

Intro to programming 8

Henri Vandendriessche
henri.vandendriessche@ens.fr

2022-10-25

Where are we now

- Now that we write programs more and more complicated, we end up encountering more and more complex situation

Where are we now

- Now that we write programs more and more complicated, we end up encountering more and more complex situation
- And more and more complicated bugs. . .

Where are we now

- Now that we write programs more and more complicated, we end up encountering more more and more complex situation
- And more and more complicated bugs. . .
- Your computer will do only what you tell it to do; it won't read your mind and do what you intended it to do

Where are we now

- Now that we write programs more and more complicated, we end up encountering more and more complex situation
- And more and more complicated bugs. . .
- Your computer will do only what you tell it to do; it won't read your mind and do what you intended it to do
- Everyone create bugs and everyone has to correct them

Where are we now

- Now that we write programs more and more complicated, we end up encountering more more and more complex situation
- And more and more complicated bugs. . .
- Your computer will do only what you tell it to do; it won't read your mind and do what you intended it to do
- Everyone create bugs and everyone has to correct them
- Fortunately, python comes with tools to help you get over them

Today

- Debugging level 0
- Assertion
- Logging
- pdb module

Disclaimer

- This document is highly based on **Automate the Boring Stuff with Python** chapter 11...
- <https://automatetheboringstuff.com/2e/chapter11/>

The easiest debugging rule

- When your program do what you asked (what you wrote) but not what you wanted. . .

The easiest debugging rule

- When your program do what you asked (what you wrote) but not what you wanted. . .
- The simplest and easiest way to see if there is a problem in you program is to check your variables at every key points of your program:

The easiest debugging rule

- When your program do what you asked (what you wrote) but not what you wanted. . .
- The simplest and easiest way to see if there is a problem in you program is to check your variables at every key points of your program:
 - When you perform a operation on your variable

The easiest debugging rule

- When your program do what you asked (what you wrote) but not what you wanted. . .
- The simplest and easiest way to see if there is a problem in you program is to check your variables at every key points of your program:
 - When you perform a operation on your variable
 - At the end of loop

The easiest debugging rule

- When your program do what you asked (what you wrote) but not what you wanted. . .
- The simplest and easiest way to see if there is a problem in you program is to check your variables at every key points of your program:
 - When you perform a operation on your variable
 - At the end of loop
 - At the end of a function if there is a return statement

The easiest debugging rule

- When your program do what you asked (what you wrote) but not what you wanted. . .
- The simplest and easiest way to see if there is a problem in you program is to check your variables at every key points of your program:
 - When you perform a operation on your variable
 - At the end of loop
 - At the end of a function if there is a return statement
 - When you import data from a file

The easiest debugging rule

- When your program do what you asked (what you wrote) but not what you wanted. . .
- The simplest and easiest way to see if there is a problem in you program is to check your variables at every key points of your program:
 - When you perform a operation on your variable
 - At the end of loop
 - At the end of a function if there is a return statement
 - When you import data from a file
- What is the best way to check on your variables and their types ?

The easiest debugging rule

- When your program do what you asked (what you wrote) but not what you wanted. . .
- The simplest and easiest way to see if there is a problem in you program is to check your variables at every key points of your program:
 - When you perform a operation on your variable
 - At the end of loop
 - At the end of a function if there is a return statement
 - When you import data from a file
- What is the best way to check on your variables and their types ?
 - print it: **print(your_variable)**

The easiest debugging rule

- When your program do what you asked (what you wrote) but not what you wanted. . .
- The simplest and easiest way to see if there is a problem in you program is to check your variables at every key points of your program:
 - When you perform a operation on your variable
 - At the end of loop
 - At the end of a function if there is a return statement
 - When you import data from a file
- What is the best way to check on your variables and their types ?
 - print it: **`print(your_variable)`**
 - print the type of your variable: **`print(type(your_variable))`**

The easiest debugging rule

- When your program do what you asked (what you wrote) but not what you wanted. . .
- The simplest and easiest way to see if there is a problem in you program is to check your variables at every key points of your program:
 - When you perform a operation on your variable
 - At the end of loop
 - At the end of a function if there is a return statement
 - When you import data from a file
- What is the best way to check on your variables and their types ?
 - print it: **`print(your_variable)`**
 - print the type of your variable: **`print(type(your_variable))`**
- **`print()`** can be useful for simple check but please don't debug your all your script with `print()`

The easiest debugging rule

- When your program do what you asked (what you wrote) but not what you wanted. . .
- The simplest and easiest way to see if there is a problem in you program is to check your variables at every key points of your program:
 - When you perform a operation on your variable
 - At the end of loop
 - At the end of a function if there is a return statement
 - When you import data from a file
- What is the best way to check on your variables and their types ?
 - print it: **`print(your_variable)`**
 - print the type of your variable: **`print(type(your_variable))`**
- **`print()`** can be useful for simple check but please don't debug your all your script with `print()`
- It's the level 0 of debugging

Try and except statements 1/5

- If you have an error in your script, the execution is stopped.

Try and except statements 1/5

- If you have an error in your script, the execution is stopped.
- Example: What's wrong in the following script:

```
def isDivided(divisor):  
    return 42 / divisor  
  
print(isDivided(2))  
print(isDivided(12))  
print(isDivided(0))  
print(isDivided(3))
```

Try and except statements 2/5

- If you have an error in your script, the execution is stopped.
- Example: What's wrong in the following script:

```
def isDivided(divisor):  
    return 42 / divisor
```

```
print(isDivided(2))
```

```
## 21.0
```

```
print(isDivided(12))
```

```
## 3.5
```

```
print(isDivided(0))
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): ZeroDivisionError:  
    division by zero
```

Try and except statements 3/5

- But you can still have your way around this error:

- `try :`
- `except ... :`

```
def isDivided(divisor):  
    try:  
        return 42 / divisor  
    except ZeroDivisionError:  
        print("What have I done again...")  
  
print(isDivided(2))
```

```
## 21.0
```

```
print(isDivided(12))
```

```
## 3.5
```

```
print(isDivided(0))
```

```
## What have I done again...  
## None
```

```
print(isDivided(3))
```

Try and except statements 4/5

- You can as well include the call of your function in the try

```
def isDivided(divisor):  
    return 42 / divisor  
  
try:  
    print(isDivided(2))  
    print(isDivided(12))  
    print(isDivided(0))  
    print(isDivided(3))  
  
except ZeroDivisionError:  
    print("What have I done again...")  
  
## 21.0  
## 3.5  
## What have I done again...
```


Try and except statements 4/5

- You can as well include the call of your function in the try

```
def isDivided(divisor):  
    return 42 / divisor  
  
try:  
    print(isDivided(2))  
    print(isDivided(12))  
    print(isDivided(0))  
    print(isDivided(3))  
  
except ZeroDivisionError:  
    print("What have I done again...")  
  
## 21.0  
## 3.5  
## What have I done again...
```

- Note that **print(isDivided(3))** is not executed. Once the execution jumps in the except statement, it does not go back to the try clause. Instead, it just continues moving down the program as normal

Try and except statements 5/5

- **try except** is useful:
 - For making some checks on your program flow
 - To get a (hopefully) clearer (or more adapted) error message than what python can provide
- **try except** is not useful:
 - To dodge error without solving them
 - To get a running program without crash

Raising Exceptions 1/2

- Python raises exceptions every time it attempts to execute an invalid code
- Exceptions are raised this way:
 - **raise** keyword
 - **Exception()** function
 - A useful sentence that will help you understand the problem in the Exception function

```
raise Exception('Ah Shit, Here We Go Again: another day another bug')
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): Exception: Ah Shit, Here We  
Go Again: another day another bug
```

- The **try** and **except** statement allows us to handle those exceptions if we anticipate them
- Without the **try** and **except** statement the program stops at the first exception raised

Raising Exceptions 2/2

- When / how to raise exception ?
- Exception can be passed as argument or returned at the end of a function

```
def doBullshit():  
    raise Exception('I did bullshit')  
  
try:  
    doBullshit()  
except Exception as err:  
    print("Ooops, ", str(err) )
```

```
## Ooops, I did bullshit
```

Getting the Traceback as a String 1/2

- Getting the information of your error.
- When your program crashes you always have an error with some information like:
 - The line of the error / the different lines if your program uses several files
 - The error message
 - The function / the sequence of functions involved (i.e., the call stack)
- All of that is called the **traceback**
- Example:

```
def callErrorTest():  
    errorTest()
```

```
def errorTest():  
    raise Exception('FATAL ERROR')
```

```
callErrorTest()
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): Exception: FATAL ERROR
```

- Those informations are here to help you locate and understand your error.

Getting the Traceback as a String 2/2

- Instead of just prompting it on your terminal, you can have access to your traceback using: `traceback.format_exc()`
- That way you can obtain your traceback information as a string
- You'll need the `traceback` module to access the function.
- It can be useful if you want to keep track of an error and write the info in a file. That way you keep it for later when you'll be mentally prepared to debug your code.

```
import traceback

try:
    raise Exception('FATAL ERROR')
except:
    errorFile = open('errorInfo.txt', 'w')
    errorFile.write(traceback.format_exc())
    errorFile.close()
    print('Don\'t have time now to debug but all info are in errorInfo.txt')
```

```
## 97
## Don't have time now to debug but all info are in errorInfo.txt
```

Assertions 1/3

- Assertion is a very a sanity check to be sure that the data has the expected format
- If the sanity check fails then an **AssertionError** exception is raised
- Assertions are raised this way:
 - **assert** keyword
 - A condition
 - A comma
 - A string to display when the check fails
- Example 1:

```
olympicGamesYears =[2021, 2012, 2008, 2024, 2016, 2000, 2004]
```

```
assert olympicGamesYears[0] < olympicGamesYears[-1] , "Years doesn't seem sorted"
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): AssertionError: Years doesn'  
t seem sorted
```

Assertions 2/3

- Assertion is a very a sanity check to be sure that the data has the expected format
- If the sanity check fails then an **AssertionError** exception is raised
- Assertions are raised this way:
 - **assert** keyword
 - A condition
 - A comma
 - A string to display when the check fails
- Example 2:

```
olympicGamesYears =[2004, 2012, 2008, 2024, 2016, 2000, 2021]
```

```
olympicGamesYears.sort()
```

```
assert olympicGamesYears[0] < olympicGamesYears[-1] , "Years doesn't seem sorted"
```


- Assertion are very useful:
 - At a key point to check that your data have the right format
 - If well design it gives you the location and the exact reason why if failed.
 - To save time in debugging
- But you need first to have a very concrete idea of what to expect and where in your program to set up some assertions

- Logging is to write down info or variable content from your script to keep track of the execution of your program
- **print()** is some form of logging.
- Why logging better than print :
 - You can have access to better information like timings for example
 - It can be systematic and organized
- Of course Python has a logging module and the name is **logging**:
<https://docs.python.org/3/library/logging.html>

- Example:

```
def factorial(n):  
    total = 1  
    for i in range(n + 1):  
        total *= i  
    return total  
  
print(factorial(5))
```

- Example:

```
def factorial(n):  
    total = 1  
    for i in range(n + 1):  
        total *= i  
    return total  
  
print(factorial(5))
```

0

- What will be the output ?

Logging 3/7

- Example with logging:

```
import logging
```

```
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')
logging.debug('Start of program')
```

```
## 2022-11-15 20:16:25,166 - DEBUG - Start of program
```

```
def factorial(n):
    logging.debug('Start of factorial(%s%%)' % (n))
    total = 1
    for i in range(n + 1):
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
    logging.debug('End of factorial(%s%%)' % (n))
    return total
```

```
logging.debug('Call of the function')
```

```
## 2022-11-15 20:16:25,177 - DEBUG - Call of the function
```

```
print(factorial(5))
```

```
## 0
```

Logging 4/7

- Example with logging:

```
import logging
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')
logging.debug('Start of program')
```

```
## 2022-11-15 20:16:25,389 - DEBUG - Start of program
```

```
def factorial(n):
    logging.debug('Start of factorial(%s%%)' % (n))
    total = 1
    for i in range(1,n + 1):
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
    logging.debug('End of factorial(%s%%)' % (n))
    return total
```

```
logging.debug('Call of the function')
```

```
## 2022-11-15 20:16:25,400 - DEBUG - Call of the function
```

```
print(factorial(5))
```

```
## 120
```

```
##
```

- The function `logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')` can be set to look at several levels:
 - **DEBUG** `logging.debug()` The lowest level. Used for small details. Usually you care about these messages only when diagnosing problems.
 - **INFO** `logging.info()` Used to record information on general events in your program or confirm that things are working at their point in the program.
 - **WARNING** `logging.warning()` Used to indicate a potential problem that doesn't prevent the program from working but might do so in the future.
 - **ERROR** `logging.error()` Used to record an error that caused the program to fail to do something.
 - **CRITICAL** `logging.critical()` The highest level. Used to indicate a fatal error that has caused or is about to cause the program to stop running entirely.

Logging 6/7

- Logging can be very useful to inspect systematically your program execution

```
import logging
logging.disable(logging.CRITICAL)
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %s')
logging.debug('Start of program')

def factorial(n):
    logging.debug('Start of factorial(%s%%)' % (n))
    total = 1
    for i in range(1, n + 1):
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
    logging.debug('End of factorial(%s%%)' % (n))
    return total

logging.debug('Call of the function')
print(factorial(5))
```


Logging 6/7

- Logging can be very useful to inspect systematically your program execution
- It needs some time to adapt to it but it's very practical and customizable at need

```
import logging
logging.disable(logging.CRITICAL)
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %s')
logging.debug('Start of program')

def factorial(n):
    logging.debug('Start of factorial(%s%%)' % (n))
    total = 1
    for i in range(1, n + 1):
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
    logging.debug('End of factorial(%s%%)' % (n))
    return total

logging.debug('Call of the function')
print(factorial(5))
```

Logging 6/7

- Logging can be very useful to inspect systematically your program execution
- It needs some time to adapt to it but it's very practical and customizable at need
- Why is it better than print ?

```
import logging
logging.disable(logging.CRITICAL)
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %')
logging.debug('Start of program')

def factorial(n):
    logging.debug('Start of factorial(%s%%)' % (n))
    total = 1
    for i in range(1,n + 1):
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
    logging.debug('End of factorial(%s%%)' % (n))
    return total

logging.debug('Call of the function')
print(factorial(5))
```

Logging 6/7

- Logging can be very useful to inspect systematically your program execution
- It needs some time to adapt to it but it's very practical and customizable at need
- Why is it better than print ?
 - print function can be non debugging feature so make the difference between debugging print and necessary print becomes hell

```
import logging
logging.disable(logging.CRITICAL)
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %
logging.debug('Start of program')

def factorial(n):
    logging.debug('Start of factorial(%s%%)' % (n))
    total = 1
    for i in range(1,n + 1):
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
    logging.debug('End of factorial(%s%%)' % (n))
    return total

logging.debug('Call of the function')
print(factorial(5))
```

Logging 6/7

- Logging can be very useful to inspect systematically your program execution
- It needs some time to adapt to it but it's very practical and customizable at need
- Why is it better than print ?
 - print function can be non debugging feature so make the difference between debugging print and necessary print becomes hell
 - can be switch off easily with `logging.disable(logging.CRITICAL)`

```
import logging
logging.disable(logging.CRITICAL)
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %
logging.debug('Start of program')

def factorial(n):
    logging.debug('Start of factorial(%s%%)' % (n))
    total = 1
    for i in range(1,n + 1):
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
    logging.debug('End of factorial(%s%%)' % (n))
    return total

logging.debug('Call of the function')
print(factorial(5))
```

- Logging is very useful for debugging without interrupting the execution
- For example when your program doesn't crash but doesn't exactly what you want
- Debugging even post mortem (after execution)
- It is especially efficient when you can not have access to the terminal (i.e when you program experiment with visual stimuli)

Python debugger: PDB 1/4

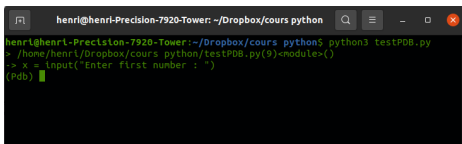
- The ultimate debugging tool is the **pdb** module: <https://docs.python.org/3/library/pdb.html>
- You just need to **import pdb** and call **pdb.set_trace()**

```
import pdb

def addition(a, b):
    answer = a * b
    return answer

pdb.set_trace()
x = input("Enter first number : ")
y = input("Enter second number : ")
sum = addition(x, y)
print(sum)
```

- When executing your program with the module **pdb** you have a screen like that



```
henri@henri-Precision-7920-Tower: ~/Dropbox/cours python
henri@henri-Precision-7920-Tower:~/Dropbox/cours python$ python3 testPDB.py
> /home/henri/Dropbox/cours python/testPDB.py(9)<module>()
-> x = input("Enter first number : ")
(Pdb)
```

Python debugger: PDB 2/4

- It is a command line tool that go sequentially at every step of the program
- There are a certain number of command to know:
 - **help** To display all commands
 - **where** Display the stack trace and line number of the current line
 - **next** Execute the current line and move to the next line ignoring function calls
 - **step** Step into functions called at the current line
 - **whatis** Check the type of variable

A terminal window with a dark background. The title bar shows the user 'henri' and the path '~/Dropbox/cours python'. The terminal text shows the execution of 'python3 testPDB.py', which prompts for an input. The user has entered a character, and the prompt has changed to '(Pdb) step' followed by 'Enter first number :', with a cursor at the end.

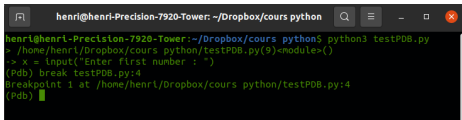
```
henri@henri-Precision-7920-Tower: ~/Dropbox/cours python
henri@henri-Precision-7920-Tower:~/Dropbox/cours python$ python3 testPDB.py
> /home/henri/Dropbox/cours python/testPDB.py(9)<module>()
-> x = input("Enter first number : ")
(Pdb) step
Enter first number : █
```

Python debugger: PDB 3/4

- You can use as well:
 - **args** To get all arguments of a function
 - **p** To get the value at a time *t* of a variable
- You can navigate in pdb prompt using:
 - **c** continue execution
 - **q** quit the debugger/execution
 - **n** step to next line within the same function
 - **s** step to next line in this function or a called function
 - **u** (up)
 - **d** (down)

Python debugger: PDB 4/4

- You can also set a breakpoint at a specific point in the script
- To do that you need to write on the terminal: **break filename: lineno, condition**

A terminal window titled 'henri@henri-Precision-7920-Tower: ~/Dropbox/cours python' showing the execution of a Python script with a PDB breakpoint. The user runs 'python3 testPDB.py', which prompts for the first number. The user enters '5', and the program hits a breakpoint at line 4 of testPDB.py, entering the PDB shell.

```
henri@henri-Precision-7920-Tower: ~/Dropbox/cours python$ python3 testPDB.py
> /home/henri/Dropbox/cours python/testPDB.py(9)<module>()
-> x = input("Enter first number : ")
(Pdb) break testPDB.py:4
Breakpoint 1 at /home/henri/Dropbox/cours python/testPDB.py:4
(Pdb)
```

- You can then use **c** to run the program until your breakpoint

A terminal window titled 'henri@henri-Precision-7920-Tower: ~/Dropbox/cours python' showing the continuation of the PDB session. The user enters 'c' to continue execution. The program prompts for the first and second numbers, which are '5' and '4' respectively. The user then runs the 'addition()' function, which calculates '5 * 4' and returns the result '20'. The program then returns to the PDB shell.

```
henri@henri-Precision-7920-Tower: ~/Dropbox/cours python$ python3 testPDB.py
> /home/henri/Dropbox/cours python/testPDB.py(9)<module>()
-> x = input("Enter first number : ")
(Pdb) break testPDB.py:4
Breakpoint 1 at /home/henri/Dropbox/cours python/testPDB.py:4
(Pdb) c
Enter first number : 5
Enter second number : 4
> /home/henri/Dropbox/cours python/testPDB.py(4)addition()
-> answer = a * b
(Pdb)
```