module2

January 22, 2024

0.1 Module 2

This HW deals primarily with loops and functions. It will introduce you to some interesting number theory as well. You have probably heard of prime numbers and composite numbers before, but have you ever heard of abundant numbers, or narcissistic numbers? This assignment will ask you to write code to identify different kinds of number properties.

We also want you to learn how to reuse code in this assignment, by writing and using functions. This is a basic strategy for reducing complexity in a program. Why? Because things change, and if you have the same thing in several places, when you change one, you have to change all the others. And you can't always find them all! In order to reuse code, you must take common pieces of code and put them into functions. Failure to do so will result in loss of points.

We also want you to add comments to your code and docstrings to your functions.

You can assume that all inputs in this assignment will be positive integers.

Each function has been defined for you, but without the code. See the docstring in each function (in the starter code) for more details on what the function is supposed to do and how to write the code. It should be clear enough. In some cases, we have provided hints and examples to get you started.

[9]:
```
############################################################
### EXECUTE THIS CELL BEFORE YOU TO TEST YOUR SOLUTIONS ###
############################################################

import unittest
from nose.tools import assert_equal, assert_true
```

[10]:
```
def getFactors(x):
    """Returns a list of factors of the given number x.
    Basically, finds the numbers between 1 and the given integer that divide␣
 ↪the number evenly.

    For example:
    - If we call getFactors(2), we'll get [1, 2] in return
    - If we call getFactors(12), we'll get [1, 2, 3, 4, 6, 12] in return
    """

    # your code here
```

```
    value = 1
    l=[]
    while value <= x :
        if x % value == 0 :
            l.append(value)
        value += 1
    return l
```

[11]:
```
############################
### TEST YOUR SOLUTION ###
############################

num = 2
factors_test = [1, 2]
factors = getFactors(num)
assert_equal(factors_test, factors, str(factors) + ' are not the factors of ' +␣
 ↪str(num))

num = 12
factors_test = [1, 2, 3, 4, 6, 12]
factors = getFactors(num)
assert_equal(factors_test, factors, str(factors) + ' are not the factors of ' +␣
 ↪str(num))

num = 13
factors_test = [1, 13]
factors = getFactors(num)
assert_equal(factors_test, factors, str(factors) + ' are not the factors of ' +␣
 ↪str(num))

# test existence of docstring
assert_true(len(getFactors.__doc__) > 1, "there is no docstring for getFactors")
print("Success!")
```

```
Success!
```

[12]:
```
def isPrime(x):
    """Returns whether or not the given number x is prime.

    A prime number is a natural number greater than 1 that cannot be formed
    by multiplying two smaller natural numbers.

    For example:
    - Calling isPrime(11) will return True
    - Calling isPrime(71) will return True
    - Calling isPrime(12) will return False
```

```
    - Calling isPrime(76) will return False
    """

    # your code here
    if x == 1:
        return False
    elif x > 1:
        for i in range(2,x):
            if (x % i) == 0:
                return False
                break
        else:
            return True
    else:
        return False
```

[13]:
```
############################
### TEST YOUR SOLUTION ###
############################

prime_numbers = [2, 3, 5, 7, 11, 13, 17, 19, 23,
     29, 31, 37, 41, 43, 47, 53, 59,
     61, 67, 71, 73, 79, 83, 89, 97,
     101, 103, 107, 109, 113, 127,
     131, 137, 139, 149, 151, 157,
     163, 167, 173, 179, 181]

for i in prime_numbers:
    assert_true(isPrime(i), str(i) + ' is prime')

not_prime_numbers = [1, 8, 12, 18, 20, 27, 28, 30,
     42, 44, 45, 50, 52, 63, 66,
     68, 70, 75, 76, 78, 92, 98,
     99, 102, 138, 148, 150, 156, 158]

for i in not_prime_numbers:
    assert_true(not(isPrime(i)), str(i) + ' is not prime')

#test existence of docstring
assert_true(len(isPrime.__doc__) > 1, "there is no docstring for isPrime")
print("Success!")
```

```
Success!
```

[14]:
```
def isComposite(x):
    """Returns whether or not the given number x is composite.
```

```
    A composite number has more than 2 factors.
    A natural number greater than 1 that is not prime is called a composite␣
 ↪number.
    Note, the number 1 is neither prime nor composite.

    For example:
    - Calling isComposite(9) will return True
    - Calling isComposite(22) will return True
    - Calling isComposite(3) will return False
    - Calling isComposite(41) will return False
    """

    # your code here
    n = 0
    for i in range(1, x+1):
        if x % i == 0:
            n += 1

    if n > 2:
        return True
    else:
        return False
```

[15]:
```
############################
### TEST YOUR SOLUTION ###
############################

composite_numbers = [4, 6, 8, 9, 10, 12, 14, 15, 16,
     18, 20, 21, 22, 24, 25, 26, 27,
     28, 30, 32, 33, 34, 35, 36,
     38, 39, 40, 42, 44, 45, 46, 48,
     49, 50, 51, 52, 54, 55, 56, 57,
     58, 60, 62, 63, 64, 65, 66, 91, 93]

for i in composite_numbers:
    assert_true(isComposite(i), str(i) + ' is composite')

not_composite_numbers = [1, 2, 3, 5, 7, 11, 13, 17, 19, 23,
     29, 31, 37, 41, 43, 47, 53, 59,
     61, 67, 71, 73, 79, 83, 89]

for i in not_composite_numbers:
    assert_true(not(isComposite(i)), str(i) + ' is not composite')

#test existence of docstring
assert_true(len(isComposite.__doc__) > 1, "there is no docstring for␣
 ↪isComposite")
```

```
print("Success!")
```

```
Success!
```

[16]:
```
def isPerfect(x):
    """Returns whether or not the given number x is perfect.

    A number is said to be perfect if it is equal to the sum of all its
    factors (for obvious reasons the list of factors being considered does
    not include the number itself).

    Example: 6 = 3 + 2 + 1, hence 6 is perfect.
    Example: 28 is another example since 1 + 2 + 4 + 7 + 14 is 28.
    Note, the number 1 is not a perfect number.
    """

    # your code here
    sum = 0

    for i in range(1, x):
        if x % i == 0:
            sum = sum + i

    if sum == x:
        return True
    else:
        return False
```

[17]:
```
perfect_numbers = [6, 28, 496, 8128, 33550336]

for i in perfect_numbers:
    assert_true(isPerfect(i), str(i) + ' is perfect')

not_perfect_numbers = [2, 3, 4, 5, 7, 8, 9, 10,
                       495, 8127, 8129,
                       33550335]

for i in not_perfect_numbers:
    assert_true(not(isPerfect(i)), str(i) + ' is not perfect')

#test existence of docstring
assert_true(len(isPerfect.__doc__) > 1, "there is no docstring for isPerfect")
print("Success!")
```

```
Success!
```

[18]:
```
def isAbundant(x):
    """Returns whether or not the given number x is abundant.

    A number is considered to be abundant if the sum of its factors
    (aside from the number) is greater than the number itself.

    Example: 12 is abundant since 1+2+3+4+6 = 16 > 12.
    However, a number like 15, where the sum of the factors.
    is 1 + 3 + 5 = 9 is not abundant.
    """

    # your code here
    s = 0
    for i in range(1, x):
        if x % i == 0:
            s += i
    if s > x:
        return True
    else:
        return False
```

[19]:
```
abundant_numbers = [12, 18, 20, 24, 30, 36, 40, 42, 48,
    54, 56, 60, 66, 70, 72, 78, 80, 84,
    88, 90, 96, 100, 102, 104, 108, 112,
    114, 120]

for i in abundant_numbers:
    assert_true(isAbundant(i), str(i) + ' is abundant')

not_abundant_numbers = [1, 2, 3, 4, 5, 6,
    7, 8, 9, 10, 11, 13,
    14, 15, 16, 17, 19,
    21, 22, 23, 25, 26, 27, 28, 29,
    91, 92, 93, 94, 95, 119]

for i in not_abundant_numbers:
    assert_true(not(isAbundant(i)), str(i) + ' is not abundant')

#test existence of docstring
assert_true(len(isAbundant.__doc__) > 1, "there is no docstring for isAbundant")
print("Success!")
```

```
Success!
```

[20]:
```
def isTriangular(x):
    """Returns whether or not a given number x is triangular.
```

```
    The triangular number Tn is a number that can be represented in the form of a triangular
    grid of points where the first row contains a single element and each subsequent row contains
    one more element than the previous one.

    We can just use the fact that the nth triangular number can be found by using a formula: Tn = n(n + 1) / 2.

    Example: 3 is triangular since 3 = 2(3) / 2
    3 --> 2nd position: (2 * 3 / 2)

    Example: 15 is triangular since 15 = 5(6) / 2
    15 --> 5th position: (5 * 6 / 2)
    """

    # your code here
    if (x < 0):
        return False
    sum, n = 0, 1
    while(sum <= x):

        sum = sum + n
        if (sum == x):
            return True
        n += 1

    return False
```

[21]:
```
triangular_numbers = [1, 3, 6, 10, 15, 21, 28, 36, 45, 55,
      66, 78, 91, 105, 120, 136, 153, 171,
      190, 210, 231]

for i in triangular_numbers:
    assert_true(isTriangular(i), str(i) + ' is triangular')

not_triangular_numbers = [2, 4, 5, 7, 8, 9, 11, 12, 13, 14,
      16, 17, 18, 19, 20,
      22, 23, 24, 25, 26, 27,
      29, 30, 31, 32, 33, 34, 35,
      37, 40, 41, 42, 43, 44,
      54, 56, 189, 191, 209, 211, 230, 232]

for i in not_triangular_numbers:
    assert_true(not(isTriangular(i)), str(i) + ' is not triangular')

#test existence of docstring
```

```
assert_true(len(isTriangular.__doc__) > 1, "there is no docstring for␣
↪isTriangular")
print("Success!")
```

```
Success!
```

[22]:
```
def isNarcissistic(x):
    """Returns whether or not a given number is Narcissistic.

    A positive integer is called a narcissistic number if it
    is equal to the sum of its own digits each raised to the
    power of the number of digits.

    Example: 153 is narcissistic because 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153.
    Note that by this definition all single digit numbers are narcissistic.
    """

    # your code here
    sum = 0
    length = len(str(x))
    for i in str(x):
        sum = sum + int(i) ** length
    if (x == sum):
        return True
    else:
        return False
```

[23]:
```
narcissistic_numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]

for i in narcissistic_numbers:
    assert_true(isNarcissistic(i), str(i) + ' is narcissistic')

non_narcissistic_numbers = [10, 11, 12, 13, 152, 154, 369, 372, 406]

for i in non_narcissistic_numbers:
    assert_true(not(isNarcissistic(i)), str(i) + ' is not narcissistic')

#test existence of docstring
assert_true(len(isNarcissistic.__doc__) > 1, "there is no docstring for␣
↪isNarcissistic")
print("Success!")
```

```
Success!
```

[24]:
```
def main():

    playing = True
```

```
    while playing == True:

        num_input = input('Give me a number from 1 to 10000.  Type -1 to exit.␣
↪')

        try:
            num = int(num_input)

            if (num == -1):
                playing = False
                continue

            if (num <= 0 or num > 10000):
                continue

            factors = getFactors(num)
            print("The factors of", num, "are", factors)

            if isPrime(num):
                print(str(num) + ' is prime')
            if isComposite(num):
                print(str(num) + ' is composite')
            if isPerfect(num):
                print(str(num) + ' is perfect')
            if isAbundant(num):
                print(str(num) + ' is abundant')
            if isTriangular(num):
                print(str(num) + ' is triangular')
            if isNarcissistic(num):
                print(str(num) + ' is narcissistic')

        except ValueError:
            print('Sorry, the input is not an int.  Please try again.')

#This will automatically run the main function in your program
#Don't change this
if __name__ == '__main__':
    main()
```

```
Give me a number from 1 to 10000.  Type -1 to exit. 876
The factors of 876 are [1, 2, 3, 4, 6, 12, 73, 146, 219, 292, 438, 876]
876 is composite
876 is abundant
Give me a number from 1 to 10000.  Type -1 to exit. -1
```

[ ]: