Intro to programming 5

Henri Vandendriessche
henri.vandendriessche@ens.fr

2022-10-18

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
- Many more bash commands to use...

Previously on Intro to Programming (Python)

- Data types:
  - integer
  - float
  - string
  - boolean
- **If**, **For** and **While** loops:
  - syntax
  - indentation
- Data collections:
  - list
  - tuple
  - set
  - dictionary
- Python Standard library
  - Python modules
  - Python built-in functions
- Functions:
  - Parameters and arguments
  - Return values
  - Scope of variable

Today

- More on functions:
- Building abstraction with :
  - Recursive functions
  - High order functions
- Exercises

Clarification 1 1/2

- Breaking a loop is possible. For example when the rest of a loop is useless

```
# Checking if a number is primitive
N = 72239
for i in range(2, 300):
  if N % i == 0:
    print(i)
    break
```

```
## 29
```

- Other example

```
# Checking a password
passwd = 'sesame'

while True:
  code = input('Password? ')
  if code == passwd:
    break
  else:
    print('invalid password')

print("You are in!")
```

Clarification 1 2/2

- The keyword continue is also very useful to pass the current iteration

```
for i in range(0,10):
  if i == 5:
    continue
  print(i)
```

```
## 0
## 1
## 2
## 3
## 4
## 6
## 7
## 8
## 9
```

Clarification 2

- Many scripts will contain a series of functions and then the line

```
if __name__ == '__main__':
```

- It will act differently if the script is the main script of if it is imported by another script
- The condition is true only if the script is executed as a python script.
- The functions defined before the **if name == 'main':** can be reused with import script
- Why does it work like that? If you are creating a library or a module, you might want to have some configuration tests or settings. At the same time you want it to be used as a module by others so that they only need the functions.

More on functions 1/4

- Information can be passed into a function with what is called Parameters or Arguments

```
def my_function(parameter1, parameter2):
  print("Hello, my name is", parameter1, "and I'm", parameter2,"years old")

argument1 = "Bob"
argument2 = 30
my_function(argument1, argument2)
```

```
## Hello, my name is Bob and I'm 30 years old
```

More on functions 1/4

- Information can be passed into a function with what is called Parameters or Arguments

```
def my_function(parameter1, parameter2):
  print("Hello, my name is", parameter1, "and I'm", parameter2,"years old")

argument1 = "Bob"
argument2 = 30
my_function(argument1, argument2)
```

```
## Hello, my name is Bob and I'm 30 years old
```

- Parameter is in the definition of a function and Argument refer to the information passed in the call of the function

More on functions 2/4

- Arbitrary Argument or *args:

```
def my_function(*parameter):
  print("Hello, my name is", parameter[1])

argument1 = "Bob"
argument2 = "Linda"
argument3 = "Peter"
argument4 = "Nancy"
my_function(argument1, argument2, argument3, argument4)
```

```
## Hello, my name is Linda
```

```
my_function(argument2, argument3)
```

```
## Hello, my name is Peter
```

- If you don't know how many arguments you'll have, you can use a * before the parameter name in the definition of your function.

More on functions 3/4

- Keyword argument. Remember that you can do

```
def f(a, b):
  print('a=', a)
  print('b=', b)

f(1 ,2)
```

```
## a= 1
## b= 2
```

```
f(2, 1)
```

```
## a= 2
## b= 1
```

```
f(b=2, a=1)    # but one can also use the names of arguments
```

```
## a= 1
## b= 2
```

```
f(b=1, a=2)
```

```
## a= 2
## b= 1
```

More on functions 4/4

- If you can pass on to a function, an unknown number or arguments (with *args) in unordered manner (with Keyword arguments), you have Arbitrary Keyword Arguments, **kwargs

```
def my_function(**parameter):
  print("Hello, my name is", parameter["name1"], "and not", parameter["name3"])

argument1 = "Bob"
argument2 = "Linda"
argument3 = "Peter"
argument4 = "Nancy"
my_function(name1= argument1, name2= argument2, name3= argument3)
```

```
## Hello, my name is Bob and not Peter
```

```
my_function(name1= argument2, name2= argument3, name3= argument4)
```

```
## Hello, my name is Linda and not Nancy
```

Summary on function

- A function is a block of instructions that is given a name
- Functions must be defined before they are called (so usually all functions are defined before any code)
- Using functions avoids to duplicate code (i.e. copy and paste code)
- Using functions typically serves to make the code more readable (and maybe shorter)
- If you do not call the function, it will never be executed
- A function can:
  - have an input (one or several arguments of any kind of data types)
  - inputs can have default value
  - have an output (return one or several data) but not necessarily (procedure)
  - they can interact with outside variables or only depend on its arguments (pure function)
- Variables defined in a function stay only in the scope of the function
- One good practice is to place functions in an external modules such as the module in the standard library
- Functions can call functions
- Functions can call themselves...

Building abstraction

- In programming, we control the intellectual complexity of our programs by building abstractions that hide details when appropriate
  - That is exactly what you do when you use function like **len()**
  - That is exactly what you do when you build your own functions/modules
- This allows you to chunk compound operations as conceptual units, give them a name and manipulate them
- In this class, you will explore two other means to build a high level of abstractions with functions: recursive functions and higher-order functions.

Recursive function

- Remember what I said about recursive functions last time ?

Recursive function

- Remember what I said about recursive functions last time ?
- Recursion is a function calling itself

Recursive function

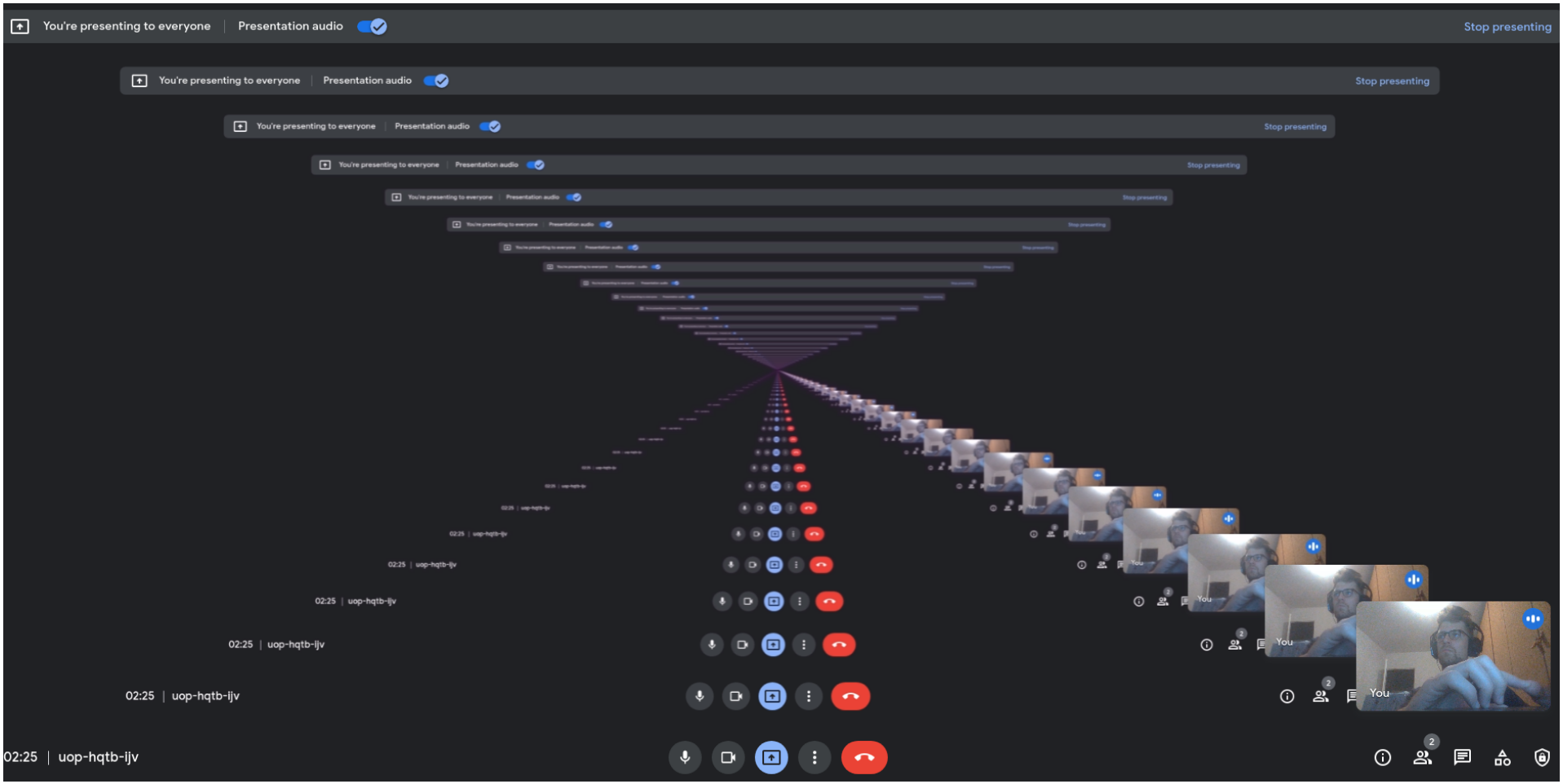
- Remember what I said about recursive functions last time ?
- Recursion is a function calling itself
- That has a termination condition

Recursive function

- Remember what I said about recursive functions last time ?
- Recursion is a function calling itself
- That has a termination condition
- And an increment statement

Recursive function example 1

- A concrete example of recursive implementation, yet bad



Recursive function example 2

- Recursion and lists: add every number in list with for loops

```
## With a loop for

def sum(list):
  sum1 = 0

  # Add every number in the list.
  for i in range(0, len(list)):
    sum1 = sum1 + list[i]

  # Return the sum.
  return sum1

print(sum([5,7,3,8,10]))
```

```
## 33
```

Recursive function example 3

- Recursion and lists: with recursion

```
def sum(list):
  if len(list)==1:
    return list[0]
  else:
    return list[0] + sum(list[1:])

print(sum([5,7,3,8,10]))
```

```
## 33
```

Recursive function example 4.1

- Factorial with for loops

```
def calcFactorial(number):
    factorial = 1

    for count in range (1, number):
        factorial = factorial*count

    factorial = factorial*number
    return factorial

print(calcFactorial(5))
```

```
## 120
```

Recursive function example 4.2

- Factorial with for recursion

```
def factorial(n):
  if n == 1:
    return 1
  else :
    return n * factorial(n-1)

print(factorial(5))
```

```
## 120
```

- But it has limitations. Everytime a function calls itself it allocates memory for that new call of the function. So you might easily run into an error. Python stops the function calls after a depth of 1000 calls. RecursionError: maximum recursion depth exceeded in comparison
- You don't have that problem with the for loop

Recursive function example 4.3

```
def calcFactorial(number):
    factorial = 1

    for count in range (1, number):
        factorial = factorial*count

    factorial = factorial*number
    return factorial

print(calcFactorial(3000))


def factorial(n):
  if n == 1:
    return 1
  else :
    return n * factorial(n-1)

print(factorial(1000))
```

Recursive function example 4.4

```
import sys

sys.setrecursionlimit(5000)

def factorial(n):
  if n == 1:
    return 1
  else :
    return n * factorial(n-1)

print(factorial(1000))
```

- But keep in mind that it has limitations and recursion is very suited for certain problems but not for all

Exercises on recursion

1 - Write a recursive function to reverse a list

2 - Write a recursive function to generate all permutations of a list of values

3 - Write a script that returns the pathnames of all the files contained inside a directory (at any depth of the hierarchy). You will need to use os.listdir() and os.path.isdir().

Higher-order functions : Intro

- A higher-order function is a function that does at least one of the following:
  - takes one or more functions as arguments (i.e. functions are assigned as variable)
  - returns a function as its result
- High order function had a new layer of complexity in term of abstraction

Reading advice:

https://wizardforcel.gitbooks.io/sicp-in-python/content/6.html

Application of high order function 1

- Example of Function as an arguments

```
# Python program to illustrate functions
# can be passed as arguments to other functions
def shout(text):
    return text.upper()

def whisper(text):
    return text.lower()

def greet(function):
    # storing the function in a variable
    greeting = function("Hi, I am created by a function \
    passed as an argument.")
    print(greeting)


greet(shout)
```

```
## HI, I AM CREATED BY A FUNCTION     PASSED AS AN ARGUMENT.
```

```
greet(whisper)
```

```
## hi, i am created by a function     passed as an argument.
```

Application of high order function 2 1/4

- Consider these examples:

```
def sum_naturals(n):
  total, k = 0, 1
  while k <= n:
    total, k = total + k, k + 1
  return total
sum_naturals(4)
```

```
## 10
```

```
def sum_cubes(n):
  total, k = 0, 1
  while k <= n:
    total, k = total + pow(k, 3), k + 1
  return total
sum_cubes(4)
```

```
## 100
```

```
def pi_sum(n):
  total, k = 0, 1
  while k <= n:
    total, k = total + 8 / (k * (k + 2)), k + 4
  return total
```

Application of high order function 2 2/4

- The pattern is

```
def <name>(n):
    total, k = 0, 1
    while k <= n:
        total, k = total + <term>(k), <next>(k)
    return total
```

- The presence of this kind of pattern is an evidence for a new level or abstraction that can be brought
- for example in putting and as arguments

Application of high order function 2 2/4

- The pattern is the following

```
def <name>(n):
    total, k = 0, 1
    while k <= n:
        total, k = total + <term>(k), <next>(k)
    return total
```

- The presence of this kind of pattern is an evidence for a new level or abstraction that can be brought
- In that case in putting and as arguments

```
#changing the name
def summation(n, term, next):
        total, k = 0, 1
        while k <= n:
            total, k = total + term(k), next(k)
        return total
```

Application of high order function 2 3/4

```
def summation(n, term, next):
        total, k = 0, 1
        while k <= n:
            total, k = total + term(k), next(k)
        return total

def cube(k):
  return pow(k,3)

def successor(k):
  return k + 1

def sum_cubes(n):
  return summation(n,cube,successor)

sum_cubes(3)
```

```
## 36
```

Application of high order function 2 4/4

```
def summation(n, term, next):
        total, k = 0, 1
        while k <= n:
            total, k = total + term(k), next(k)
        return total

def identity(k):
  return k

def successor(k):
  return k + 1

def sum_naturals(n):
  return summation(n, identity, successor)

sum_naturals(3)
```

```
## 6
```