Introduction to Programming Homework 2 Solutions

Note : There are (many) diffenrent ways to answer many of these questions. I have placed the solutions in-line with the questions, however, they should be contaiend in their own module files.

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`.

In [5]:

```
def fibonacci(n) :
    if n < 1 :
        return 0
    prev = 0
    curr = 1
    for i in range(n-1) :
        temp = curr
        curr += prev
        prev = temp
    return curr
```

- **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.`

In [6]:

```
# Note : I am not reusing the fibonacci function here
# because this is more efficient than computing F_n
# and F_{n+1} separately.
def golden_ratio(n) :
    if n < 2 :
        return 1.
    prev = 0
    curr = 1
    for i in range(n) :
        temp = curr
        curr += prev
        prev = temp
    return curr/prev
```

- **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

$$\pi = 2\prod_{k=1}^{\infty} \frac{4k^2}{4k^2 - 1}$$

In [7]:

```
def wallis_pi(n) :
    result = 2.
    for k in range(1,n+1) :
        result *= (4. * k**2)/(4. * k**2 - 1.)
    return result
```

- **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

In [8]:

```
def collatz(n) :
    steps = -1
    while n > 0 : # Don't run for negative numbers
        steps += 1
        if n == 1 :
            break
        if n % 2 == 0 :
            n = n // 2
        else :
            n = 3*n +1
    return steps
```

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.

In [9]:

```
def is_perm(p) :
    if type(p) is not tuple :
        return False
    return sorted(p) == list(range(len(p)))
```

- **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 `()`.

In [10]:

```
def compose(a,b) :
    if (not is_perm(a)
     or not is_perm(b)
     or len(a) != len(b)) :
        return ()
    result = [ b[a[i]] for i in range(len(a)) ]
    return tuple(result)
```

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`.

In [ ]:

```
def bits_needed(n) :
    if n == 0 :
        return 0
    return len('{0:+b}'.format(n))
```

- **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, $1 = 2^0$.

In [ ]:

```
def is_power_of_2(n) :
    if n < 1 :
        return False
    # We use the signed formatting binary string
    binary = '{0:+b}'.format(n)
    # binary starts with +1, if there is a 1
    # afterwards, then we are not a power
    if '1' in binary[2:] :
        return False
    else :
        return True
```

- **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`.

In [ ]:

```
def bad_log_base_2(n) :
    if not is_power_of_2(n) :
        return -1
    else :
        return bits_needed(n) - 2
```