Introduction to Programming Homework 2

Due Wednesday October 5 by 14h00

You will turn in your homework via e-mail (andrew.yarmola@uni.lu). For this homework, you will work in a text editor of your choosing. I recommend `spyder` (included with Anaconda) or get the popular text editor Atom (atom.io). You can also use/learn `vim` or `emacs` if you have the time; they can be very useful to know. For each excercise you will create a text file `module_name.py`. In the text file you will define functions which you can then test and load as modules.

To test your work

- Put your all of your `module_name.py` files into a directory. For example `C:\Users\Andrew\itp\homework2\` on Windows or `/Users/Andrew/itp/homework2/` on macOS.
- Start `ipython` and use the following command to change your working directory. Replace the directory path as necessary.

  ```
  %cd "/Users/Andrew/itp/homework2/"
  ```

- Now you should be able to load your modules by

  ```
  import module_name
  module_name.function_to_test(args)
  ```

- If you make changes to your `module_name.py`, you will need to **reload** the module. To do this, use the commands

  ```
  import importlib
  importlib.reload(module_name)
  ```

  Note that you only need to call `import importlib` once and then `importlib.reload(module_name)` every time you want to reload your work. You can testing examples at the end of this homework.

Remark

If I say that the input will be of a certain type, feel free to assume that it will always be so.

Exercise 1 (Classic examples)

Create a module called `classic.py`. Inside the module define the following functions. **Do not import any additional modules inside `classic.py`**

- **a.** Write a function called `fibonacci` which taken an integer `n` and return F_n, the n^{th} Fibonacci number with $F_n = 0$ for $n < 1$ and $F_1 = 1$. For example, `classic.fibonacci(6)` should return `8`.

- **b.** Write a function called `golden_ratio` which takes an integer `n` and returns the golden ratio approximation using F_{n+1}/F_n. If `n < 1`, return `1`.

- **c.** Write a function called `wallis_pi` which takes an integer `n` and returns an approximation to π using the product of the first `n` multiplicands of the Wallis formula
$$\pi = 2\prod_{k=1}^{\infty}\frac{4k^2}{4k^2-1}$$

- **d.** Write a function called `collatz` which takes a *positive* integer `n` and returns the number of steps in the Collatz (or Syracuse) sequence it takes to reach `1`. For example, `classic.collatz(10)` should return `6`.
  - Hint : feel free to adapt the code from the lecture

Excercise 2 (Permutations)

Create a module called `perms.py`. Inside the module define the following functions. **Do not import any additional modules inside `perms.py`**

In this exercise, a **permutation** will be a `tuple` that contain the numbers `0, ..., n` **exactly once**. For example `a = (2,1,0)` is a permutation where `a` is the map `a[0] = 2, a[1] = 1, a[2] = 0`. Tuples like `(1,1,2)` and `(3,2,1)` are not permutations.

- **a.** Write a function called `is_perm` which takes a `tuple` and returns `True` if the list is a permutation and `False` otherwise.

- **b.** Write a function called `compose` which takes two tuples `a` and `b` and returns $b \circ a$ if both are permutations and `()` if `a` or `b` is not a permutation **or are not composable**. For example, `perms.compose((0,2,1),(0,2,1))` should return `(0,1,2)`, while `perms.compose((0,2,1),(1,2,1))` should return `()`.

Exercise 3 (Base 2)

Create a module called `base_2.py`. Inside the module define the following functions. **Do not import any additional modules inside `base_2.py`**

- **a.** Write a function called `bits_needed` which takes an integer `n` and returns the length of the binary number needed to represent it, **including the sign**. For example, `base_2.bits_needed(8)` should return `5` because `8` is `+1000` in binary. Similarly, `base_2.bits_needed(-17)` should return `6` because `-17` is `-10001`.

- **b.** Write a function called `is_power_of_2` which takes an integer `n` and returns `True` if `n` is a power of 2 and `False` otherwise. For example `base_2.is_power_of_2(8)` should return `True` and `base_2.is_power_of_2(-3)` should return `False`. Note, $1 = 2^0$.

- **c.** Write a function called `bad_log_base_2` which takes an integer `n` and returns `-1` if `n` is **not** a power of `2` and returns the integer $log_2(n)$ is `n` is a power of `2`. For example, `base_2.bad_log_base_2(8)` should return `3` while `base_2.bad_log_base_2(-3)` should return `-1`.

Test you work !!!

Here is a **non-exhaustive** list of commands that you can use in `ipython` to test your work. All of the print calls should print `True` and no errors should appear if your code is working correctly.

In [ ]:

```
import importlib
import classic
import perms
import base_2

# Exercise 1
importlib.reload(classic) # Just in case you need to reload to the latest chan
ges
# a.
print(classic.fibonacci(0) == 0)
print(classic.fibonacci(2) == 1)
print(classic.fibonacci(6) == 8)
# b.
print(classic.golden_ratio(1) == 1.)
print(classic.golden_ratio(3) == 3/2)
print(classic.golden_ratio(6) == 13/8)
# c.
print(classic.wallis_pi(0) == 2.)
print(classic.wallis_pi(100) == 882082760940707/281474976710656)
# d.
print(classic.collatz(10) == 6)
print(classic.collatz(100) == 25)

# Exercise 2
importlib.reload(perms) # Just in case you need to reload to the latest change
s
# a.
print(perms.is_perm((2,1,0)) == True)
print(perms.is_perm((2,1,3)) == False)
print(perms.is_perm((2,1,1)) == False)

# b.
print(perms.compose((0,2,1),(0,2,1)) == (0,1,2))
print(perms.compose((0,2,1),(1,2,1)) == ())
print(perms.compose((0,2,1),(1,0)) == ())
print(perms.compose((0,1),(1,0,2)) == ())
print(perms.compose((0,2,1,3),(3,2,1,0)) == (3, 1, 2, 0))
```

```
# Exercise 3
importlib.reload(base_2) # Just in case you need to reload to the latest chang
es

# a.
print(base_2.bits_needed(8) == 5)
print(base_2.bits_needed(-17) == 6)

# b.
print(base_2.is_power_of_2(2**42) == True)
print(base_2.is_power_of_2(-2) == False)
print(base_2.is_power_of_2(0) == False)
print(base_2.is_power_of_2(3) == False)

# c.
print(base_2.bad_log_base_2(8) == 3)
print(base_2.bad_log_base_2(2**47) == 47)
print(base_2.bad_log_base_2(-2) == -1)
print(base_2.bad_log_base_2(0) == -1)
print(base_2.bad_log_base_2(3) == -1)
```