Programming

Functions

Vassilis Markos, Mediterranean College

Week 04

Contents

1. Last Time’s Exercises

2. Functions

3. Fun Time!

Last Time's Exercises

In–class Exercise #001

Write a Python program that computes the following sum:

$$s = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots + \frac{1}{2^n},$$

where n is provided by the user.
Then, make any modifications needed to compute the following sum:

$$s = 1 - \frac{1}{2} + \frac{1}{4} - \frac{1}{8} + \frac{1}{16} - \cdots \pm \frac{1}{2^n}.$$

In–class Exercise #002

Write a Python program that asks the user for a positive integer, n, and computes its factorial, denoted by $n!$. As a reminder, the factorial of a number, n, is given by the following formula:

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1.$$

So, for instance:

$$\begin{aligned}
3! &= 3 \cdot 2 \cdot 1 = 6, \\
5! &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120, \\
8! &= 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 40320.
\end{aligned}$$

In–class Exercise #003: Part A

In combinatorics, a quite useful quantity is the number of subsets of k elements from a set of n elements, with $k < n$, given by the following formula:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Using code from your previous exercise, write a Python program that asks the user for n, k and computes $\binom{n}{k}$ (which is often called the **binomial coefficient**).

In–class Exercise #003: Part B

Test your previous program on the following input:

$$n = 10000, \quad k = 5000.$$

- How did it perform?
- Can you find a way to fix it? You can, of course, look around the web for ideas.
- Explain the rationale of your solution.

In–class Exercise #004: Part A

A positive integer number, n, is said to be prime if:

- $n > 1$, and;
- the only divisors of n are 1 and n.

Explain why the following program, intended to check whether a number n is prime is wrong:

```python
# source/exercise_004a.py

n = int(input("Please, enter an integer: "))
if n > 1 and n % n ==0 and n % 1 == 0:
    print("Prime!")
else:
    print("Not Prime!")
```

In–class Exercise #004: Part B

A positive integer number, n, is said to be prime if:

- $n > 1$, and;
- the only divisors of n are 1 and n.

Write a Python program that asks the user for a positive integer and computes whether it is prime or not. Test your program on the following inputs:

$$1, 2, 3, 4, 5, 7, 9, 14, 23, 57, 101.$$

The outputs should be (`True` for prime, `False` for non–prime):

`False`, `True`, `True`, `False`, `True`, `True`, `False`, `False`, `True`, `False`, `True`.

In–class Exercise #004: Part C

Test your previous program about primes on the following input:

$$1234567891.$$

- How long did it take to terminate?
- Can you make it run faster?
- How long does your new implementation take on 10101012107?

In–class Exercise #005

Write a Python function that asks the user for the number of rows and columns and prints a shape like what is shown next on screen (this is for 5 rows and 6 columns).

```
*   *   *   *   *   *

*   *   *   *   *   *

*   *   *   *   *   *

*   *   *   *   *   *

*   *   *   *   *   *
```

In–class Exercise #006

Write a Python function that asks the user for the number of rows / columns (it should be the same in this case) and prints a shape like what is shown next on screen (this is for 5 rows / columns).

```
*   *   *   *   *
*   *   *   *
*   *   *
*   *
*
```

In–class Exercise #007

Write a Python function that asks the user for the number of rows / columns (it should be the same in this case) and prints a shape like what is shown next on screen (this is for 5 rows / columns).

```
                *
            *   *
        *   *   *
    *   *   *   *
*   *   *   *   *
```

In–class Exercise #008

Write a Python function that asks the user for the number of rows / columns (it should be the same in this case) and prints a shape like what is shown next on screen (this is for 5 rows / columns).

```
*
*
*
*
* * * * *
```

In–class Exercise #009

Write a Python function that asks the user for the number of rows / columns (it should be the same in this case) and prints a shape like what is shown next on screen (this is for 5 rows / columns).

```
*   *   *   *   *

*               *

*               *

*               *

*   *   *   *   *
```

Functions

Primality Check

As we saw in some of last time’s exercises, one way to check if a number is prime is a follows:

```python
# source/primality_001.py

n = int(input("Please, enter a positive integer: "))
i = 2
if n < 2:
    print("Not Prime!")
else:
    while n % i != 0:
        i = i + 1
    if n == i:
        print("Prime!")
    else:
        print("Not Prime!")
```

Many Primes

What if we want to check for many primes one after the other?

Many Primes

What if we want to check for many primes one after the other?

```python
# source/primality_002.py

n = int(input("Please, enter a positive integer: "))
while n > 0:
    i = 2
    if n < 2:
        print("Not Prime!")
    else:
        while n % i != 0:
            i = i + 1
        if n == i:
            print("Prime!")
        else:
            print("Not Prime!")
    n = int(input("Please, enter a positive integer: "))
```

What About Distribution?

- Suppose we want to share this amazing functionality about primes with other people / programs.

What About Distribution?

- Suppose we want to share this amazing functionality about primes with other people / programs.
- For instance, we would like to write something like the following:

```python
# some/other/file.py
n = int(input("Please, enter an integer: "))
# check if n is prime...
if n is prime:
    # do some stuff
else:
    # do some other stuff
```

What About Distribution?

- Suppose we want to share this amazing functionality about primes with other people / programs.
- For instance, we would like to write something like the following:

```python
# some/other/file.py
n = int(input("Please, enter an integer: "))
# check if n is prime...
if n is prime:
    # do some stuff
else:
    # do some other stuff
```

- What are our options?

What About Distribution?

- One way is to copy-paste the primality check code we have written in `source/primality_001.py`.

What About Distribution?

- One way is to copy–paste the primality check code we have written in `source/primality_001.py`.
- What’s the problem with that?

What About Distribution?

- One way is to copy–paste the primality check code we have written in `source/primality_001.py`.
- What's the problem with that?
- Consider a simple code–maintenance case:

What About Distribution?

- One way is to copy–paste the primality check code we have written in `source/primality_001.py`.
- What's the problem with that?
- Consider a simple code–maintenance case:
  - We find better ways to check primality (as we, actually, did).
  - So, we have to update our code.
  - But, this means that we also have to update any copies we have created.
  - But, what about other people using our code?

What About Distribution?

- One way is to copy–paste the primality check code we have written in `source/primality_001.py`.
- What's the problem with that?
- Consider a simple code–maintenance case:
  - We find better ways to check primality (as we, actually, did).
  - So, we have to update our code.
  - But, this means that we also have to update any copies we have created.
  - But, what about other people using our code?
- Evidently, this is not efficient!

Functions

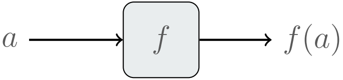
Can you recall the definition of a function from your Maths module?

Functions

Can you recall the definition of a function from your Maths module?

> *A function $f : A \to B$ is a subset $f \subseteq A \times B$ such that* ***for each*** *$a \in A$ there exists a* ***unique*** *$b \in B$ such that $(a, b) \in f$. We write $b = f(a)$ and we call b the value of f at a.*

But, don't worry about this formalism. A useful mental model of a function for our purposes is that of a **black–box:**



a

f

$f(a)$

More Functions

Functions serve as wrappers that enclose some functionality under a certain name. In Python, we can define our own functions as follows:

```python
# source/is_prime_001.py

def is_prime(n):
    i = 2
    if n < 2:
        return False
    else:
        while n % i != 0:
            i = i + 1
        if n == i:
            return True
        else:
            return False
```

The Anatomy Of A Python Function

- In line 3 we define a function using:
  - The reserved keyword `def`;
  - The function's name (`is_prime`);
  - A (potentially empty) comma-separated list of arguments enclosed in parentheses.
- Each function definition is always followed by a colon (`:`).

```
 1  # source/is_prime_001.py
 2
 3  def is_prime(n):
 4      i = 2
 5      if n < 2:
 6          return False
 7      else:
 8          while n % i != 0:
 9              i = i + 1
10          if n == i:
11              return True
12          else:
13              return False
```

The Anatomy Of A Python Function

- Lines 4 – 13 contain what is called the **function body** which is the code that is being executed whenever a function is being called.
- In lines 6, 11, and 13, the reserved keyword `return` is used to indicate the output value the function returns in this case (roughly, $f(a)$ in the above).

```
 1  # source/is_prime_001.py
 2
 3  def is_prime(n):
 4      i = 2
 5      if n < 2:
 6          return False
 7      else:
 8          while n % i != 0:
 9              i = i + 1
10          if n == i:
11              return True
12          else:
13              return False
```

`return` Breaks Code Execution!

Are the following equivalent?

```python
# source/is_prime_001.py

def is_prime(n):
    i = 2
    if n < 2:
        return False
    else:
        while n % i != 0:
            i = i + 1
        if n == i:
            return True
        else:
            return False
```

```python
# source/is_prime_002.py

def is_prime(n):
    i = 2
    if n < 2:
        return False
    while n % i != 0:
        i = i + 1
    if n == i:
        return True
    return False
```

`return` Breaks Code Execution!

Yes, they are!

- `return` **breaks code execution**, which means that any piece of code below a `return` within the same block will not be executed.
- Thus, e.g., if we use `return` within an `if` statement, we need not use an `else` block, since skipping the `if` block is essentially equivalent to not returning in this case.
- Nice trick to save up some typing. Other than that, there is nothing wrong with using code as in the first (leftmost) example!

Calling Functions

Now, using the `is_prime` function our code can be more readable:

```python
# source/is_prime_003.py

def is_prime(n):
    i = 2
    if n < 2:
        return False
    while n % i != 0:
        i = i + 1
    if n == i:
        return True
    return False

n = int(input("Please, enter an integer: "))
while n > 0:
    if is_prime(n):
        print("Prime!")
    else:
        print("Not Prime!")
    n = int(input("Please, enter an integer: "))
```

Calling Functions

We can even use our function from another file:

```python
# source/is_prime_004.py

import is_prime_003

n = int(input("Please, enter an integer: "))
while n > 0:
    if is_prime_003.is_prime(n):
        print("Prime!")
    else:
        print("Not Prime!")
    n = int(input("Please, enter an integer: "))
```

Heads up!

Importing Scripts

Did you observe something strange before?

Importing Scripts

Did you observe something strange before?

- Why did the script not terminate when you first entered a non–positive integer?

Heads up!

Importing Scripts

Did you observe something strange before?

- Why did the script not terminate when you first entered a non−positive integer?
- Also, why did it terminate the second time?

Importing Scripts

Did you observe something strange before?

- Why did the script not terminate when you first entered a non–positive integer?
- Also, why did it terminate the second time?
- When importing scripts, Python roughly “copy–pastes” any code from the imported script into the new one.

Importing Scripts

Did you observe something strange before?

- Why did the script not terminate when you first entered a non–positive integer?
- Also, why did it terminate the second time?
- When importing scripts, Python roughly "copy–pastes" any code from the imported script into the new one.
- That is, any code that is executable in the imported script is also executed in the importing script.

Importing Scripts

Did you observe something strange before?

- Why did the script not terminate when you first entered a non–positive integer?
- Also, why did it terminate the second time?
- When importing scripts, Python roughly “copy–pastes” any code from the imported script into the new one.
- That is, any code that is executable in the imported script is also executed in the importing script.
- But, what if we don’t want that behaviour?

Importing Scripts

In order to inform Python that the executable part of a script should not be executed when importing this we can use the following:

```
<non executable code>
if __name__ == "__main__":
    <executable code>
```

Under the `if` statement we put any code we want to be executed whenever the script is directly called while outside it we put any “importable” code, e.g., function declarations.

Proper Imports

```python
# source/is_prime_005.py

def is_prime(n):
    i = 2
    if n < 2:
        return False
    while n % i != 0:
        i = i + 1
    if n == i:
        return True
    return False

if __name__ == "__main__":
    n = int(input("Please, enter an integer: "))
    while n > 0:
        if is_prime(n):
            print("Prime!")
        else:
            print("Not Prime!")
        n = int(input("Please, enter an integer: "))
```

```python
# source/is_prime_006.py

import is_prime_005

n = int(input("Please, enter an integer: "))
while n > 0:
    if is_prime_005.is_prime(n):
        print("Prime!")
    else:
        print("Not Prime!")
    n = int(input("Please, enter an integer: "))
```

Shorter Imports

```python
# source/is_prime_005.py

def is_prime(n):
    i = 2
    if n < 2:
        return False
    while n % i != 0:
        i = i + 1
    if n == i:
        return True
    return False

if __name__ == "__main__":
    n = int(input("Please, enter an integer: "))
    while n > 0:
        if is_prime(n):
            print("Prime!")
        else:
            print("Not Prime!")
        n = int(input("Please, enter an integer: "))
```

```python
# source/is_prime_007.py

from is_prime_005 import is_prime

n = int(input("Please, enter an integer: "))
while n > 0:
    if is_prime(n):
        print("Prime!")
    else:
        print("Not Prime!")
    n = int(input("Please, enter an integer: "))
```

Fancy Imports

```python
# source/is_prime_005.py

def is_prime(n):
    i = 2
    if n < 2:
        return False
    while n % i != 0:
        i = i + 1
    if n == i:
        return True
    return False

if __name__ == "__main__":
    n = int(input("Please, enter an integer: "))
    while n > 0:
        if is_prime(n):
            print("Prime!")
        else:
            print("Not Prime!")
        n = int(input("Please, enter an integer: "))
```

```python
# source/is_prime_008.py

from is_prime_005 import is_prime as ip

n = int(input("Please, enter an integer: "))
while n > 0:
    if ip(n):
        print("Prime!")
    else:
        print("Not Prime!")
    n = int(input("Please, enter an integer: "))
```

Why Use Functions?

- As discussed above, because using functions improves code **portability** and **distribution.**

Why Use Functions?

- As discussed above, because using functions improves code **portability** and **distribution.**
- Also, it is much (much) **easier to debug** code split into simple distinct functions, as the root cause is usually easier spotted.

Why Use Functions?

- As discussed above, because using functions improves code **portability** and **distribution.**
- Also, it is much (much) **easier to debug** code split into simple distinct functions, as the root cause is usually easier spotted.
- It resembles human thought when it comes to **solving more complex problems,** e.g, splitting a larger problem into simpler subtasks and then repeat that process, if needed, until the resulting tasks are easy enough the be addressed separately.

Why Use Functions?

- As discussed above, because using functions improves code **portability** and **distribution.**
- Also, it is much (much) **easier to debug** code split into simple distinct functions, as the root cause is usually easier spotted.
- It resembles human thought when it comes to **solving more complex problems,** e.g, splitting a larger problem into simpler subtasks and then repeat that process, if needed, until the resulting tasks are easy enough the be addressed separately.
- It **makes code more readable,** which, in general, facilitates both distribution and debugging.

What Will This Print?

```python
# source/foo_001.py

def foo(x):
    x = x + 4
    return x

if __name__ == "__main__":
    x = 6
    y = foo(x)
    print(f"x: {x}, y: {y}.")
```

What Happens In Vegas...

- x at line 3 is a **local variable**, i.e., a variable that is defined within the scope of the function and destroyed after its execution.
- So, x at line 3 is not the same as x at line 8.
- In general, this is a common quirk of functions: any argument we define (often called *parameter*) stays local, so no effects appear outside.

```python
1  # source/foo_001.py
2
3  def foo(x):
4      x = x + 4
5      return x
6
7  if __name__ == "__main__":
8      x = 6
9      y = foo(x)
10     print(f"x: {x}, y: {y}.")
```

Fun Time!

In–class Exercise #001

Write a Python function that:

- takes a single integer as an argument, and;
- returns `True` or `False` depending on whether this number is even or odd.

Demonstrate the functionality of your function by properly using it in a simple Python script.

In–class Exercise #002

The Fibonacci numbers, f_n, are a sequence of integer numbers given by the following relation:

$$f_n = f_{n-1} + f_{n-2}, \quad f_0 = 0,\ f_1 = 1.$$

That is, each term is the sum of its previous two. For instance, the first 10 Fibonacci numbers are:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34.$$

Write a Python function that takes n as input and prints the n–th Fioinacci number, f_n.

In–class Exercise #003

Write a Python program that:

- asks the user for consecutive positive integers (non–positive input terminates number insertion), and;
- computes and prints their sum and average.

You are required to use **at least three different functions** for your solution and explain your rationale!

In–class Exercise #004

The standard deviation, s, of a set of n numbers $x_1, x_2, \ldots, x_n$ is computed by the following formula:

$$s = \sqrt{\frac{(x_1 - \mu)^2 + (x_2 - \mu)^2 + \cdots + (x_n - \mu)^2}{n}},$$

where μ is the mean value of those numbers.
Write a Python program that asks the user for some non–zero numbers (insertion terminated by inserting 0) and computes their standard deviation. Make sure your program uses **at least two functions!**

In–class Exercise #005

A string is said to be a **palindrome** if it reads the same left–to–right and right–to–left. Write a Python function that:

- takes a single string as an argument, and;
- returns `True` or `False` depending on whether this string is a palindrome.

Demonstrate the functionality of your function by properly using it in a simple Python script.

Hint: In order to access the i*–th character of a string named* `s` *you can use the syntax* `s[i]`*.*

In–class Exercise #006

One way to estimate the square root of a positive float, a, is to use the following method:

$$x_n = \frac{1}{2}\left(x_{n-1} + \frac{a}{x_{n-1}}\right),$$

where the first estimate, x_0, is an arbitrary positive float. We say that x_n is an estimation of $\sqrt{a}$ of accuracy $\varepsilon > 0$ if $|x_n - x_{n-1}| < \varepsilon$, i.e., if the two latest estimates we have made are no further apart than ε.

Write a Python function that takes x_0, a, and ε as arguments and returns the corresponding estimate, x_n.

In–class Exercise #007

The Towers of the Hanoi is a well–known puzzle where you have to move disks of different sizes one at a time from a peg to another peg with the help of an auxiliary peg and without ever moving a larger disk on top of a smaller one. You can familiarise yourselves with the game below:

`https://www.mathsisfun.com/games/towerofhanoi.html`

Develop a Python function that accepts a positive integer n corresponding to the number of disks on the first peg and prints on screen the required steps to solve the problem.

In–class Exercise #008

Start working on all Labs found in today's materials `homework` directory. To help me assess those files, you can name them as follows:

```
task_xxx.py
```

where `xxx` is the number of the task. For instance, task 5 file could be named `task_005.py`.
Submit your work via email at: `v.markos@mc-class.gr`

Homework

- In this week's materials, under the `homework` directory, you can find some Python programming Tasks. Complete as many of them as you can (preferably all).
- This is important, since tasks such as those provided with this lecture's materials will most probably be part of your course assessment portfolio. So, take care to solve as many of those tasks as possible!
- Share your work at: `v.markos@mc-class.gr`

Any Questions?

Do not forget to fill in the questionnaire shown right!

https://forms.gle/dKSrmE1VRVWqxBGZA