Intro to programming 5

Henri Vandendriessche
henri.vandendriessche@ens.fr

2023-10-23

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...

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

- Further exploration of functions:
- Building abstraction with :
  - Recursive functions
  - High-order functions
- Exercises

More on functions 1/4

- Information can be passed into a function using either Parameters or Arguments.

```python
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 using either 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
```

- Parameters are defined in the function's definition, while Arguments refer to the information passed during the function call.

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
```

- To handle an arbitrary number of arguments, you can use an asterisk (*) before the parameter name in the function's definition.

More on functions 3/4

- Keyword arguments allow you to pass arguments to a function using specific names.

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

- This feature is called “Keyword arguments,” and it allows you to pass arguments in any order, regardless of their position in the function’s parameter list.

More on functions 4/4

- If you can pass an unknown number of arguments to a function using *args and accept them in an unordered manner with Keyword arguments, you have the capability to use **Arbitrary Keyword Arguments**, also known as **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 named block of instructions.
- Functions must be defined before they are called, so it's common to define all functions before any code.
- Functions prevent code duplication, making it more maintainable and concise.
- Using functions often enhances code readability.
- If you never call a function, it won't be executed.
- A function can:
  - have inputs, which can be one or several arguments of various data types.
  - inputs can have default values.
  - have an output, returning one or several data values, but not necessarily (some functions are procedures).
  - they can interact with outside variables or solely rely on their arguments (pure functions).
- Variables defined within a function are local to that function's scope.
- It's a good practice to place functions in external modules, such as modules in the standard library.
- Functions can call other functions.
- Functions can call themselves, which is known as recursion.

Building abstraction

- In programming, we manage the intellectual complexity of our programs by creating abstractions that conceal details when necessary.
    - this is precisely what you do when you use functions like **len()**
    - it's also what you do when you create your own functions and modules.
- This approach enables you to group compound operations into conceptual units, give them names, and manipulate them.
- In this class, you will delve into two additional methods for creating high-level abstractions with functions: recursive functions and higher-order functions.

Recursive function

- As a reminder from our previous discussion:

Recursive function

- As a reminder from our previous discussion:
  - recursion is a function calling itself.

Recursive function

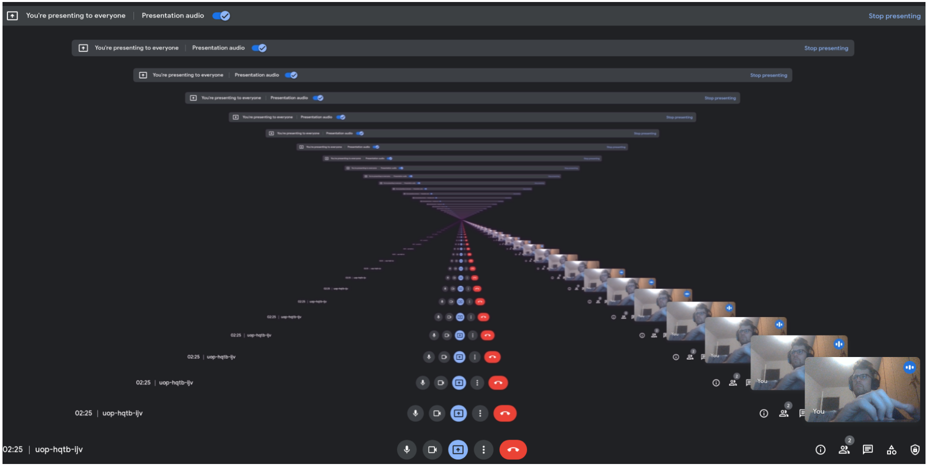
- As a reminder from our previous discussion:
    - recursion is a function calling itself.
    - it requires a termination condition.

Recursive function

- As a reminder from our previous discussion:
    - recursion is a function calling itself.
    - it requires a termination condition.
    - it also involves an increment statement.

Recursive function example 1

- Let's consider a concrete example of a recursive implementation, although it may not be the most efficient:



Recursive function example 2

- Recursion and lists: Adding every number in a 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: Achieving the same 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

- Calculating factorial with for loops.

```python
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

- Calculating factorial with recursion.

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

print(factorial(5))
```

```
## 120
```

- However, recursion has its limitations. Each time a function calls itself, it allocates memory for the new call. This can lead to errors. In Python, function calls are stopped after reaching a depth of 1000 calls. RecursionError: maximum recursion depth exceeded in comparison
- These limitations are not present when using for loops.

Recursive function example 4.3

```python
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))
```

- Remember that recursion is well-suited for specific 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, regardless of the hierarchy's depth. 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 (functions assigned as variables).
  - 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
# 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.
```

```python
greet(whisper)
```

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

Application of high order function 2 1/5

- Let's 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
```

- These functions share many commonalities, except for their names and the calculation of k.

Application of high order function 2 2/5

- Another example:

```
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)
```

```
## 3.121594652591009
```

- These functions share many commonalities, except for their names and the calculation of k.

Application of high order function 2 3/5

- The presence of this kind of pattern indicates an opportunity for a new level of abstraction, such as passing and as arguments.

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

Application of high order function 2 3/5

- The presence of this kind of pattern indicates an opportunity for a new level of abstraction, such as passing and as arguments.

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

- The pattern can be summarized as follows:

```
#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 4/5

```
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 5/5

```python
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
```

Conclusion 4/4

- Building abstraction is crucial:
  - to make your script more concise (e.g., using functions).
  - to simplify your script (e.g., using modules).
  - in general, to hide the internal implementations of a process or method from the user.
- Abstraction can also add complexity to your program:
  - it's easy to lose track of what your functions or programs are doing, especially with concepts like recursion and higher-order functions.
- Keep in mind that while abstraction is incredibly useful, it should be maintained at a certain level of comprehensibility.
- We will delve further into the topic of abstraction when we work on Object-Oriented Programming (OOP) in Python later in the class.