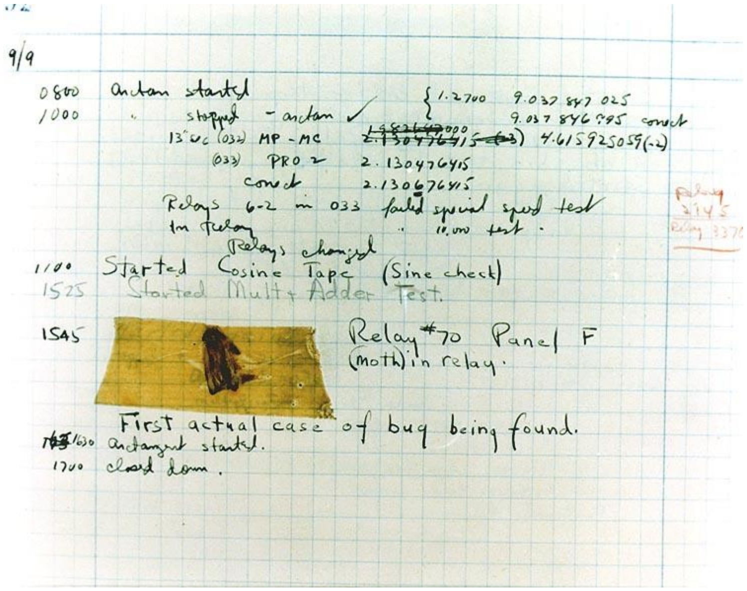
L7: Errors and Debugging

The first *bug* in computing

- The word "bug", means error in programming, which is a resulting fault during the process of programming.



```
9/9

0800   Anctan started
1000     "     stopped - anctan ✓      { 1.2700   9.037 847 025
                                       {          9.037 846 995 conect
        13" uc (032) MP - MC   ~~1.982147000~~
                               2.130476415 (-3)) 4.615925059(-2)
               (033) PRO 2     2.130476415
                     conect    2.130676415
        Relays 6-2 in 033 failed special speed test
        in relay            "      11.000 test.
              Relays changed
1100   Started Cosine Tape (Sine check)
1525   Started Mult + Adder Test.
1545                           Relay #70 Panel F
                               (moth) in relay.
           First actual case of bug being found.
1630   Antangent started.
1700   closed down.
```

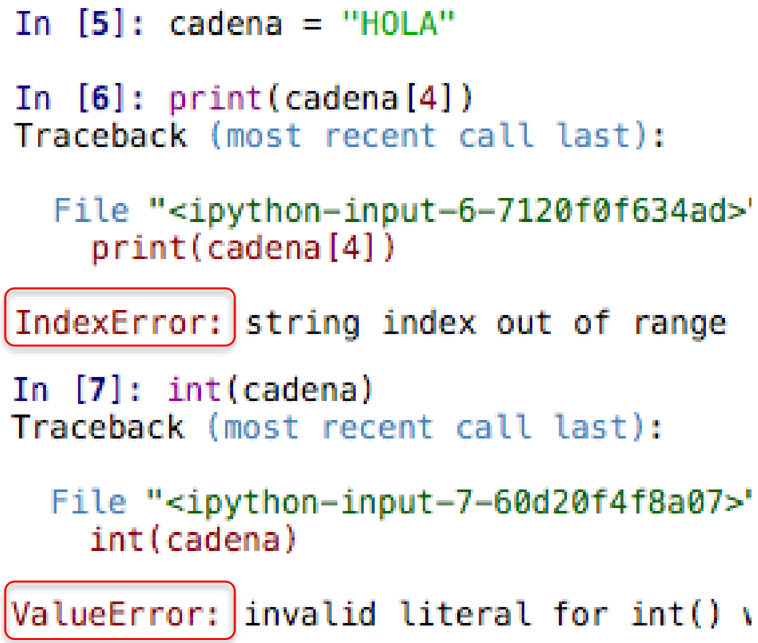
Relay 2145
Relay 3370

- In 1947, engineers working on Mark II informed that the computer failed due to an electromagnetic relay. When investigating this issue, they found a moth (bug) that provoked that the electromagnetic relay remained open.

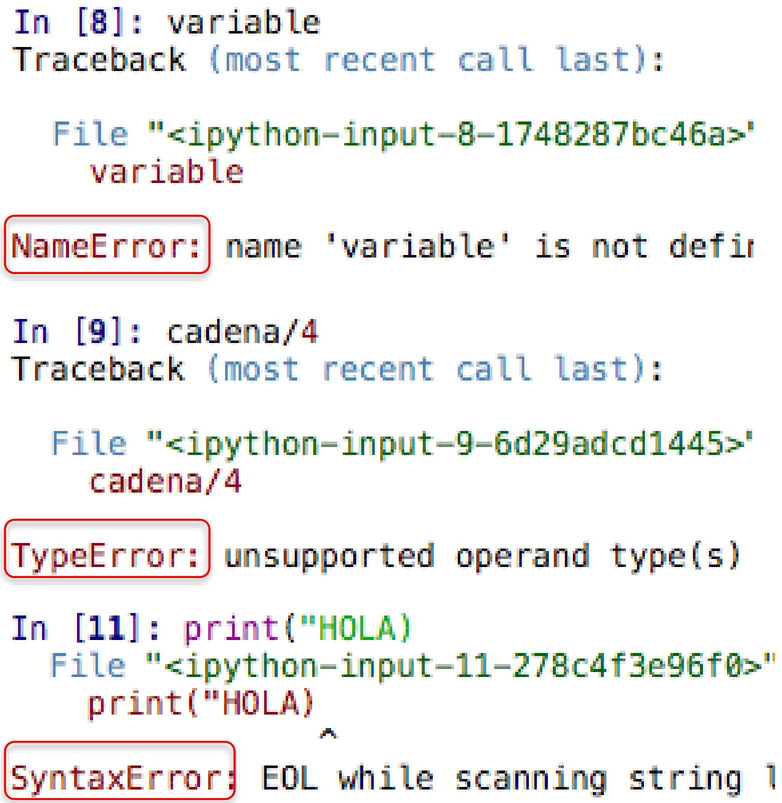
Error messages

Python is an interpreted language, so errors occur when the code is executed.

The interpreter gives us a message that helps us solve the error:



```
In [5]: cadena = "HOLA"

In [6]: print(cadena[4])
Traceback (most recent call last):

  File "<ipython-input-6-7120f0f634ad>'
    print(cadena[4])

IndexError: string index out of range

In [7]: int(cadena)
Traceback (most recent call last):

  File "<ipython-input-7-60d20f4f8a07>'
    int(cadena)

ValueError: invalid literal for int() w
```



```
In [8]: variable
Traceback (most recent call last):

  File "<ipython-input-8-1748287bc46a>'
    variable

NameError: name 'variable' is not defin

In [9]: cadena/4
Traceback (most recent call last):

  File "<ipython-input-9-6d29adcd1445>'
    cadena/4

TypeError: unsupported operand type(s)

In [11]: print("HOLA)
  File "<ipython-input-11-278c4f3e96f0>"
    print("HOLA)
               ^
SyntaxError: EOL while scanning string l
```

Logic errors: the hardest ones

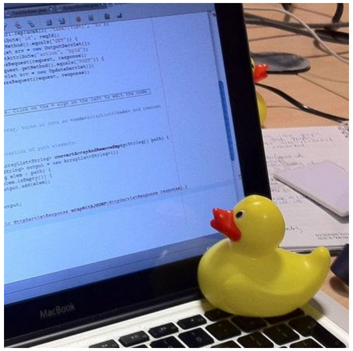
The code works, but it does not do what is expected. These are the hardest to find because the program does not return any error.

What to do?

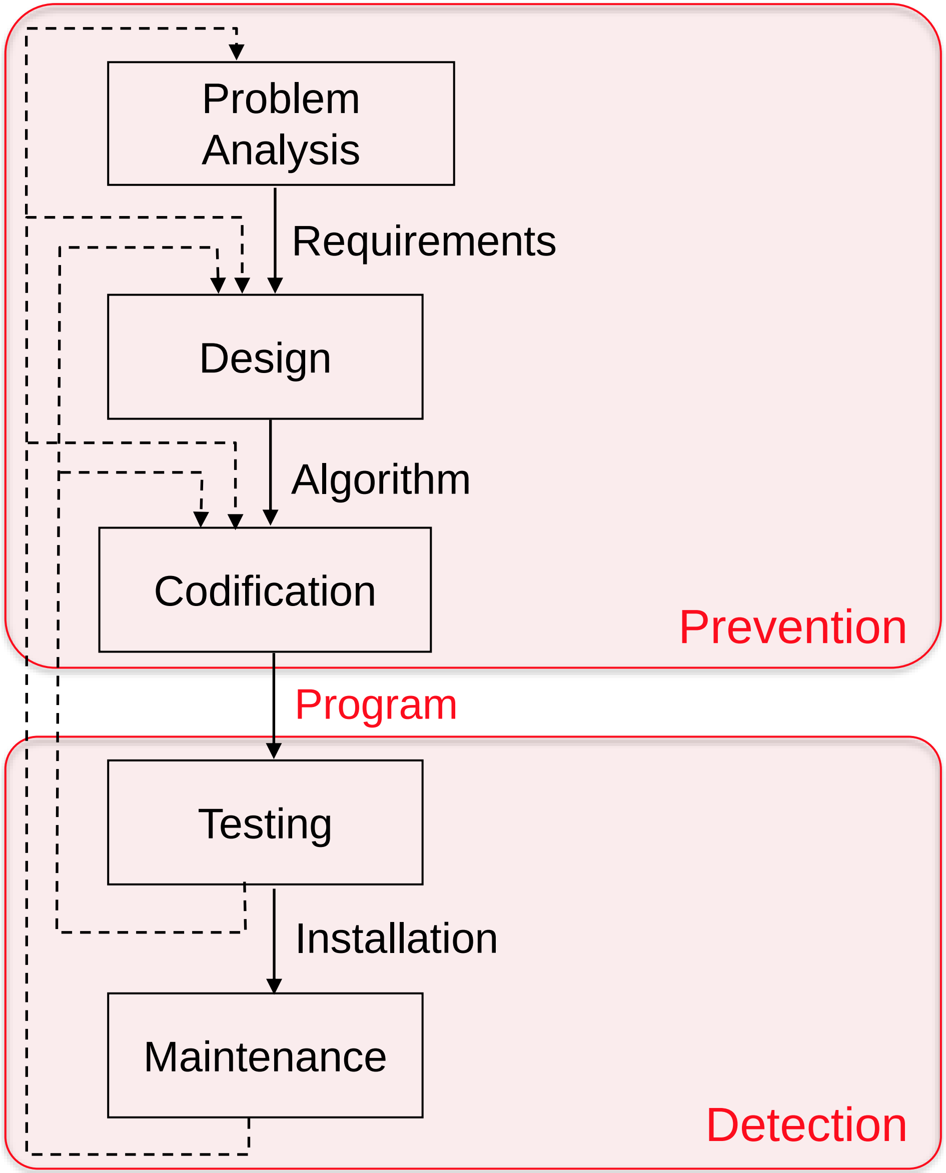
- Understand
  - Before writing any line of code, think in the solution

- Model
  - Write diagrams (flowcharts)
  - Use tools for algorithm design

- Interact
  - Explain your solution to somebody... A rubber duck debugging: debug the code forcing you to explain it, line by line, to the duck



Origin of errors



➡ Preventive Programming

➡ Test/Validation Debugging

Actions to perform

From the beginning,
design the code to facilitate the detection of errors

⬇

- Preventive programming:
  - Analysis: Write the specifications of the functions
  - Design: Apply the concept of modularity (divide and conquer)
  - Coding: Check the conditions of input/output (asserts)

- Test / Validation
  - Compare input/output with the expected ones defined at the specifications of functions (analysis stage)
  - Design the set of tests

- Debugging
  - Study the events that cause an error

Actions to perform

From the beginning,
design the code to facilitate the detection of errors

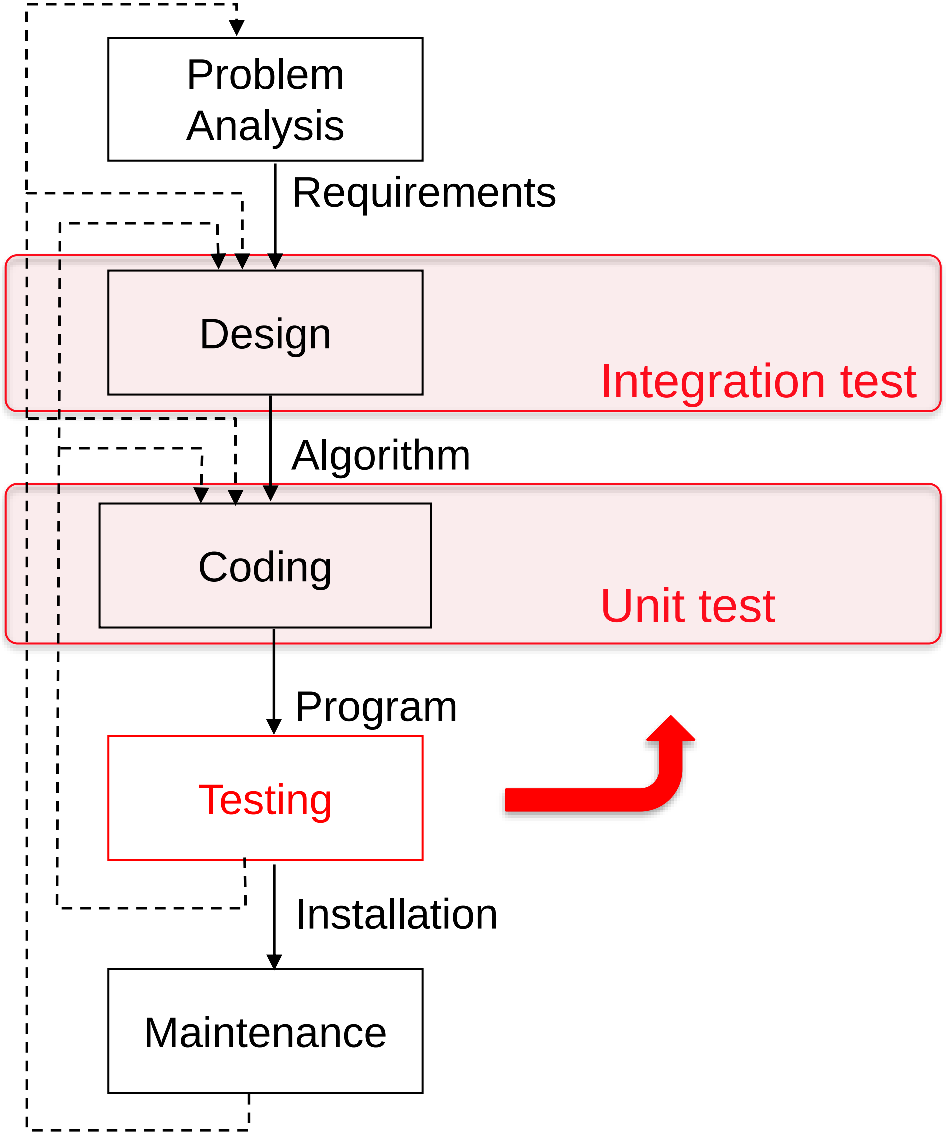
⬇

- Preventive programming:
  - Analysis: Write the specifications of the functions
  - Design: Apply the concept of modularity (divide and conquer)
  - Coding: Check the conditions of input/output (asserts)

- Test / Validation
  - Compare input/output with the expected ones defined at the specifications of functions (analysis stage)
  - Design the set of tests

- Debugging
  - Study the events that cause an error

Test / Validation



Problem Analysis

Requirements

Design

Integration test

Algorithm

Coding

Unit test

Program

Testing

Installation

Maintenance

Black box testing
White box testing

Black Box testing

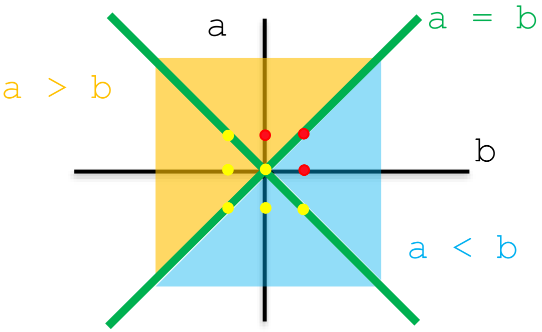
Definition: to define test cases only from the specification of the function, not the implementation of the function.

Example:

```
def maxim(a,b):
    """
    Function maxim (a,b)
    Returns the max of two vàlues: the two input parameters

    If both values are equal, it returns this value.
    """
```

Test cases: without checking the code.

```
a = b ⟶ a=1 b=1
a > b ⟶ a=1 b=0
a < b ⟶ a=0 b=1
```



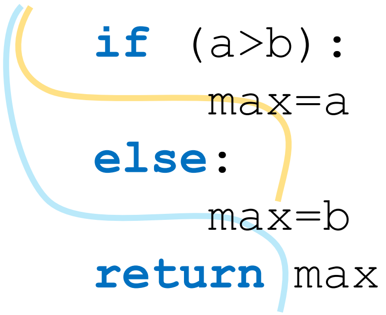
White Box testing

Definition: to define test cases only knowing how the function has been implemented. Objective: to visit each potential path at least once.

Example:

```
def maxim(a,b):
    """
    Function maxim (a,b)
    Returns the max of two vàlues: the two input parameters

    If both values are equal, it returns this value.
    """
```

Test cases: from the code.



```
if (a>b):
    max=a
else:
    max=b
return max
```

```
if (a > b)  →  a=1 b=0
else        →  a=0 b=1
```

Guide:

Conditionals:

- All condition cases (True,False)

Iterations:

- Conditions of input/output,
- Repeat once
- Repeat more than 1 time

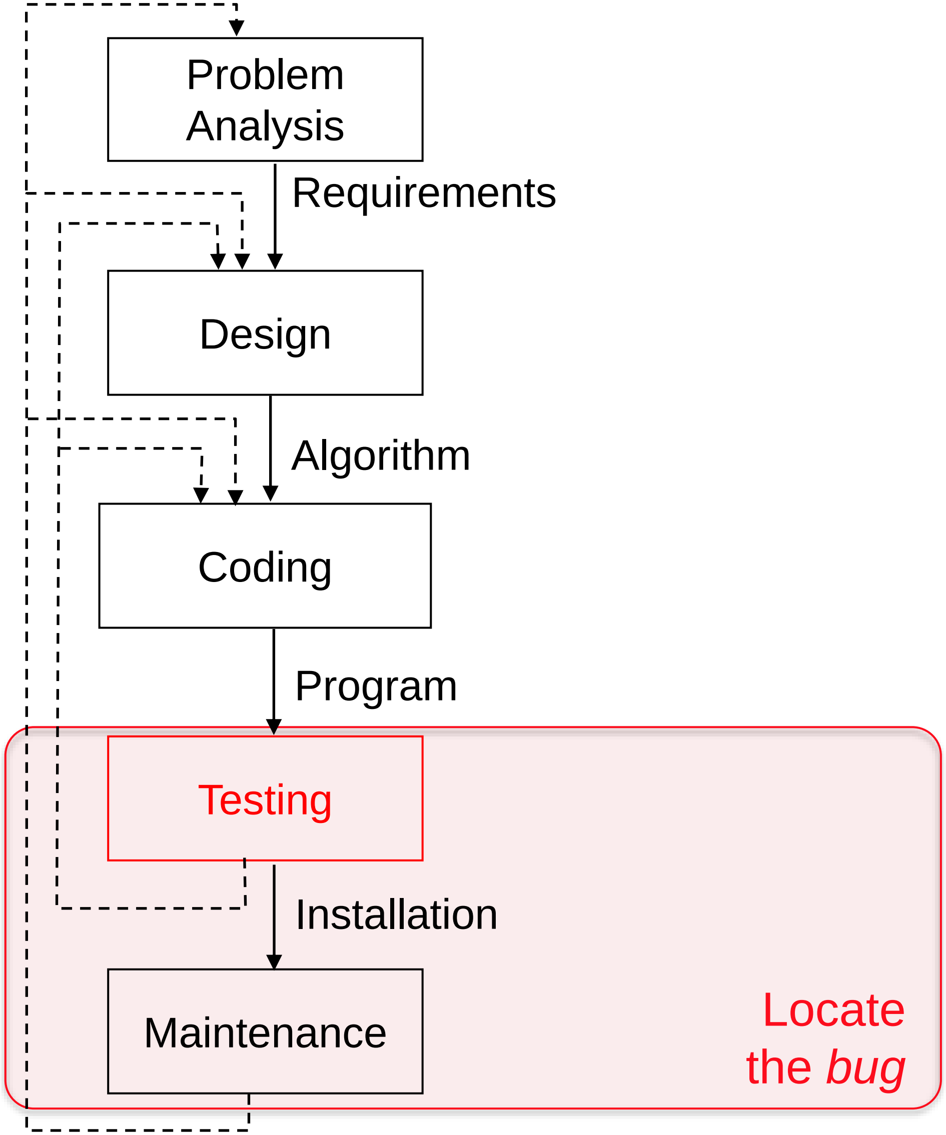
Actions to perform

From the beginning,
design the code to facilitate the detection of errors

⬇

- Preventive programming:
  - Analysis: Write the specifications of the functions
  - Design: Apply the concept of modularity (divide and conquer)
  - Coding: Check the conditions of input/output (asserts)
- Test / Validation
  - Compare input/output with the expected ones defined at the specifications of functions (analysis stage)
  - Design the set of tests
- Debugging
  - Study the events that cause an error

Test / Validation

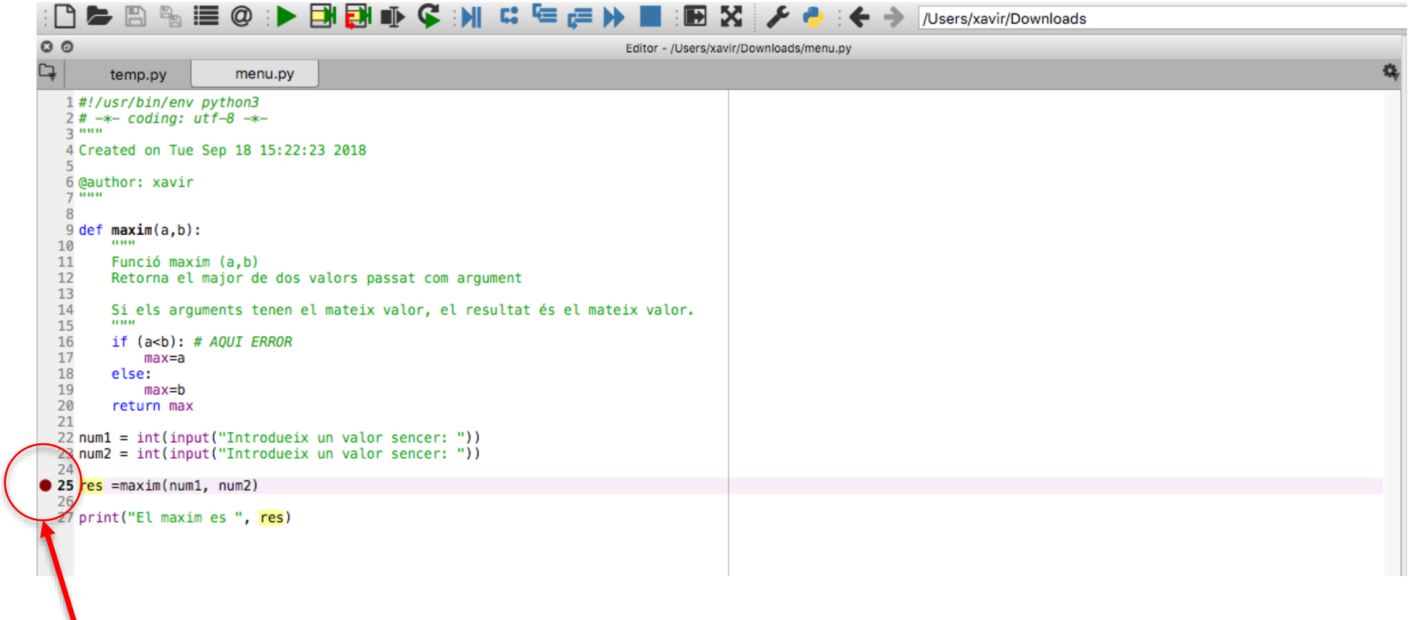


Debug IDE:
`print`

Spider Debug

Debug: a tool to analyse the code. It allows us to reproduce step by step how an unexpected result has occurred that causes a malfunction.

Break Points: when (exact instruction) we want the execution of the program to stop.

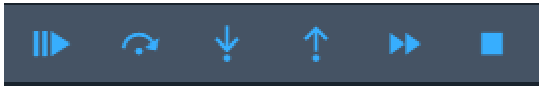


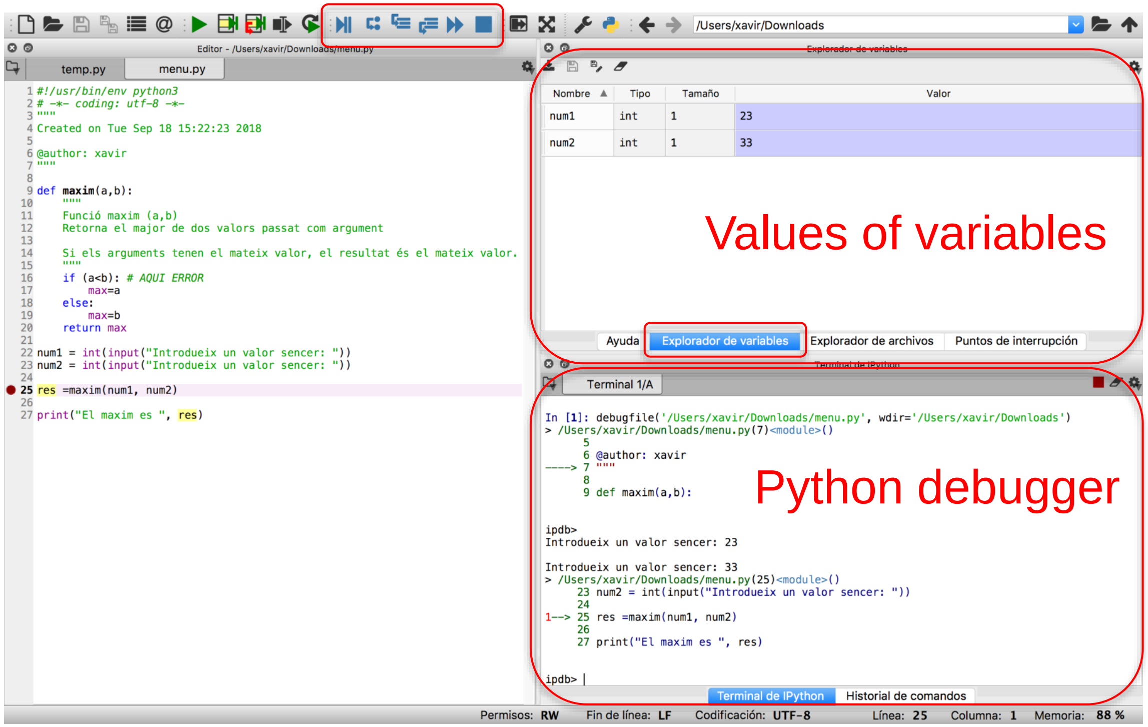
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Sep 18 15:22:23 2018

@author: xavir
"""

def maxim(a,b):
    """
    Funció maxim (a,b)
    Retorna el major de dos valors passat com argument

    Si els arguments tenen el mateix valor, el resultat és el mateix valor.
    """
    if (a<b): # AQUI ERROR
        max=a
    else:
        max=b
    return max

num1 = int(input("Introdueix un valor sencer: "))
num2 = int(input("Introdueix un valor sencer: "))

res =maxim(num1, num2)

print("El maxim es ", res)
```

Double click before the line of code

Spider Debug

Windows:





```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Sep 18 15:22:23 2018

@author: xavir
"""

def maxim(a,b):
    """
    Funció maxim (a,b)
    Retorna el major de dos valors passat com argument

    Si els arguments tenen el mateix valor, el resultat és el mateix valor.
    """
    if (a<b): # AQUI ERROR
        max=a
    else:
        max=b
    return max

num1 = int(input("Introdueix un valor sencer: "))
num2 = int(input("Introdueix un valor sencer: "))

res =maxim(num1, num2)

print("El maxim es ", res)
```

Nombre ▲	Tipo	Tamaño	Valor
num1	int	1	23
num2	int	1	33

Values of variables

```
In [1]: debugfile('/Users/xavir/Downloads/menu.py', wdir='/Users/xavir/Downloads')
> /Users/xavir/Downloads/menu.py(7)<module>()
      5
      6 @author: xavir
----> 7 """
      8
      9 def maxim(a,b):


ipdb>
Introdueix un valor sencer: 23

Introdueix un valor sencer: 33
> /Users/xavir/Downloads/menu.py(25)<module>()
     23 num2 = int(input("Introdueix un valor sencer: "))
     24
1--> 25 res =maxim(num1, num2)
     26
     27 print("El maxim es ", res)


ipdb> |
```

Python debugger

Permisos: RW Fin de línea: LF Codificación: UTF-8 Línea: 25 Columna: 1 Memoria: 88 %

Actions:

Start debugging the code

Execute the line

Execute the function

Exit from the function

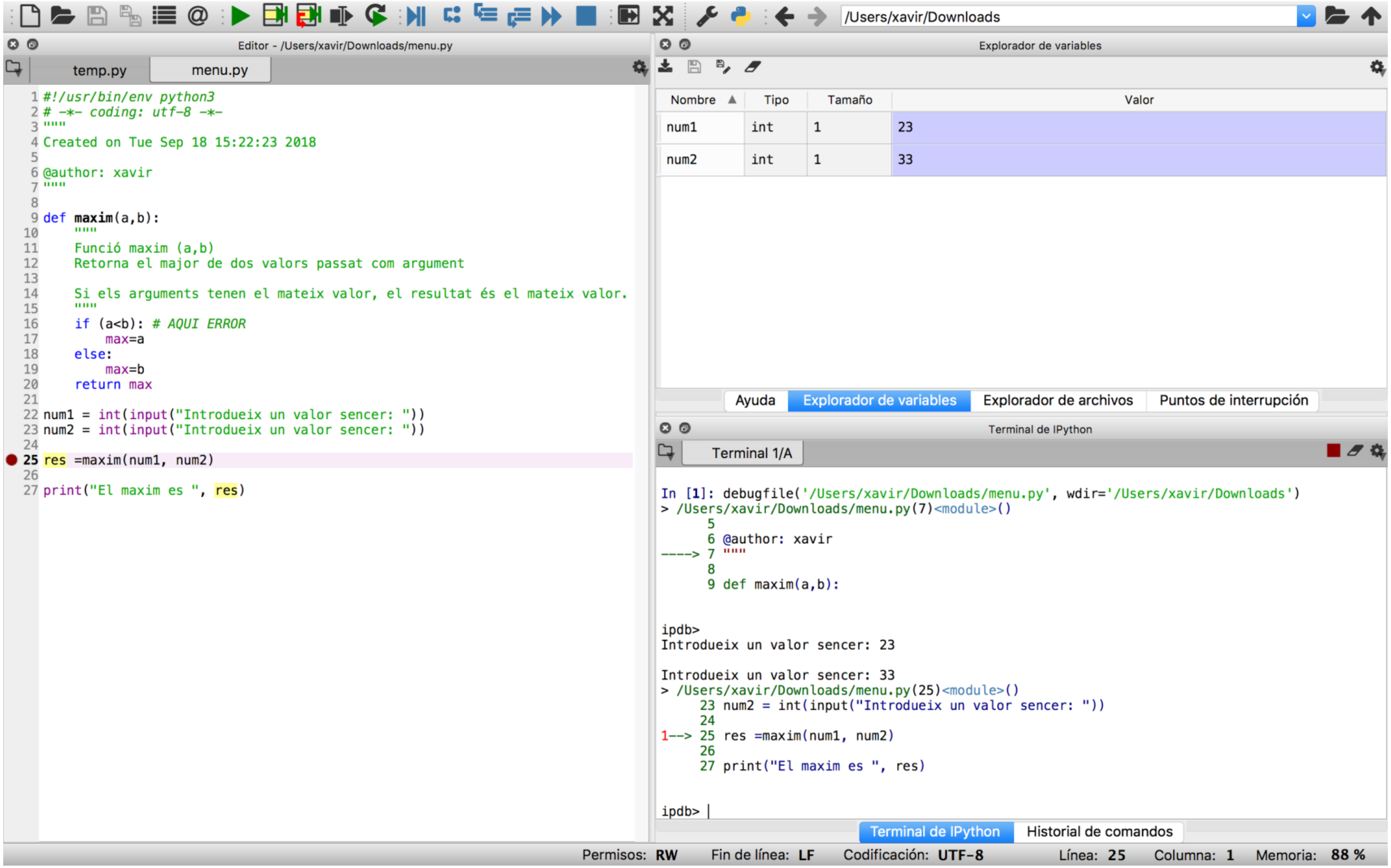
Jump (execute) to the next break point

Stop the debugger

Spider Debug

```
def maxim(a,b):
    """
    Function maxim (a,b)
    Returns the max of two vàlues: the two input parameters

    If both values are equal, it returns this value.
    """
    if (a<b): # ERROR HERE
        max=a
    else:
        max=b
    return max

num1 = int(input("Write an integer: "))
num2 = int(input("Write an integer: "))

● res =maxim(num1, num2)

print("The max is", res)
```

Spider Debug



```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Sep 18 15:22:23 2018

@author: xavir
"""

def maxim(a,b):
    """
    Funció maxim (a,b)
    Retorna el major de dos valors passat com argument

    Si els arguments tenen el mateix valor, el resultat és el mateix valor.
    """
    if (a<b): # AQUI ERROR
        max=a
    else:
        max=b
    return max

num1 = int(input("Introdueix un valor sencer: "))
num2 = int(input("Introdueix un valor sencer: "))

res =maxim(num1, num2)

print("El maxim es ", res)
```

Nombre ▲	Tipo	Tamaño	Valor
num1	int	1	23
num2	int	1	33

```
In [1]: debugfile('/Users/xavir/Downloads/menu.py', wdir='/Users/xavir/Downloads')
> /Users/xavir/Downloads/menu.py(7)<module>()
      5
      6 @author: xavir
----> 7 """
      8
      9 def maxim(a,b):


ipdb>
Introdueix un valor sencer: 23

Introdueix un valor sencer: 33
> /Users/xavir/Downloads/menu.py(25)<module>()
     23 num2 = int(input("Introdueix un valor sencer: "))
     24
1--> 25 res =maxim(num1, num2)
     26
     27 print("El maxim es ", res)


ipdb> 
```

Permisos: RW   Fin de línea: LF   Codificación: UTF-8   Línea: 25   Columna: 1   Memoria: 88 %

Summary: debug and validation / test

When programming

Not recommended	Recommended
Write all the program	Write one function
Test all the program at the same time	Test the function, debug the function
Debug the whole program	Perform Integration tests

Correcting bugs

Not recommended	Recommended
Make changes, directly	Make a backup, then make changes
To “remember”, but not writing down, the potential bugs	Write down the potential bugs
Test, and when it works, forget it!	Test, and when it works, write in the documentation what was done
Panic! Nothing works!!	Stay calm! Compare with previous versions

Actions to perform

From the beginning,
design the code to facilitate the detection of errors

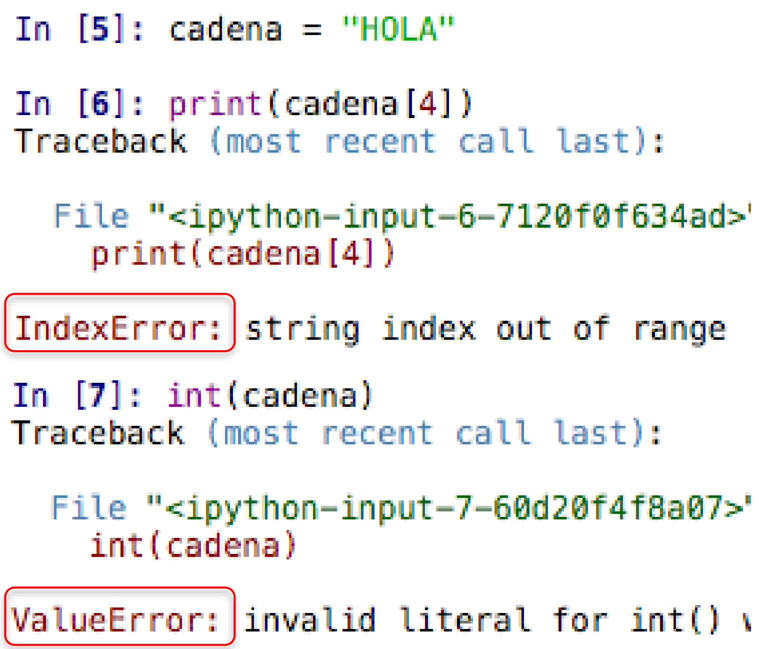
⬇

- Preventive programming:
  - Analysis: Write the specifications of the functions
  - Design: Apply the concept of modularity (divide and conquer)
  - Coding: Check the conditions of input/output (asserts)
- Test / Validation
  - Compare input/output with the expected ones defined at the specifications of functions (analysis stage)
  - Design the set of tests
- Debugging
  - Study the events that cause an error

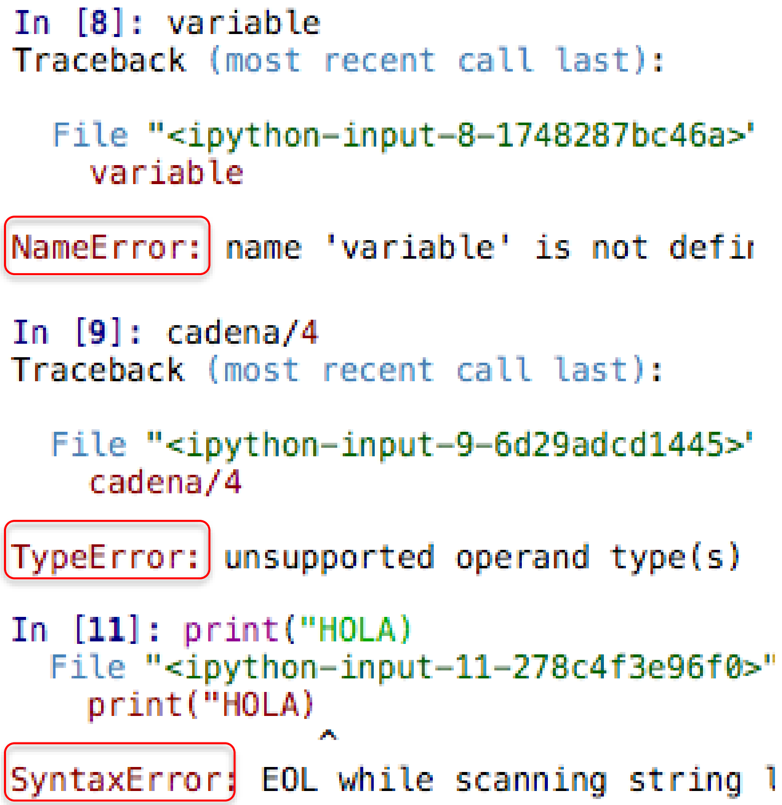
Exceptions and assertions

What happens when the execution of a program is in an unexpected condition (situation)?

Python generates an **exception** ... at the expected condition



```
In [5]: cadena = "HOLA"

In [6]: print(cadena[4])
Traceback (most recent call last):

  File "<ipython-input-6-7120f0f634ad>'
    print(cadena[4])

IndexError: string index out of range

In [7]: int(cadena)
Traceback (most recent call last):

  File "<ipython-input-7-60d20f4f8a07>'
    int(cadena)

ValueError: invalid literal for int() \
```



```
In [8]: variable
Traceback (most recent call last):

  File "<ipython-input-8-1748287bc46a>'
    variable

NameError: name 'variable' is not defi

In [9]: cadena/4
Traceback (most recent call last):

  File "<ipython-input-9-6d29adcd1445>'
    cadena/4

TypeError: unsupported operand type(s)

In [11]: print("HOLA)
  File "<ipython-input-11-278c4f3e96f0>"
    print("HOLA)
               ^
SyntaxError: EOL while scanning string l
```

Other exceptions

- `SyntaxError`: Python can not analyze the program
- `NameError`: the name of the variable is not defined
- `AttributeError`: an invalid attribute reference is made
- `ValueError`: the type of the operand is valid but not its value
- `IOError`: The input/output systems have detected an error
  (e.g. the filename does not exist)

Complete list of exceptions: https://docs.python.org/3.12/library/exceptions.html

Treating exceptions

In Python it is not only possible to identify and track errors, but to take corrective and preventive actions.

Python provides handlers to handle exceptions.

```
try:
    a = int(input("Numerator: "))
    b = int(input("Denominator: "))
    print(a/b)
except:
    print("ERROR: Incorrect values")
```

“Risky” code block

If there is an exception

Exceptions **raised** by any instruction in the test body `try`, are **handled** by the `except` statement.

In case of an error, the execution continues with the body of the statement `except`

Treating specific exceptions

In case of having different treatments for each exception, Python provides a mechanism to manage each one.

```
try:
    a = int(input("Numerator: "))
    b = int(input("Denominator: "))
    print(a/b)
except ValueError:
    print("ERROR: Incorrect values.")
except ZeroDivisionError:
    print("ERROR: Division by zero.")
except:
    print("ERROR: Something went wrong.")
```

Exception:

Input is not a number

Denominator = 0

Other errors

In case of identifying an error, for exemple, `ValueError` or `ZeroDivisionError`, the code for each specific error will be executed.

Other non-identified errors will be treated in the last exception.

Else, finally

In addition to except, Python has two other resources that complete the handling of exceptions:

The instruction `else`

As with the conditional `if`, the `else` statement is executed when no exception is thrown.

The instruction `finally`

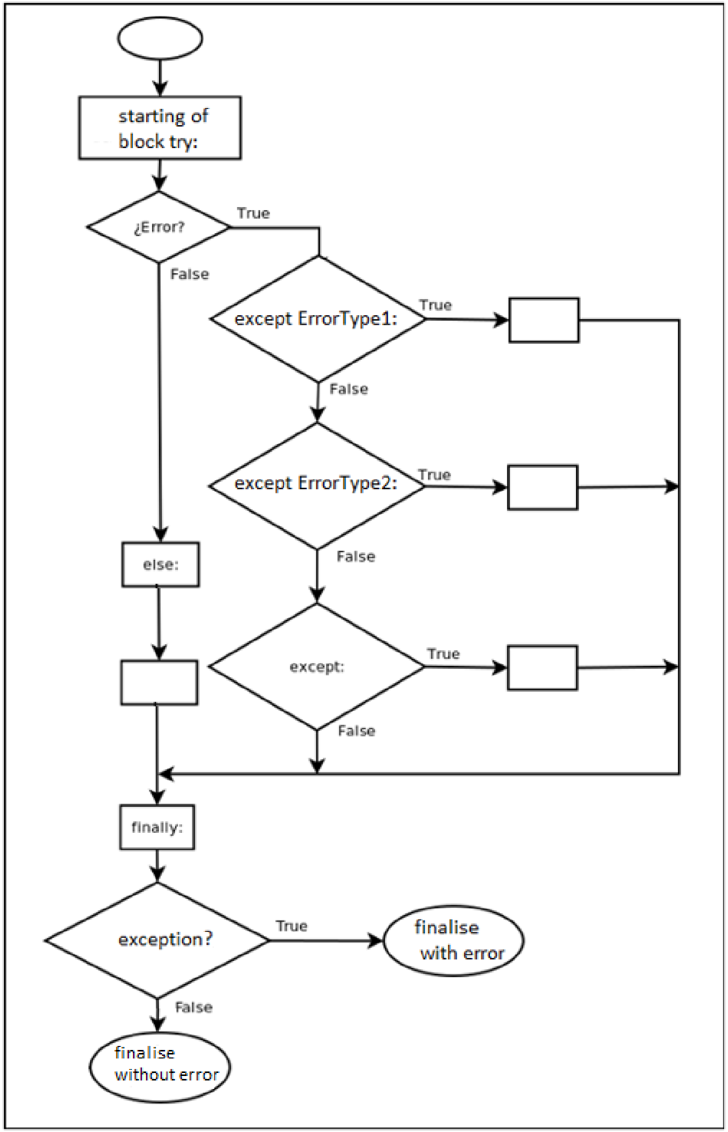
In some situations, some operations must be performed regardless whether or not an exception has occurred.

This is common when closing an internet communication, closing access to a database, or closing a file in write mode.

For these situations the `finally` statement is used, which is executed no matter if there was an exception or not.

else i *finally*

```
error = True

try:
    a = int(input("Numerator: "))
    b = int(input("Denominator: "))
    print(a/b)
except ValueError:
    print("ERROR: Incorrect values.")
except ZeroDivisionError:
    print("ERROR: Division by zero.")
except:
    print("ERROR: Something went wrong.")
else:
    error = False
finally:
    message = "The operation was "
    if error:
        message +="NOT "
    message += "correctly performed."
    print(message)
```



Exemple: Calculator

```
try:
    operand1 = float(input("First operand: "))
    operador = input("Operation: ")
    operand2 = float(input("Second operand: "))

    if (operador == "+"):    #Sum
        result = operand1 + operand2
    elif (operador == "-"):  #Rest
        result = operand1 - operand2
    elif (operador == "*"):  #Product
        result = operand1 * operand2
    elif (operador == "/"):  #Division
        result = operand1 / operand2
    else:                    #Operation not defined
        raise SyntaxError("Operation not defined")

except ZeroDivisionError:
    print("ERROR: Division by zero")
except ValueError:
    print("ERROR: Operands must be numbers")
except SyntaxError as message:
    print("ERROR:", message)
else:
     print(operand1,operador,operand2," = ",result)
```

What to do when we find a bug?

We write a function… but how do we handle possible errors?

- Make a silent error?
  - To solve the error or to do nothing
  - Bad idea! User has not feedback

```
def square_root(num):
    if(num>=0):
        result =(num)**0.5
    else:
        result =(-num)**0.5

    return result
```

- Return a value of error?
  - Problem, which value to choose?
  - Choosing a value complicates the program
  - It must be documented

```
def square_root(num):
"""
Returns the square root
If error, it returns -1
"""
    if(num>=0):
        result =(num)**0.5
    else:
        result = -1
    return result
```

- Stop execution and return an error message:
  - Python `raise` an exception
  - `raise <type> (message)`

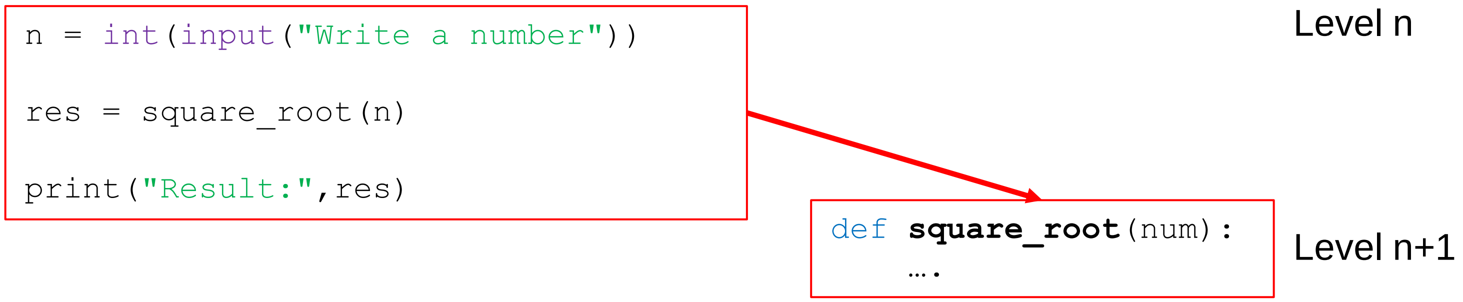
```
def square_root(num):
    if(num>=0):
        result =(num)**0.5
    else:
        raise ValueError ("ERROR: Function domain not defined")
    return result
```

Name error to raise

Optional
Message explaining what happened

Exceptions as workflow

When we call a function, we are setting a hierarchy



```
n = int(input("Write a number"))

res = square_root(n)

print("Result:",res)
```

Level n

```
def square_root(num):
    ….
```

Level n+1

- Error message (`print`)
  - We invert the hierarchy, the function decides how to handle the error
- Return an error value
  - We break the principle of abstraction, because we need to give additional information on how the function is coded (inside).

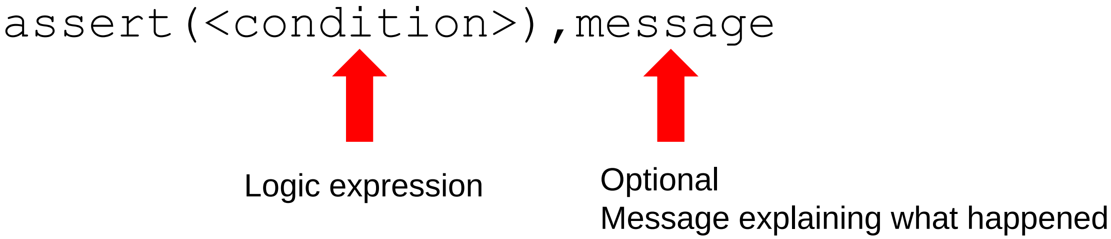
- Raise (`raise`) is the way to communicate that the operation could not be performed. It is from the place where the function is called that the management of errors are done (`try/excepts`).

Assert

On certain occasions and particularly in development environments where security is critical, conditions can be identified where one wants to ensure that assumptions about the state of computing, or the value of data, are as expected.

The instruction `assert`

Provokes an exception of type *AssertionError* if the result of the logic expression is `False`.



```
assert(<condition>),message
```

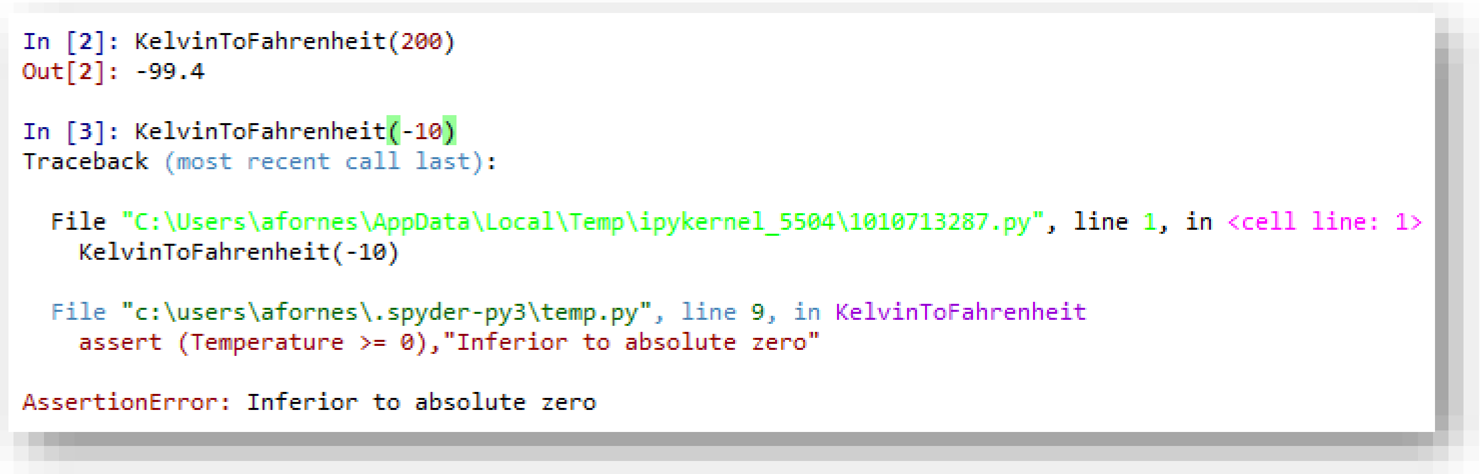
Logic expression

Optional
Message explaining what happened

The use of the instruction `assert` is a good practice of defensive programming.

Example: KelvinToFahrenheit

```
def KelvinToFahrenheit(Temperature):
    assert (Temperature >= 0),"Inferior to absolute zero"
    return ((Temperature-273)*1.8)+32
```



```
In [2]: KelvinToFahrenheit(200)
Out[2]: -99.4

In [3]: KelvinToFahrenheit(-10)
Traceback (most recent call last):

  File "C:\Users\afornes\AppData\Local\Temp\ipykernel_5504\1010713287.py", line 1, in <cell line: 1>
    KelvinToFahrenheit(-10)

  File "c:\users\afornes\.spyder-py3\temp.py", line 9, in KelvinToFahrenheit
    assert (Temperature >= 0),"Inferior to absolute zero"

AssertionError: Inferior to absolute zero
```

```
""" Main Program """
try:
    tempK = float(input("Temperature in Kelvin: "))
    tempF = KelvinToFahrenheit(tempK)
except ValueError:
    print("ERROR: Temperature must be a number")
except AssertionError as message:
    print("ERROR:", message)
else:
    print(tempK,"degrees Kelvin are",tempF,"degrees Fahrenheit")
```

Assert for defensive programming

The use of the instruction `assert` is a good practice of defensive programming.

`Asserts` do not allow the programmer to control the response to unexpected conditions, but...

- It ensures that execution stops when the expected condition is not met
- It is usually used to check input parameters to functions
- It can be used to check the outputs of a function to prevent the spread of incorrect values
- It can make the location of the source of an error much easier