play ▶️

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?
```

Play ▶️

- 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 )
```

positional arguments

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

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

positional argument

keyword arguments

- 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 used for optional arguments in some functions.

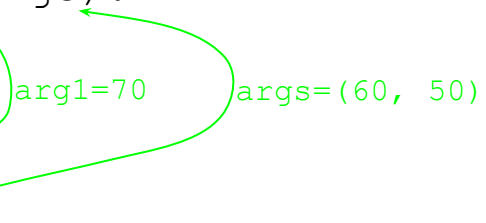
```
print(1, 2, 3)
print(1, 2, 3, sep='->')
print(1, 2, 3, sep='->', end='\n-FIM-\n')
```

Variable-length arguments

- (Advanced topic. Not required.)
- You can define a function to accept a variable number of arguments.
- These so-called *variable-length arguments* are assigned as a tuple to a special parameter in the function definition.

```
def printinfo(arg1, *args):
    print(arg1)
    for var in args:
        print(var)

printinfo(70, 60, 50)  #the last two are passed as a tuple
printinfo(10)
```



- The asterisk (*) indicates the parameter that receives the values of all (positional) 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.
  We'll see *examples later in the course.*

Why use functions?

- **Reusability**:
  - Once defined, a function may be called many times, in the same program or in different programs!
    How many times have you called `print`? Never copied its code!
  - Less redundancy. Smaller programs. Easier to maintain.
- **Abstraction**:
  - To use a function you only have to know **what** it does, **not how** it does it!
    Do you know how `math.sqrt` works? But you've used it!
  - You only need to know the details of how a function works when you define it, not when you call it.
  - This makes programs simpler to understand and debug.

Exercises

- Do these codecheck exercises.

- Answer this review quiz.

- What was the muddiest point in class?