Intro to programming 8

Henri Vandendriessche
henri.vandendriessche@ens.fr

2023-11-21

Terminal cheat sheet reminder

- Bash commands to navigate directories
  - Print Working Directory. Print the path of the current directory

```
pwd
```

  - List all files of the current directory

```
ls folder
```

  - Moving into folder1 and subfolder2 at once.

```
cd folder1/subfolder2
```

  - Moving out of a directory

```
cd ..
```

  - Going back and forth in the directory tree

```
cd ../../folder1/subfolder1
```

  - Going back to the root directory

```
cd ~
```

- "**Tab**" to use the auto-completion
- **Ctrl + C** to stop a program execution
- "**Upper arrow**" to see last commands
- Many more bash commands to use...

Where are we now

- As programs become increasingly complex, we find ourselves dealing with more intricate situations.

Where are we now

- As programs become increasingly complex, we find ourselves dealing with more intricate situations.
- Consequently, more complex bugs may arise.

Where are we now

- As programs become increasingly complex, we find ourselves dealing with more intricate situations.
- Consequently, more complex bugs may arise.
- Remember, your computer only executes what you instruct it to do; it can't read your mind and perform your intended actions.

Where are we now

- As programs become increasingly complex, we find ourselves dealing with more intricate situations.
- Consequently, more complex bugs may arise.
- Remember, your computer only executes what you instruct it to do; it can't read your mind and perform your intended actions.
- Everyone creates bugs, and everyone has to correct them.

Where are we now

- As programs become increasingly complex, we find ourselves dealing with more intricate situations.
- Consequently, more complex bugs may arise.
- Remember, your computer only executes what you instruct it to do; it can't read your mind and perform your intended actions.
- Everyone creates bugs, and everyone has to correct them.
- Fortunately, Python comes with tools to assist you in overcoming these challenges.

Today

- Debugging level 0
- Assertion
- Logging
- pdb module

Disclaimer

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

The easiest debugging rule

- When your program does what you instructed (what you wrote) but not what you intended. . .

The easiest debugging rule

- When your program does what you instructed (what you wrote) but not what you intended. . .
- The simplest and most straightforward way to identify issues in your program is to check your variables at key points:

The easiest debugging rule

- When your program does what you instructed (what you wrote) but not what you intended. . .
- The simplest and most straightforward way to identify issues in your program is to check your variables at key points:
  - After performing an operation on your variable.

The easiest debugging rule

- When your program does what you instructed (what you wrote) but not what you intended. . .
- The simplest and most straightforward way to identify issues in your program is to check your variables at key points:
  - After performing an operation on your variable.
  - At the end of a loop.

The easiest debugging rule

- When your program does what you instructed (what you wrote) but not what you intended. . .
- The simplest and most straightforward way to identify issues in your program is to check your variables at key points:
  - After performing an operation on your variable.
  - At the end of a loop.
  - At the end of a function, especially if there is a return statement.

The easiest debugging rule

- When your program does what you instructed (what you wrote) but not what you intended. . .
- The simplest and most straightforward way to identify issues in your program is to check your variables at key points:
  - After performing an operation on your variable.
  - At the end of a loop.
  - At the end of a function, especially if there is a return statement.
  - When importing data from a file.

The easiest debugging rule

- When your program does what you instructed (what you wrote) but not what you intended. . .
- The simplest and most straightforward way to identify issues in your program is to check your variables at key points:
  - After performing an operation on your variable.
  - At the end of a loop.
  - At the end of a function, especially if there is a return statement.
  - When importing data from a file.
- How can you best check your variables and their types?

The easiest debugging rule

- When your program does what you instructed (what you wrote) but not what you intended. . .
- The simplest and most straightforward way to identify issues in your program is to check your variables at key points:
  - After performing an operation on your variable.
  - At the end of a loop.
  - At the end of a function, especially if there is a return statement.
  - When importing data from a file.
- How can you best check your variables and their types?
  - print the variable: **print(your_variable)**

The easiest debugging rule

- When your program does what you instructed (what you wrote) but not what you intended. . .
- The simplest and most straightforward way to identify issues in your program is to check your variables at key points:
  - After performing an operation on your variable.
  - At the end of a loop.
  - At the end of a function, especially if there is a return statement.
  - When importing data from a file.
- How can you best check your variables and their types?
  - print the variable: **print(your_variable)**
  - print the type of the variable: **print(type(your_variable))**

The easiest debugging rule

- When your program does what you instructed (what you wrote) but not what you intended. . .
- The simplest and most straightforward way to identify issues in your program is to check your variables at key points:
  - After performing an operation on your variable.
  - At the end of a loop.
  - At the end of a function, especially if there is a return statement.
  - When importing data from a file.
- How can you best check your variables and their types?
  - print the variable: **print(your_variable)**
  - print the type of the variable: **print(type(your_variable))**
- While **print()** can be useful for simple checks, it's advisable not to debug your entire script with print().

The easiest debugging rule

- When your program does what you instructed (what you wrote) but not what you intended. . .
- The simplest and most straightforward way to identify issues in your program is to check your variables at key points:
  - After performing an operation on your variable.
  - At the end of a loop.
  - At the end of a function, especially if there is a return statement.
  - When importing data from a file.
- How can you best check your variables and their types?
  - print the variable: **print(your_variable)**
  - print the type of the variable: **print(type(your_variable))**
- While **print()** can be useful for simple checks, it's advisable not to debug your entire script with print().
- Consider it as the initial level (level 0) of debugging.

Errors and exceptions

- Until now we haven't spent much time looking at the error messages.
- There are (at least) two distinguishable kinds of errors: syntax errors and exceptions.
- Example of syntax errors:

```
test = True
if test == False
  print("Failed")
```

```
## invalid syntax (<string>, line 2)
```

Exceptions

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.
- Errors detected during execution are called logical errors or exceptions. They are not unconditionally fatal but python raises an exception whenever it tries to run an invalid code.

```
number = 1000

divided_by_0 = number/0
```

```
## division by zero
```

Handling exceptions

- You have plenty of categories of exceptions:
  - IndexError: when the index doesn't correspond to any value in a list for example.
  - AttributeError: When attribute assignment is failed.
  - IndentationError: When the indentation of the code is wrong.
  - ImportError: When the imported module is not found
  - NameError : undeclared variable
  - TypeError: When operation are applied to a wrong type of varialble.
  - ...
- Exceptions in itself doesn't do much. It's a type of data that manipulate python errors.
- To raise or handle exceptions there are several options:
  - The **raise** statement.
  - The **assert** statement.
  - The **try** and **except** statement.

Raising Exceptions 1/2

- Python raises exceptions every time it attempts to execute invalid code.
- Python does it automatically with a certain precision, but you can create you own exception
- Exceptions are raised using the following:
  - The **raise** keyword
  - The **Exception()** function
  - A descriptive sentence that helps you understand the problem in the Exception function.

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

```
## Ah Shit, Here We Go Again: another day another bug
```

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')

doBullshit()
```

```
## I did bullshit
```

Assertions 1/3

- Assertion is a sanity check to ensure that the data has the expected format.
- If the sanity check fails, then an **AssertionError** exception is raised.
- Assertions are raised in the following way:
  - Using the **assert** keyword
  - Including a condition.
  - Separating the condition with a comma.
  - Providing 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"
```

```
## Years doesn't seem sorted
```

Assertions 2/3

- Assertion is a sanity check to ensure that the data has the expected format.
- If the sanity check fails, then an **AssertionError** exception is raised.
- Assertions are raised in the following way:
  - Using the **assert** keyword
  - Including a condition.
  - Separating the condition with a comma.
  - Providing 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"
```

Assertions 3/3

- Assertion are very useful:
  - At key points to check that your data has the right format.
  - If well designed, they provide you with the location and the exact reason why it failed.
  - To save time in debugging.
- However, you need to first have a very concrete idea of what to expect and where in your program to set up some assertions.

Try and except statements 1/5

- If you encounter an error in your script, the execution is halted.

Try and except statements 1/5

- If you encounter an error in your script, the execution is halted.
- 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 encounter an error in your script, the execution is halted.
- 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))
```

```
## 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 also include the call to your function in the try block.

```
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 also include the call to your function in the try block.

```
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 to 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:
  - To perform checks on your program flow.
  - To obtain a (hopefully) clearer or more adapted error message than what Python can provide.
- **try except** is not useful:
  - To avoid errors without resolving them.
  - To achieve a running program without a crash.

Getting the Traceback as a String 1/2

- Gathering information about your error.
- When your program crashes, you always receive an error with information such as:
  - 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 this is referred to as the **traceback**
- Example:

```
def callErrorTest():
  errorTest()

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

callErrorTest()
```

```
## FATAL ERROR
```

- That information is here to help you locate and understand you error.

Getting the Traceback as a String 2/2

- Instead of just displaying it on your terminal, you can access your traceback using: **traceback.format_exc()**
- This 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 information to a file. This allows you to 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
```

Logging 1/7

- Logging involves writing down information or variable content from your script to keep track of the execution of your program.
- **print()** serves as a form of logging.
- Why is logging better than print :
  - You can access better information, such as timings, for example.
  - It can be systematic and organized.
- Of course, Python has a logging module, and its name is **logging**: https://docs.python.org/3/library/logging.html

Logging 2/7

- Example:

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

print(factorial(5))
```

Logging 2/7

- 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 -  %(m
logging.debug('Start of program')
```

```
## 2023-12-05 13:52:22,766 -  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')
```

```
## 2023-12-05 13:52:22,773 -  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 -  %(m
logging.debug('Start of program')
```

```
## 2023-12-05 13:52:22,809 -  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')
```

```
## 2023-12-05 13:52:22,815 -  DEBUG -  Call of the function
```

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

```
## 120
##
```

Logging 5/7

- 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 for systematically inspecting your program execution.

```
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 for systematically inspecting your program execution.
- It takes some time to adapt to it, but it's highly practical and customizable as needed.

```
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 for systematically inspecting your program execution.
- It takes some time to adapt to it, but it's highly practical and customizable as needed.
- 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 for systematically inspecting your program execution.
- It takes some time to adapt to it, but it's highly practical and customizable as needed.
- Why is it better than print ?
  - The print function can serve non-debugging purposes, making it challenging to differentiate between debugging print and essential print statements.

```
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 for systematically inspecting your program execution.
- It takes some time to adapt to it, but it's highly practical and customizable as needed.
- Why is it better than print ?
  - The print function can serve non-debugging purposes, making it challenging to differentiate between debugging print and essential print statements.
  - Logging can be easily switched off 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 7/7

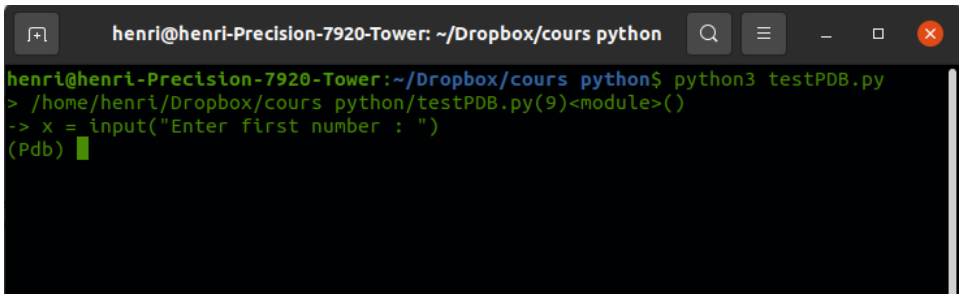
- Logging is highly useful for debugging without interrupting the execution.
- For example, when your program doesn't crash but doesn't produce the desired output.
- Debugging can even be done post-mortem (after execution).
- It is especially efficient when you cannot access the terminal (i.e., when your program involves 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()**

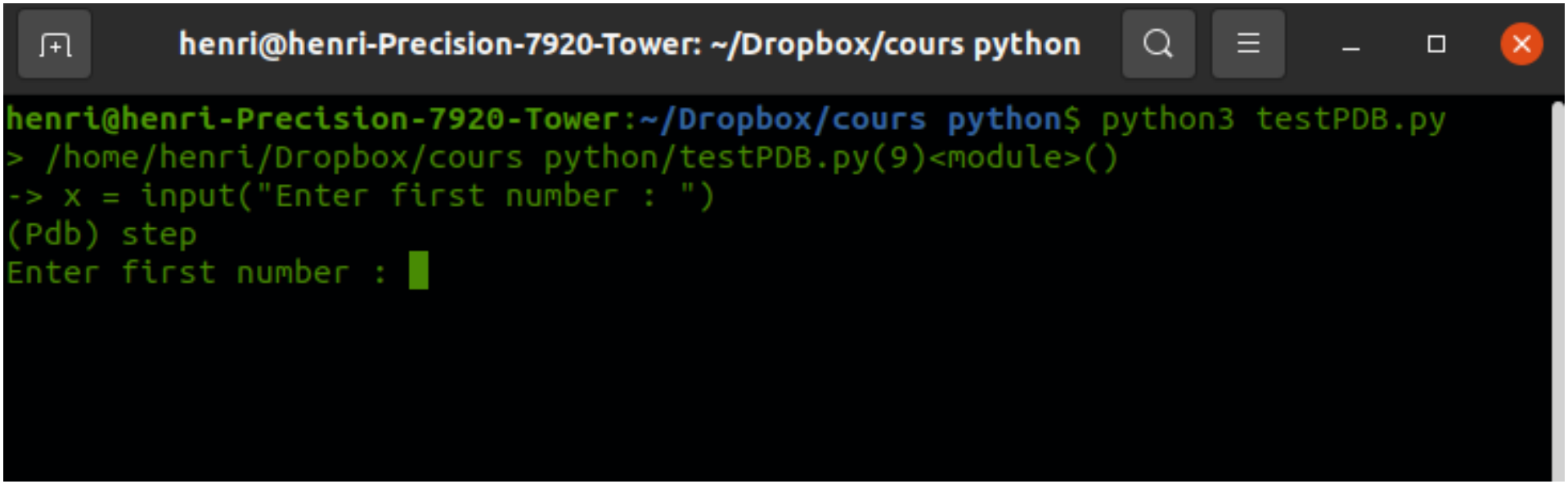
```
import pdb

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



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

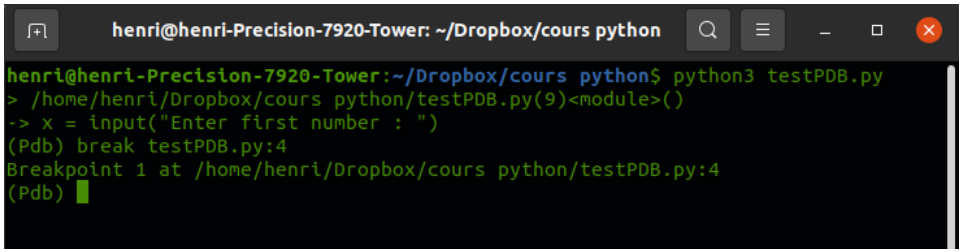


Python debugger: PDB 3/4

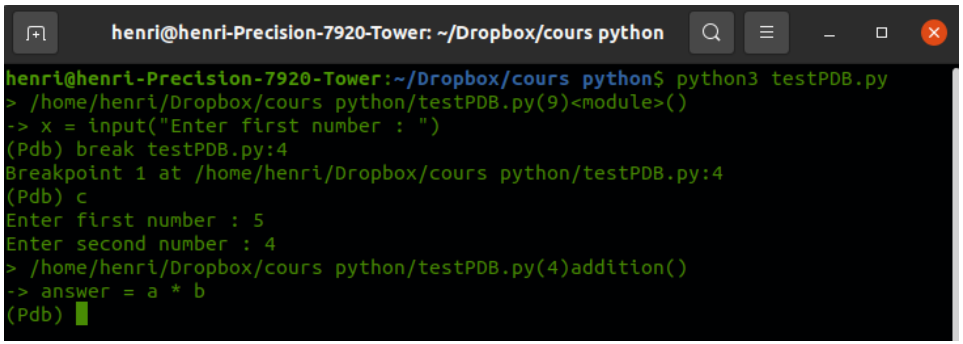
- You can use as well:
  - **args** To get all arguments of a function
  - **p** To get the value at a time t of a variable
- You can navigate in pdb prompt using:
  - **c** continue execution
  - **q** quit the debugger/execution
  - **n** step to next line within the same function
  - **s** step to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



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



Exercise on logging

- Use my exercise correction from last week on the cameras and:
  - 1 log every argument/parameter sent to a function
  - 2 log every variable in a return statement

Exercise on pdb 1/3

- Use my exercise correction from last week on the cameras and:
  - 1 execute line by line the program using pdb
  - 2 print argument(s) of all function and check their type
  - 3 set breakpoints right before every return statement of a function

Exercise on pdb 2/3

- Debug the following adding program:

```
print('Enter the first number to add:')
first = input()
print('Enter the second number to add:')
second = input()
print('Enter the third number to add:')
third = input()
print('The sum is ' + first + second + third)
```

Exercise on pdb 3/3

- Debug the following coin toss program:

```
import random
guess = ''
while guess not in ('heads', 'tails'):
    print('Guess the coin toss! Enter heads or tails:')
    guess = input()
toss = random.randint(0, 1) # 0 is tails, 1 is heads
if toss == guess:
    print('You got it!')
else:
    print('Nope! Guess again!')
    guesss = input()
    if toss == guess:
        print('You got it!')
    else:
        print('Nope. You are really bad at this game.')
```

Exercise 1

- Write a program that asks for the first name, last name, age and date of birth of a user. Then create one or several functions that perform input validation
- Check that the age is a valid number,
- Check that the first name and last name are only letters
- Check that the date of birth is in the valid format dd/mm/yyy and is coherent with the their age
- Otherwise, ask again the invalid data

Exercise 2

- Write a Python program to create a Caesar cipher https://en.wikipedia.org/wiki/Caesar_cipher.
- Include one function to encrypt and decrypt the text with a shift. Use a positive shift for encryption and a negative shift for decryption.
- The program should consider both uppercase and lowercase letters.
- Hint: check the function ord(), char() and the Unicode table.

Exercise 3

- Write a program that attempts to perform a Brute-force attack:
  - Using these collections of characters: first lower case letters “abcdefghijklmnopqrstuvwxyz,” then lower case and lower case letters and numbers “abcdefghijklmnopqrstuvwxyz1234567890,” and finally lower and upper case letters, numbers and special characters “abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890,;:!?./§%*+=“.
- Ask the user a password (1 - 5 character)
- Write a function that tries every possible combination of our set of characters with all possible lengths (from 1 to 5).
- Record and print the time to find the solution
- Write at every step the number length you are testing and time taken so far.
- To continue, calulate the number of combination according to the length of the password and the collection of character selected.
- Hint: check itertools module