intro to programming 4

Henri Vandendriessche

12/10/2021

So far

- Python, its life, its choice
- Data types: (integer / float / string / boolean)
- **If**, **For** and **While** loops:
- Data collections: (list, tuple, set, dictionary)
- Functions

Today

- Back on functions

- Building abstraction with :

+ Recursive functions

+ High order functions (part 1)

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 module, you will explore two other means to build abstractions with functions: recursive functions and higher-order functions.

Back on function

- A function is a block of instructions that is given a name
- Functions must be defined before they are called
- Using functions avoids to duplicate code (i.e. by cutting and pasting)
- 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)
  - have an output (return one or several data)
  - but not necessarily
- Variables defined in a function stay in the function
- One good practice is to place functions in an external modules such as in the standard library
- Functions can call functions
- Functions can call themselves...

Recursive function

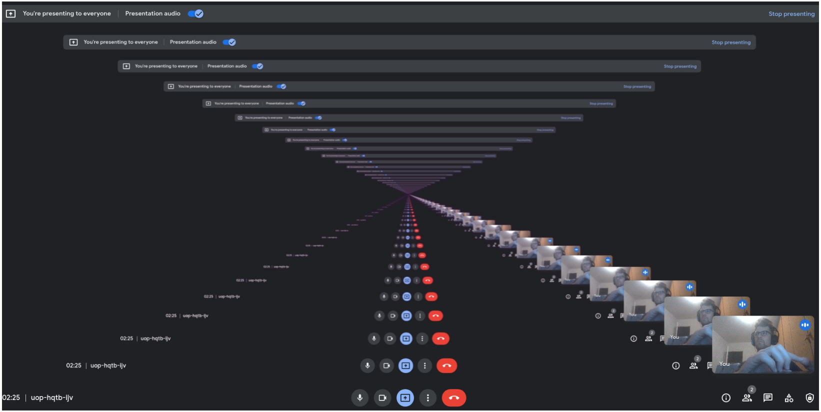
- Remember what I said about recursive functions last time ?

Recursive function

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

Recursive function example 1

- OK boomer



Recursive function example 2

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

```
## With a loop for

def sum(list):
  sum = 0

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

  # Return the sum.
  return sum

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

```
## 33
```

Recursive function example 2

- 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 3

- 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 3

- 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 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 3

```
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

Exercices on recursion

1 - Write a recursive function to reverse a list

2 - Write a script that displays the Koch snowflake (see on wikipedia) using a recursive function. You can use the turtle graphics module to draw on screen (as shown in section 16.5 of thinkcspy).

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

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

Exercice 1

1 - Write a recursive function to reverse a list

```python
def rev(l):
  if len(l) == 0:
    return []
  return [l[-1]] + rev(l[:-1])

print(rev(["a", "b", "c", "d"]))
```

```
## ['d', 'c', 'b', 'a']
```

Exercice 2

2 - Write a script that displays the Koch snowflake (see on wikipedia) using a recursive function. You can use the turtle graphics (https://docs.python.org/3.8/library/turtle.html) module to draw on screen (as shown in section 16.5 of thinkcspy).

https://en.wikipedia.org/wiki/Koch_snowflake

```
import turtle

def Koch(n, l):
    if n == 0:
        turtle.forward(l)
    else:
        Koch(n - 1, l / 3)
        turtle.left(60)
        Koch(n - 1, l / 3)
        turtle.right(120)
        Koch(n - 1, l / 3)
        turtle.left(60)
        Koch(n - 1, l / 3)

turtle.penup()
turtle.backward(600)
turtle.pendown()

Koch(1, 400)
Koch(2, 400)
Koch(3, 400)
```

Exercice 3

3 - Write a recursive function to generate all permutations of a string of character

```
def perms(s):
    if(len(s)==1):
      return [s]
    result=[]
    for i,v in enumerate(s):
        result += [v+p for p in perms(s[:i]+s[i+1:])]
    return result


perms('abc')
```

```
## ['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
```

Exercice 4

4 - 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().

```
import os

def listdirs(rootdir):
  for file in os.listdir(rootdir):
    print(file)
    d = os.path.join(rootdir, file)
    if os.path.isdir(d):
      print(d)
      listdirs(d)

rootdir = '/home/henri/Documents/Cours/PCBS/slides/2021-intro-to-programming/'
listdirs(rootdir)
```

```
## fourth class
## /home/henri/Documents/Cours/PCBS/slides/2021-intro-to-programming/fourth cla
## intro-to-programming-4.Rmd
## intro-to-programming-4.pdf
## cleancode.png
## intro-to-programming-4.html
## recursivity.png
## .Rhistory
## second class
## /home/henri/Documents/Cours/PCBS/slides/2021-intro-to-programming/second cla
```

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
pi_sum(100)
```

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)
```

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
```