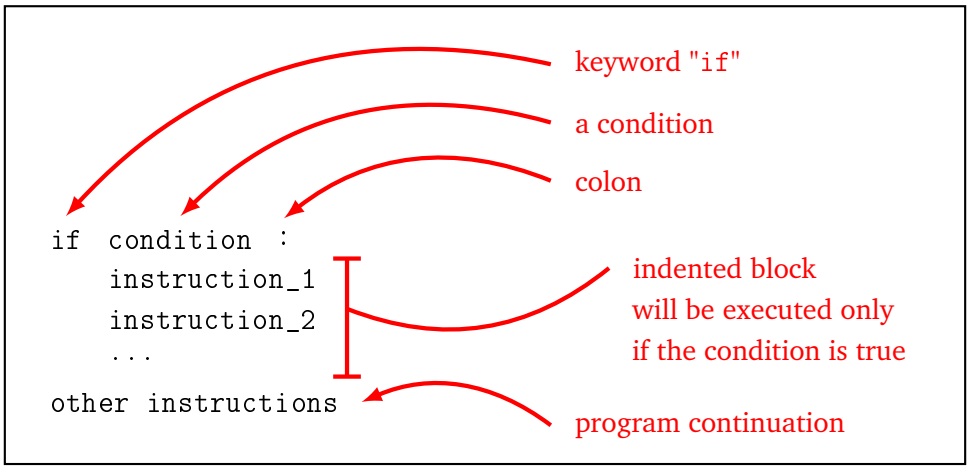
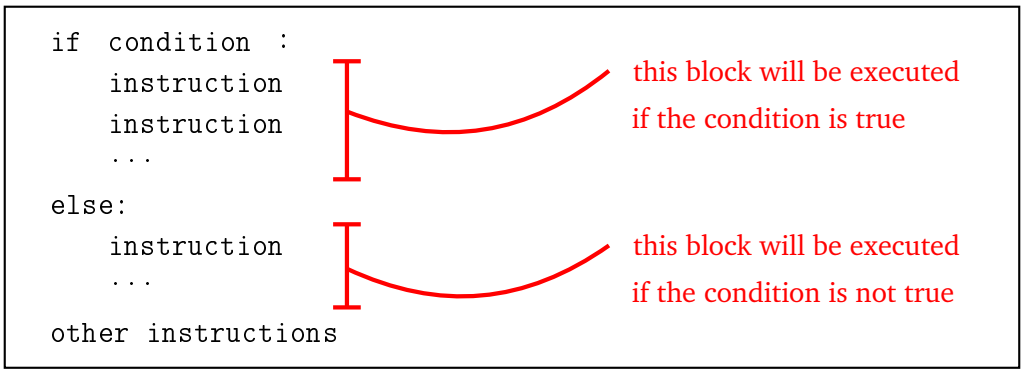
Python survival guide

1. Test and loops

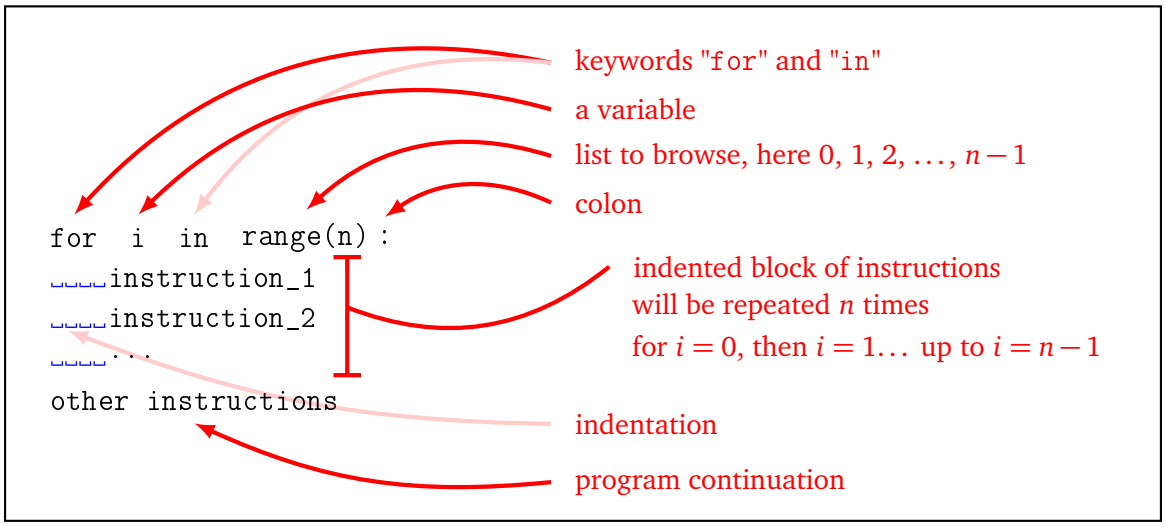
1.1. If ... then ...



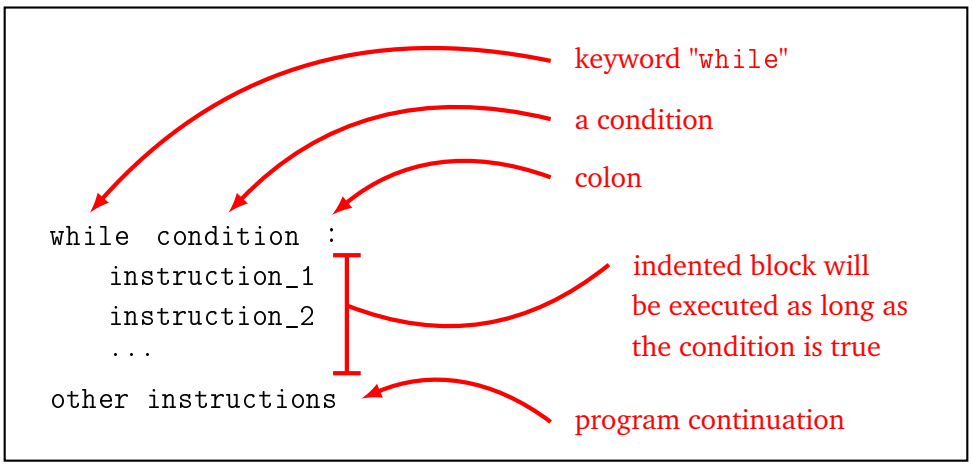
1.2. If ... then ... else ...



1.3. Loop for



1.4. Loop while



1.5. Quit a loop

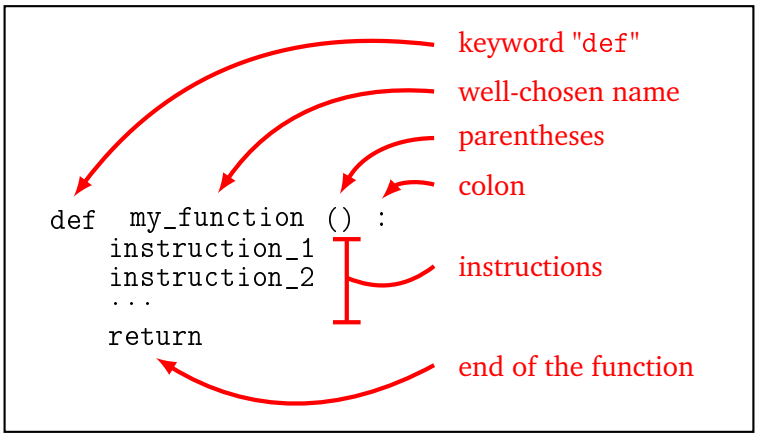
The command `break` immediately exits a “while” loop or a “for” loop.

2. Data type

- `int` Integer. Examples: `123` or `-15`.
- `float` Floating point (or decimal) number. Examples: `4.56`, `-0.001`, `6.022e23` (for 6.022×10^{23}), `4e-3` (for $0.004 = 4 \times 10^{-3}$).
- `str` Character or string. Examples: `'Y'`, `"k"`, `'Hello'`, `"World!"`.
- `bool` Boolean. `True` or `False`.
- `list` List. Example: `[1,2,3,4]`.

3. Define functions

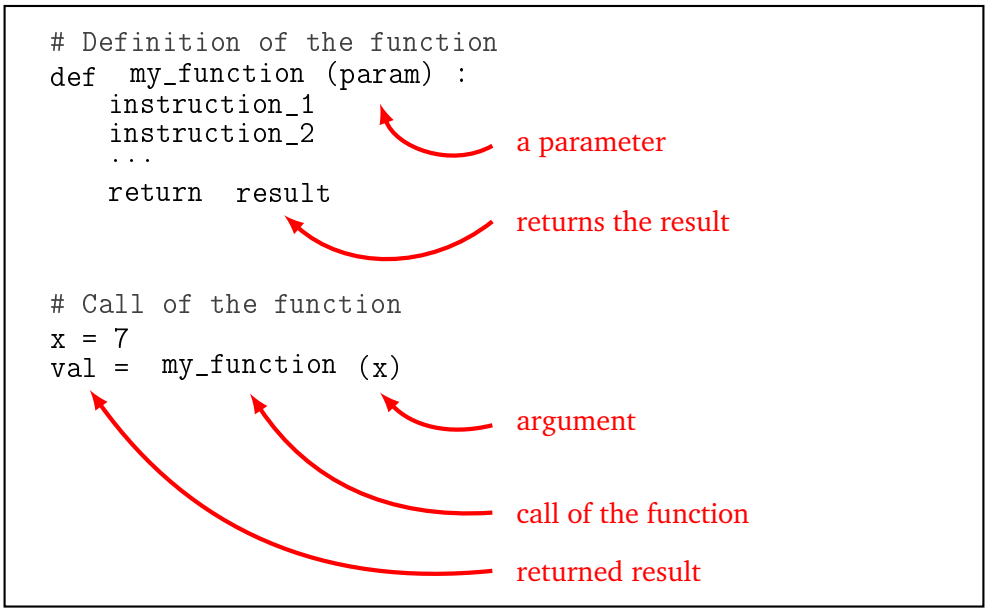
3.1. Definition of a function



3.2. Function with parameter

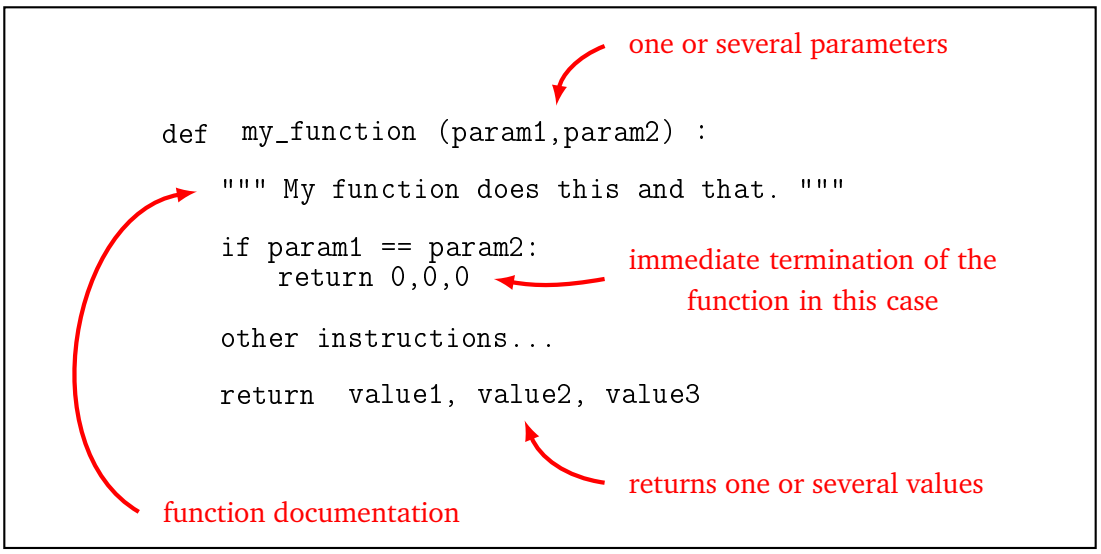
Functions achieve their full potential with:

- an ***input***, which defines variables that serve as ***parameters***,
- an ***output***, which is a result returned by the function (and which will often depend on the input parameters).



3.3. Function with several parameters

There can be several input parameters, there can be several output results.



Here is an example of a function with two parameters and two outputs.

```
def sum_product(a,b):
    """ Computes the sum and product of two numbers. """
    s = a + b
    p = a * b
    return s, p

# Call of the function
mysum, myprod = sum_product(6,7)  # Results
print("The sum is:",mysum)        # Display
print("The product is:",myprod)   # Display
```

- Very important! Do not confuse displaying and returning a value. Display (by the command `print()`) just displays something on the screen. Most functions do not display anything, but instead return one (or more) value. This is much more useful because this value can then be used elsewhere in the program.
- As soon as the program encounters the instruction `return`, the function stops and returns the result. There may be several instances of the `return` instruction in a function but only one will be executed. It is also possible not to put an instruction `return` if the function returns nothing.
- You can, of course, call other functions in the body of your function!

3.4. Comments and docstring

- **Comment.** Anything following the hash sign `#` is a comment and is ignored by `Python`. For example:

```
# Main loop
while r != 0:   # While this number is not zero
    r = r - 1   # Decrease it by one
```

- **Docstring.** You can describe what a function does by starting it with a *docstring*, i.e. a description (in English) surrounded by three quotation marks. For example:

```
def product(x,y):
    """ Compute the product of two numbers
    Input: two numbers x and y
    Output: the product of x and y """
```

```
    p = x * y
    return p
```

3.5. Local variable

Here is a very simple function that takes a number as an input and returns the number increased by one.

```
def my_function(x):
    x = x + 1
    return x
```

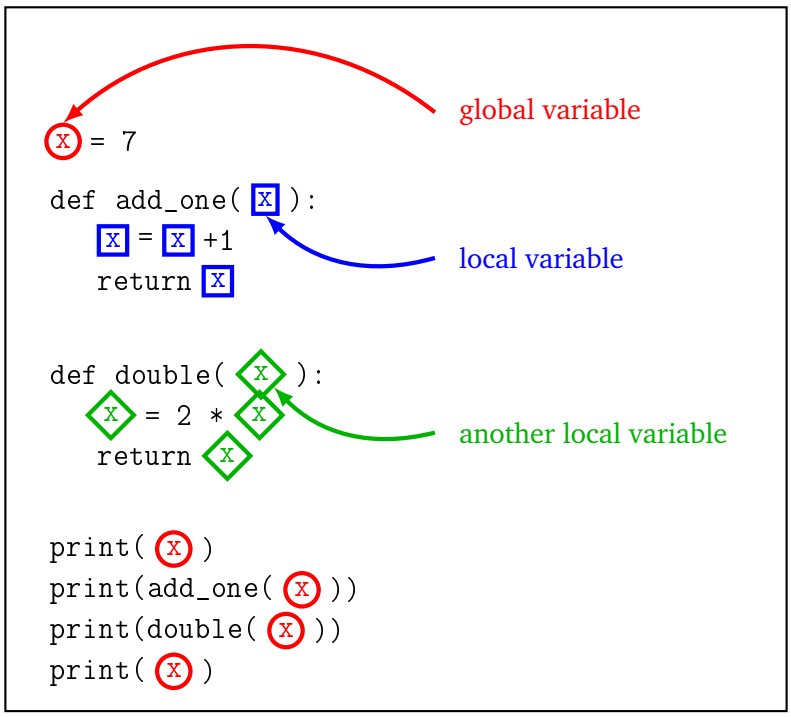
- Of course `my_function(3)` returns 4.
- If I define a variable by `y = 5` then `my_function(y)` returns 6. And the value of `y` has not changed, it is still equal to 5.
- Here is the delicate situation that you must understand:

  ```
  x = 7
  print(my_function(x))
  print(x)
  ```

  - The variable `x` is initialized to 7.
  - The call of the function `my_function(x)` is therefore the same as `my_function(7)` and logically returns 8.
  - What is the value of `x` at the end? The variable `x` is unchanged and is still equal to 7! Even if in the meantime there has been an instruction `x = x + 1`. This instruction changed the `x` inside the function, but not the `x` outside the function.

- Variables defined within a function are called ***local variables***. They do not exist outside the function.
- If there is a variable in a function that has the same name as a variable in the program (like the `x` in the example above), it is as if there were two distinct variables; the local variable only exists inside the function.

To understand the scope of the variables, you can color the global variables of a function in red, and the local variables with one color per function. The following small program defines two functions. The first adds one and the second calculates the double.



The program first displays the value of `x`, so 7, then it increases it by 1, so it displays 8, then it displays twice as much as `x`, so 14. The global variable `x` has never changed, so the last display of `x` is still 7.

3.6. Global variable

A ***global variable*** is a variable that is defined for the entire program. It is generally not recommended to use such variables but it may be useful in some cases. Let us look at an example.
The global variable, here the gravitational constant, is declared at the beginning of the program as a classic variable:

```
gravitation = 9.81
```

The content of the variable `gravitation` is now available everywhere. On the other hand, if you want to change the value of this variable in a function, you must specify to `Python` that you are aware of modifying a global variable.
For example, for calculations on the Moon, it is necessary to change the gravitational constant, which is much lower there.

```
def on_the_moon():
    global gravitation    # Yes, I really want to modify this variable!
    gravitation = 1.625   # New value for the entire program
    ...
```

3.7. Optional arguments

It is possible to create optional arguments. Here is how to define a function (that would draw a line) by giving default values:

```
def draw(length, width=5, color="blue"):
```

- The command `draw(100)` draws my line, and as I only specified the length, the arguments `width` and `color` get the default values (5 and blue).
- The command `draw(100, width=10)` draws my line with a new thickness (the color is the default one).

- The command `draw(100, color="red")` draws my line with a new color (the thickness is the default one).
- The command `draw(100, width=10, color="red")` draws my line with a new thickness and a new color.
- We can also use:
  - `draw(100, 10, "red")`: no need to specify the names of the arguments if you maintain the order.
  - `draw(color="red", width=10, length=100)`: if you name the arguments, then you can pass them in any order.

4. Modules

4.1. Use a module

- `from math import *` Imports all functions from the `math` module. You are now able to use the sine function for example, by `sin(0)`. This is the simplest method and it is the one we use in this book.
- `import math` Allows you to use the functions of the `math` module. You can then access the sine function with `math.sin(0)`. This is the officially recommended method to avoid conflicts between modules.

4.2. Main modules

- `math` contains the main mathematical functions.
- `random` simulates random draws.
- `turtle` the `Python` turtle, it is some equivalent of *Scratch*.
- `matplotlib` allows you to draw graphs and visualize data.
- `tkinter` allows you to display graphics and windows.
- `time` for date, time and duration.
- `timeit` to measure the execution time of a function.

There are many other modules!

5. Errors

5.1. Indentation errors

```
a = 3
 b = 2
```

`Python` returns the error message *IndentationError: unexpected indent*. It also indicates the line number where the indentation error is located, it even points using the symbol "^" to the exact location of the error.

5.2. Syntax errors

- ```
  while x >= 0
      x  = x - 1
  ```
  ```

`Python` returns the error message *SyntaxError: invalid syntax* because the colon is missing after the condition. It should be `while x >= 0 :`

- `string = Hello world!` returns an error because the quotation marks to define the string are missing.
- `print("Hi there"` `Python` returns the error message *SyntaxError: unexpected EOF while parsing* because the expression is incorrectly parenthesized.
- `if val = 1:` Another syntax error, because you would have to write `if val == 1:`

5.3. Type errors

- **Integer**

```
n = 7.0
for i in range(n):
    print(i)
```

`Python` returns the error message *TypeError: 'float' object cannot be interpreted as an integer*. Indeed `7.0` is not an integer, but a floating point number.

- **Floating point number**

```
x = "9"
sqrt(x)
```

`Python` returns the error message *TypeError: a float is required*, because `"9"` is a string and not a number.

- **Wrong number of arguments**
`gcd(12)` `Python` returns the error message *TypeError: gcd() takes exactly 2 arguments (1 given)* because the `gcd()` function of the `math` module requires two arguments, such as `gcd(12,18)`.

5.4. Name errors

- `if y != 0: y = y - 1` `Python` returns the message *NameError: name 'y' is not defined* if the variable `y` has not yet been defined.
- This error can also occur if upper/lower case letters are not scrupulously respected. `variable`, `Variable` and `VARIABLE` are three different variable names.
- `x = sqrt(2)` `Python` returns the message *NameError: name 'sqrt' is not defined*, you must import the `math` module to be able to use the `sqrt()` function.
- **Function not yet defined**

```
product(6,7)

def product(a,b):
    return a*b
```

Returns an error *NameError: name 'product' is not defined* because a function must be defined before it can be used.

5.5. Exercise

Fix the code! Python should display 7 5 9.

```
a == 7
if (a = 2) or (a >= 5)
    b = a - 2
     c = a + 2
else
b = a // 2
c = 2 * a
print(a b c)
```

5.6. Other problems

The program starts but stops along the way or doesn't do what you want? That's where the trouble starts, you have to debug the code! There are no general solutions but only a few tips:

- A clean, well structured, well commented code, with well chosen variable and function names, is easier to read.
- Test your algorithm with paper and pencil for easy cases.
- Do not hesitate to display the values of the variables, to see their evolution over time. For example `print(i,mylist[i])` in a loop.
- A better way to inspect the code is to view the values associated with the variables using the features *debug* of your favorite Python editor. It is also possible to make a step by step execution.
- Does the program work with some values and not others? Have you thought about extreme cases? Is n zero but was not allowed to? Is the list empty, while the program does not handle this case? etc.

Here are some examples.

- I want to display the squares of integers from 1 to 10. The following program does not return any errors but does not do what I want.

  ```
  for i in range(10):
      print(i ** 2)
  ```

  The loop iterates on integers from 0 to 9. You have to write `range(1,11)`.

- I want to display the last item in my list.

  ```
  mylist = [1,2,3,4]
  print(mylist[4])
  ```

  Python returns the error message *IndexError: list index out of range* because the last element is the one of rank 3.

- I want to do a countdown. The next program never stops.

  ```
  n = 10
  while n != "0":
      n = n - 1
      print(n)
  ```

  With a loop "while" you have to take great care to write the condition well and check that it ends up being wrong. Here, it is poorly formulated, it should be `while n!= 0:`