Logging and Debugging

This lesson will deal on how to make sure that our code is doing what is supposed to be doing.

This is related to testing, but can be viewed as complementary to it

We will discuss:

- debugging
- assertions
- logging
- warnings
- linters

All these tools help us ensure that the program is doing what is supposed to be doing.

While tests check that the program logic is correct, these tools helps you check that the program is implementing the operations you think it's doing.

the logic

most of these techniques are necessary to help you understand what you program is *actually* doing, instead of what you think it is supposed to be doing

Debugging

A debugger is a program that attach itself to yours and monitor it state and allow you to control the execution in real time.

It doesn't sound that crazy for python, but consider that the original debuggers were doing it for C programs...

What we do when we execute our programs one line at the time from an editor like spyder is basically a manual for of a debugger

Python and IPython comes with a basic debugger for the command line, but most programs provide you with more advanced (and easy to use ones).

We will discuss the basic ones, but (as usual) just to explain the basic concepts that they use.

The basic operations can be divided in two main categories:

- managing the execution

- examine and modify the program state

- **l**(list) Lists the code at the current position
- **w**(here) What is the exact position in the call stack
- **n**(ext) Execute the next line (does not go down in new functions)
- **s**(tep) Execute the next statement (goes down in new functions)
- **r**(eturn) Return out of a subroutine
- **bt** Print the call stack
- **p**(rint) print the result of the expression
- **a** Print the local variables
- !command Execute the given Python command (instead of pdb commands)

- **u**(p) Walk up the call stack
- **d**(own) Walk down the call stack
- **h**(elp) Show a list of commands, or find help on a specific command
- **q**(uit) Quit the debugger and the program
- **c**(ontinue) Quit the debugger, continue in the program
- `<Return>` Repeat the previous command

In [1]:
```
%%file test.py

dati = [1, 2, 3, 4]

def my_function():
    for dato in dati:
        print(dati[dato])

my_function()
```

```
Overwriting test.py
```

In [2]:
```
%run test.py
```

```
2
3
4
```

```
---------------------------------------------------------------------------
-
IndexError                                Traceback (most recent call las
t)
~/didattica/corso_programmazione_1819/programmingCourseDIFA/test.py in <mo
dule>
      6         print(dati[dato])
      7
----> 8 my_function()

~/didattica/corso_programmazione_1819/programmingCourseDIFA/test.py in my_
function()
      4 def my_function():
      5     for dato in dati:
----> 6         print(dati[dato])
      7
      8 my_function()

IndexError: list index out of range
```

In [5]:
```
%run -d test.py
```

```
*** Blank or comment
*** Blank or comment
NOTE: Enter 'c' at the ipdb>  prompt to continue execution.
> /home/enrico/didattica/corso_programmazione_1819/programmingCourseDIFA/t
est.py(2)<module>()
      1
----> 2 dati = [1, 2, 3, 4]
      3
      4 def my_function():
      5     for dato in dati:

ipdb> n
> /home/enrico/didattica/corso_programmazione_1819/programmingCourseDIFA/t
est.py(4)<module>()
      2 dati = [1, 2, 3, 4]
      3
----> 4 def my_function():
      5     for dato in dati:
      6         print(dati[dato])

ipdb> p dati
[1, 2, 3, 4]
ipdb> !dati[-1] = 3
ipdb> p dati
[1, 2, 3, 3]
ipdb> c
2
3
3
3
```

breakpoints

Sometimes we know where the error is likely t be, and would like the program to proceed with the regular execution until some given point. These points are called **breakpoints**, and allow for a smoother debugging experience

Since Python 3.7 there's a built-in `breakpoint()` function that calls the configured debugger.

Just place `breakpoint()` anywhere in the code to get into the debugger shell.

Additionally, if you want to run a python script and ignore all `breakpoint()` calls in the code it's possible to do so by setting the environment variable PYTHONBREAKPOINT=0

In [1]:
```
%%file test.py

dati = [1, 2, 3, 4]

def my_function():
    breakpoint()
    for dato in dati:
        print(dati[dato])

my_function()
```

```
Writing test.py
```

In [3]:
```
%run test.py
```

```
> c:\users\enrico\documents\didattica\programmingcoursedifa\test.py(6)my_f
unction()
-> for dato in dati:
(Pdb) l
  1
  2     dati = [1, 2, 3, 4]
  3
  4     def my_function():
  5         breakpoint()
  6  ->     for dato in dati:
  7             print(dati[dato])
  8
  9     my_function()
[EOF]
(Pdb) a dati
(Pdb) p dati
[1, 2, 3, 4]
(Pdb) q
```

```
---------------------------------------------------------------------------
BdbQuit                                   Traceback (most recent call last)
~\Documents\didattica\programmingCourseDIFA\test.py in <module>
      7         print(dati[dato])
      8 
----> 9 my_function()

~\Documents\didattica\programmingCourseDIFA\test.py in my_function()
      4 def my_function():
      5     breakpoint()
----> 6     for dato in dati:
      7         print(dati[dato])
      8 

~\Documents\didattica\programmingCourseDIFA\test.py in my_function()
      4 def my_function():
      5     breakpoint()
----> 6     for dato in dati:
      7         print(dati[dato])
      8 

~\Miniconda3\lib\bdb.py in trace_dispatch(self, frame, event, arg)
     86             return # None
     87         if event == 'line':
---> 88             return self.dispatch_line(frame)
     89         if event == 'call':
     90             return self.dispatch_call(frame, arg)

~\Miniconda3\lib\bdb.py in dispatch_line(self, frame)
    111         if self.stop_here(frame) or self.break_here(frame):
    112             self.user_line(frame)
--> 113             if self.quitting: raise BdbQuit
    114         return self.trace_dispatch
    115 

BdbQuit: 
```

Assertions

they are useful to express your expectations about the code

In [2]:
```python
assert 1==0, "I was not expecting that"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-2-54e203d14fc0> in <module>()
----> 1 assert 1==0, "I was not expecting that"

AssertionError: I was not expecting that
```

but don't rely on it, as they can be removed from the execution by using the flag `-O` (Optimize) when calling the python interpreter.

A common mistake is to use assertions to check the inputs of a function. this is an improper use of the assert, both due to the fact that it can't be relied and that they are not providing discrimintive and informative errors to the user.

they should be used for testing for sure, but in normal code they have only one real use: express invariant of your code (think back about property testing).

They express in code that some characteristics of the code **has to** be valid, or something has gone very very wrong.

```
def my_smart_sort(sequence):
    # some sorting code
    sorted_sequence = someting(partial_result)
    # the resulting sequence has the same lenght of the original
one or someting is wrong
    assert len(sorted_sequence) == len(sequence)
    return sorted_sequence
```

you can look at asserts as a stronger form of comments: comments can potentially go out of snc with your code, while assert can't, so they can communicate to the programmer reading the function with a higher degree of confidence.

asserts are a little fidgety, and there a couple of ways of making their use more "humane"

the main problem with asserts is that unless one does a good job at using the return string, they will be completely obscure as a reason for an error

In [4]:
```
observed = 3
expected = 4
assert observed == expected
```

```
---------------------------------------------------------------------------
-
AssertionError                            Traceback (most recent call las
t)
Input In [4], in <cell line: 3>()
      1 observed = 3
      2 expected = 4
----> 3 assert observed == expected

AssertionError: 
```

first of all, one can write a longer error message by wrapping the string between parenthesis.

IMPORTANT: do not wrap the whole expression in parenthesis, or it will always pass!

WRONG, always passes! (but in modern python will raise a warning)

```
assert (observed == expected, "they are different")
```

CORRECT (but **do not** separate the strings with commas):

```python
assert observed == expected, (
    "they are different"
    "and I can write a string as long as I please"
)
```

Second, one can use modern string formatting (`f-strings`) to obtain informative formatting for low cost.

here we use two special capabilities of the f-strings:

- `{x=}` will print the string `"x=<value of the variable x>"`
- `{x!r}` will print the representation of the value using the `repr` function, that allows for example to distinguish between numbers and string when printed

In [7]:
```python
observed = 3
expected = '3'
assert observed == expected, (
    f"{observed=!r} different from {expected=!r}"
)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call las
t)
Input In [7], in <cell line: 3>()
      1 observed = 3
      2 expected = '3'
----> 3 assert observed == expected, (
      4     f"{observed=!r} different from {expected=!r}"
      5 )

AssertionError: observed=3 different from expected='3'
```

Third and last, similarly to testing, consider using helper functions to express more clearly what the assert is checking for.

It will be shorter to write and easier to read (and will force you to think a bit more about what you are doing

this is opaque to understand

```python
mylist = [1, 2, 3]
assert all(type(i)==type(mylist[0]) for i in mylist)
```

compare to this:

```python
def all_elements_are_same_type(iterable):
    return all(type(i)==type(iterable[0]) for i in iterable)

mylist = [1, 2, 3]
assert all_elements_are_same_type(mylist)
```

this is better:

- easier to undestand
- less prone to error
- easier to reuse in your code

- easier to change the implementation

the only limitation of this approach is that one still have to write their one error string, and cannot trivially generate it while performing the test.

there is a solution, albeit exotic, and that require implementing objects.

I'll leave a simple implementation if you feel like taking a look at it

I'm not sure if it would be worth for anything less than a full fledged program that also requires not using any other external library.

it does have the advantage that, same as the code for general expressions, perform basically no computation if the `assert` are removed...

In [11]:
```python
from collections import Counter

class All_elements_are_same_type:
    def __init__(self, *args, **kwargs):
        self.args = args
        self.kwargs = kwargs

    def test(self):
        iterable = self.args[0]
        result = all(type(i)==type(iterable[0]) for i in iterable)
        return result

    def error_message(self):
        iterable = self.args[0]
        types = Counter(type(i) for i in iterable)
        return f"the tested iterable {iterable!r} has different types: {t

mylist = [1, 2, '3']
test = All_elements_are_same_type(mylist)
assert test.test(), test.error_message()
```

```
---------------------------------------------------------------------------
-
AssertionError                            Traceback (most recent call las
t)
Input In [11], in <cell line: 20>()
     18 mylist = [1, 2, '3']
     19 test = All_elements_are_same_type(mylist)
---> 20 assert test.test(), test.error_message()

AssertionError: the tested iterable [1, 2, '3'] has different types: Count
er({<class 'int'>: 2, <class 'str'>: 1})
```

Logging

When you execute your code and print the internal state of the program to check that is working properly, that is a rudimentary form of logging.

Printing the state of your program works fine as long as your program is simple and the amount of state is small.

For anything more complicated, you need to use a logging system.

The basic idea of a logging system is to standardize how and what gets written on a file

In [54]:
```
import logging

logging.basicConfig(level=logging.WARNING)

logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
```

```
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
```

In [ ]:
```
logging.basicConfig(filename='app.log', filemode='w',
                    format='%(name)s - %(levelname)s - %(message)s')
```

if you need a simple timer with decent time formatting, you can use this one:

In [56]:
```
from contextlib import contextmanager
from datetime import datetime

@contextmanager
def Timer(description):
    start = datetime.now()
    try:
        yield
    finally:
        end = datetime.now()
        timedelta = end-start
        message = f"{description}, started: {start}, ended: {end}, ellaps
        logging.warning(message)

with Timer("test run"):
    import time
    time.sleep(2)
```

```
WARNING:root:test run, started: 2022-03-23 17:12:48.767673, ended: 2022-03
-23 17:12:50.769854, ellapsed: 0:00:02.002181
```

A better library is called **eliot**, that allows for a more structured logging instead of just printing to stderr

In [41]:
```
%%file test.py

from eliot import start_action, to_file, log_message, log_call, start_tas
to_file(open("test.log", "w"))

@log_call
def myfunction(value):
    result = 1/value
    log_message("during operation", vars=locals())
    return result
```

```
with start_task(action_type="processing numbers"):
    for number in [4, 1, 0, 2, 4]:
        with start_action(action_type="start evaluation", number=number):
            point = number *2
            total = sum(i for i in range(point))
            with start_action(action_type="inside evaluation", total=tota
                result = myfunction(total)
```

```
Overwriting test.py
```

In [42]:
```
!rm test.log
%run test.py
```

```
---------------------------------------------------------------------------
-
ZeroDivisionError                         Traceback (most recent call las
t)
File ~/didattica/programmingCourseDIFA_repo/master/test.py:17, in <module>
     15 total = sum(i for i in range(point))
     16 with start_action(action_type="inside evaluation", total=total):
---> 17     result = myfunction(total)

File <boltons.funcutils.FunctionBuilder-8>:2, in myfunction(value)

File /home/storage/miniconda3/lib/python3.9/site-packages/eliot/_action.p
y:943, in log_call.<locals>.logging_wrapper(*args, **kwargs)
    940     callargs = {k: callargs[k] for k in include_args}
    942 with start_action(action_type=action_type, **callargs) as ctx:
--> 943     result = wrapped_function(*args, **kwargs)
    944     if include_result:
    945         ctx.add_success_fields(result=result)

File ~/didattica/programmingCourseDIFA_repo/master/test.py:7, in myfunctio
n(value)
      5 @log_call
      6 def myfunction(value):
----> 7     result = 1/value
      8     log_message("during operation", vars=locals())
      9     return result

ZeroDivisionError: division by zero
```

In [43]:
```
!eliot-tree test.log
```

```
8414afd7-131d-448f-8db0-b0188c00aca0
└── processing numbers/1 ⇒ started 2022-03-23 15:54:47 ⧖ 0.015s
    ├── start evaluation/2/1 ⇒ started 2022-03-23 15:54:47 ⧖ 0.004s
    │   ├── number: 4
    │   ├── inside evaluation/2/2/1 ⇒ started 2022-03-23 15:54:47 ⧖ 0.003s
    │   │   ├── total: 28
    │   │   ├── __main__.myfunction/2/2/2/1 ⇒ started 2022-03-23 15:54:47
⧖ 0.001s
    │   │   │   ├── value: 28
    │   │   │   ├── during operation/2/2/2/2 2022-03-23 15:54:47
    │   │   │   │   └── vars:
    │   │   │   │       ├── result: 0.03571428571428571
    │   │   │   │       └── value: 28
    │   │   │   └── __main__.myfunction/2/2/2/3 ⇒ succeeded 2022-03-23 15:
54:47
    │   │   │       └── result: 0.03571428571428571
    │   │   └── inside evaluation/2/2/3 ⇒ succeeded 2022-03-23 15:54:47
    │   └── start evaluation/2/3 ⇒ succeeded 2022-03-23 15:54:47
    ├── start evaluation/3/1 ⇒ started 2022-03-23 15:54:47 ⧖ 0.004s
    │   ├── number: 1
    │   ├── inside evaluation/3/2/1 ⇒ started 2022-03-23 15:54:47 ⧖ 0.003s
    │   │   ├── total: 1
    │   │   ├── __main__.myfunction/3/2/2/1 ⇒ started 2022-03-23 15:54:47
⧖ 0.001s
    │   │   │   ├── value: 1
    │   │   │   ├── during operation/3/2/2/2 2022-03-23 15:54:47
    │   │   │   │   └── vars:
    │   │   │   │       ├── result: 1.0
    │   │   │   │       └── value: 1
    │   │   │   └── __main__.myfunction/3/2/2/3 ⇒ succeeded 2022-03-23 15:
54:47
    │   │   │       └── result: 1.0
    │   │   └── inside evaluation/3/2/3 ⇒ succeeded 2022-03-23 15:54:47
    │   └── start evaluation/3/3 ⇒ succeeded 2022-03-23 15:54:47
    ├── start evaluation/4/1 ⇒ started 2022-03-23 15:54:47 ⧖ 0.004s
    │   ├── number: 0
    │   ├── inside evaluation/4/2/1 ⇒ started 2022-03-23 15:54:47 ⧖ 0.002s
    │   │   ├── total: 0
    │   │   ├── __main__.myfunction/4/2/2/1 ⇒ started 2022-03-23 15:54:47
⧖ 0.001s
    │   │   │   ├── value: 0
    │   │   │   └── __main__.myfunction/4/2/2/2 ⇒ failed 2022-03-23 15:54:
47
    │   │   │       ├── exception: builtins.ZeroDivisionError
    │   │   │       └── reason: division by zero
    │   │   └── inside evaluation/4/2/3 ⇒ failed 2022-03-23 15:54:47
    │   │       ├── exception: builtins.ZeroDivisionError
    │   │       └── reason: division by zero
    │   └── start evaluation/4/3 ⇒ failed 2022-03-23 15:54:47
    │       ├── exception: builtins.ZeroDivisionError
    │       └── reason: division by zero
    └── processing numbers/5 ⇒ failed 2022-03-23 15:54:47
        ├── exception: builtins.ZeroDivisionError
        └── reason: division by zero
```

In [44]:
```
!head test.log
```

```
{"action_status": "started", "timestamp": 1648050887.3135529, "task_uuid":
"8414afd7-131d-448f-8db0-b0188c00aca0", "action_type": "processing number
s", "task_level": [1]}
{"number": 4, "action_status": "started", "timestamp": 1648050887.314634,
"task_uuid": "8414afd7-131d-448f-8db0-b0188c00aca0", "action_type": "start
evaluation", "task_level": [2, 1]}
{"total": 28, "action_status": "started", "timestamp": 1648050887.3152003,
"task_uuid": "8414afd7-131d-448f-8db0-b0188c00aca0", "action_type": "insid
e evaluation", "task_level": [2, 2, 1]}
{"value": 28, "action_status": "started", "timestamp": 1648050887.3162124,
"task_uuid": "8414afd7-131d-448f-8db0-b0188c00aca0", "action_type": "__mai
n__.myfunction", "task_level": [2, 2, 2, 1]}
{"vars": {"value": 28, "result": 0.03571428571428571}, "timestamp": 164805
0887.3167036, "task_uuid": "8414afd7-131d-448f-8db0-b0188c00aca0", "task_l
evel": [2, 2, 2, 2], "message_type": "during operation"}
{"result": 0.03571428571428571, "action_status": "succeeded", "timestamp":
1648050887.3173218, "task_uuid": "8414afd7-131d-448f-8db0-b0188c00aca0",
"action_type": "__main__.myfunction", "task_level": [2, 2, 2, 3]}
{"action_status": "succeeded", "timestamp": 1648050887.3181512, "task_uui
d": "8414afd7-131d-448f-8db0-b0188c00aca0", "action_type": "inside evaluat
ion", "task_level": [2, 2, 3]}
{"action_status": "succeeded", "timestamp": 1648050887.319012, "task_uui
d": "8414afd7-131d-448f-8db0-b0188c00aca0", "action_type": "start evaluati
on", "task_level": [2, 3]}
{"number": 1, "action_status": "started", "timestamp": 1648050887.319731,
"task_uuid": "8414afd7-131d-448f-8db0-b0188c00aca0", "action_type": "start
evaluation", "task_level": [3, 1]}
{"total": 1, "action_status": "started", "timestamp": 1648050887.3206432,
"task_uuid": "8414afd7-131d-448f-8db0-b0188c00aca0", "action_type": "insid
e evaluation", "task_level": [3, 2, 1]}
```

What if you want to see the results in real time?

A better solution is to periodically refresh the result of `eliot-tree`, piping the tail of the log file in it:

```
tail -f test.log | eliot-tree
```

for example, if we make our program be slower (simulating a slow computation)

The only limitations are:

- you have to interrupt the tail process manually
- actions are written only when completed, so if you have very high level actions this will not print until they are done

Warning systems

Warnings are a way to comunicate directly with the user and let them know that there is something fishy going on.

While logging is something that is run consistently, warnings should appear only in some specific situations.

A tipical case is to inform your user that one of the functions that is being used is going to be removed from the next version of your library and that they should use something different

In [85]:
```
import warnings
warnings.warn("this is an old script, use a new one!")
```

```
/home/enrico/miniconda3/lib/python3.6/site-packages/ipykernel_launcher.py:
2: UserWarning: this is an old script, use a new one!
```

if you want to raise an error when you catch a warning, you can use a specific context manager

In [86]:
```
with warnings.catch_warnings():
        warnings.simplefilter('error', category=Warning)
        warnings.warn("there is a problem!")
```

```
---------------------------------------------------------------------------
-
UserWarning                                 Traceback (most recent call las
t)
<ipython-input-86-15bf7a32a4ae> in <module>()
      1 with warnings.catch_warnings():
      2         warnings.simplefilter('error', category=Warning)
----> 3         warnings.warn("there is a problem!")

UserWarning: there is a problem!
```

on the opposite, you might want to silence certain warnings as you know what you're doing.

SPOILER: you probably should not use this code!

In [88]:
```
with warnings.catch_warnings():
        warnings.simplefilter('ignore', category=Warning)
        warnings.warn("there is a problem!")
```

Linters

Linters are programs that check your code for possible mistakes and errors before the execution.

there are several of them, with various level of informations that they can gather.

Most editors can be configured to run them in the background and show the resulting informations directly in the editor window.

examples of things that they will catch are:

- variable definited but not used
- variable used but not assigned
- overloading of existing functions

- syntax errors

and so on.

They are extremely useful when working with a dynamic language such as python, as they provide some functionalities of the traditional compilers

some of these linters are:

- pylint
- pycodestyle (previously called pep8)
- pyflakes
- flake8

Linters can perform a variety of different checks, depending on their goal.

For example the `perflint` linter claims to warn the user against bad code practices that will affect python performances.

https://github.com/tonybaloney/perflint

In [45]:
```
%%file test.py

def eleva(n):
    return n**2

dati = [1, 2, 3, 4]

for dato in dati:
    print(eleva(dati[dato]))

print(data)
```

```
Overwriting test.py
```

In [60]:
```
!batcat test.py --theme=GitHub -A
```

```
───────┬────────────────────────────────────────────────
       │ File: test.py
───────┼────────────────────────────────────────────────
   1   │ ␊
   2   │ def•eleva(n):␊
   3   │ ••••return•n**2␊
   4   │ ␊
   5   │ dati•=•[1,•2,•3,•4]␊
   6   │ ␊
   7   │ for•dato•in•dati:␊
   8   │ ••••print(eleva(dati[dato]))␊
   9   │ ••••␊
  10   │ print(data)␊
───────┴────────────────────────────────────────────────
```

In [47]:
```
!pylint test.py
```

```
************* Module test
test.py:9:0: C0303: Trailing whitespace (trailing-whitespace)
test.py:1:0: C0114: Missing module docstring (missing-module-docstring)
test.py:2:10: C0103: Argument name "n" doesn't conform to snake_case namin
g style (invalid-name)
test.py:2:0: C0116: Missing function or method docstring (missing-function
-docstring)
test.py:10:6: E0602: Undefined variable 'data' (undefined-variable)

--------------------------------------------------------------------
Your code has been rated at -5.00/10 (previous run: -5.00/10, +0.00)
```

the warning of pylint are formatted in a way that allow for most editors to go to the offending code:

`<filename>:<line number>:<character number>: <error code>: description`

where the lines are numbered starting from 1, anche the characters starting from 0

obviously not all suggestions are equally relevant, but they can spot several **code smells** before you run a long simulation