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 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. Even professional programmers create bugs all the time, so don't feel discouraged if your program has a problem.

Fortunately, there are a few tools and techniques to identify what exactly your code is doing and where it's going wrong. First, you will look at logging and assertions, two features that can help you detect bugs early. In general, the earlier you catch bugs, the easier they will be to fix.

Debugging

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 stropped.
- 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 goes 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

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 differents 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 you 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 tracback 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

An assertion is a sanity check to make sure your code isn't doing something obviously wrong. These sanity checks are performed by assert statements. If the sanity check fails, then an AssertionError exception is raised. In code, an assert statement consists of the following:

Logging

- Why logging better than print

Python debugger: PDB