CS1001.py

Extended Introduciton to Computer Science with Python, Tel-Aviv University, Spring 2013

Recitation 2 - 7-11.3.2013

Last update: 5.3.2013

Getting input from the user

You can get input from the user by calling the `input` function:

```
m = int(input("enter a positive integer to apply Collatz algorithm: "))
```

Note that `input` returns a *string* and therefore you are responsible to convert it to the appopriate type.

Collatz Conjecture

The Collatz Conjecture (also known as the *3n+1* conjecture) is the conjecture that the following process is finite for every natural number: > If the number n is even divide it by two $(n/2)$, if it is odd multiply it by 3 and add 1 $(3n+1)$. Repeat this process untill you get the number 1.

Implementation

We start with the "Half Or Triple Plus One" process:

```
m = 100 # integer to apply the conjecture on

n = m
while n != 1:
    print(n)
    if n % 2 == 0:
        n = n // 2
    else:
        n = 3 * n + 1
print(1) # 1 was not printed
print(m, "is OK")
```

```
100
50
25
76
38
19
58
29
88
44
22
11
34
17
52
26
13
40
20
10
5
16
8
4
2
1
100 is OK
```

Next we add another loop that will run the conjecture check on a range of numbers:

```
limit = 10
m = 1
while m <= limit:
    n = m
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
    print(m, "is OK")
    m += 1
```

```
1 is OK
2 is OK
```

```
3 is OK
4 is OK
5 is OK
6 is OK
7 is OK
8 is OK
9 is OK
10 is OK
```

When a loop goes over a simple range it is easier to use the `range` function with a `for` loop - and more robust against bugs:

```
start, stop = 99, 110
for m in range(start, stop + 1):
    n = m
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
    print(m, "is OK")
```

```
99 is OK
100 is OK
101 is OK
102 is OK
103 is OK
104 is OK
105 is OK
106 is OK
107 is OK
108 is OK
109 is OK
110 is OK
```

Lists

Lists are sequences of values.

Lists can contain a mix of types:

```
mixed_list = [3, 5.14, "hello", True]
print(mixed_list, type(mixed_list))
```

```
[3, 5.14, 'hello', True] <class 'list'>
```

Lists are indexable, starting at 0:

```
mixed_list[0]
```

```
3
```

```
mixed_list[2]
```

```
'hello'
```

Negative indices are counted from the tail:

```
mixed_list[-1]
```

```
True
```

```
mixed_list[-2] == mixed_list[2]
```

```
True
```

Lists can be sliced:

```
mixed_list[1:3]
```

```
[5.14, 'hello']
```

```
mixed_list[:2]
```

```
[3, 5.14]
```

```
mixed_list[1:]
```

```
[5.14, 'hello', True]
```

```
mixed_list[:-2]
```

```
[3, 5.14]
```

Lists can be concatenated:

```
mixed_list + [1, 2, 3]
```

```
[3, 5.14, 'hello', True, 1, 2, 3]
```

But this doesn't change the list, but creates a new list:

```
mixed_list
```

```
[3, 5.14, 'hello', True]
```

```
mixed_list = mixed_list + [1, 2, 3]
mixed_list
```

```
[3, 5.14, 'hello', True, 1, 2, 3, 1, 2, 3]
```

Some functions can be used on lists:

```
numbers = [10, 3, 1, 56]
numbers
```

```
[10, 3, 1, 56]
```

```
sum(numbers)
```

```
70
```

```
len(numbers)
```

```
4
```

```
print(sorted(numbers))
print(numbers)
print(numbers.sort())
print(numbers)
```

```
[1, 3, 10, 56]
[1, 3, 10, 56]
None
[1, 3, 10, 56]
```

Lists are iterable:

```
for item in mixed_list:
    if type(item) == str:
        print(item)
```

```
hello
```

```
for i in range(len(mixed_list)):
    if i == 2:
        print(mixed_list[i])
```

```
hello
```

A list of numbers can be created using *list comprehension*. The syntax is: `[**statement** for **variable** in **iterable** if **condition**]` The `if **condition**` part is optional, the statement and the condition can use variable.

Create a list of the squares of numbers between 1 and 10:

```
[x ** 2 for x in range(1, 11)]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Create a list of the square roots of odd numbers between 1 and 20:

```
[x ** 0.5 for x in range(1, 21) if x % 2 == 1]
```

```
[1.0,
 1.7320508075688772,
 2.23606797749979,
 2.6457513110645907,
 3.0,
 3.3166247903554,
 3.605551275463989,
 3.872983346207417,
 4.123105625617661,
 4.358898943540674]
```

Grades problem

Given a list of grades, count how many are above the average.

```
grades = [33, 55,45,87,88,95,34,76,87,56,45,98,87,89,45,67,45,67,76,73,33,87,12,100,77,89,92
```

```
avg = sum(grades)/len(grades)
above = 0
for gr in grades:
```

```
    if gr > avg:
        above += 1

print(above, "grades are above the average", avg)
```

```
15 grades are above the average 68.07407407407408
```

Using *list comprehension*:

```
avg = sum(grades)/len(grades)
above = len([gr for gr in grades if gr > avg])

print(above, "grades are above the average", avg)
```

```
15 grades are above the average 68.07407407407408
```

Functions

Maximum and minimum

```
def max2(a,b):
    if a >= b:
        return a
    else:
        return b

def min2(a,b):
    if a <= b:
        return a
    else:
        return b

def max3(a,b,c):
    if a >= b and a >= c:
        return a
    elif b >= a and b >= c:
        return b
    else:
        return c

def max3v2(a,b,c):
    max_ab = max2(a,b)
    return max2(max_ab,c)
```

```
print(max2(5,10))
print(min2(5,10))
print(max3(5,10,10))
print(max3v2(5,10,10))
```

```
10
5
10
10
```

- Which should be faster, `max3` or `max3v2`?
- How would you implement `max4`?

Perfect numbers

A perfect number is a number that is equal to the sum of its divisors:

```
def is_perfect(n):
    '''
    is_perfect(integer) -> bool
    Return True iff n equals the sum of its divisors
    '''
    if n == sum(divisors(n)):
        return True
    else:
        return False

help(is_perfect)
```

```
Help on function is_perfect in module __main__:

is_perfect(n)
    is_perfect(integer) -> bool
    Return True iff n equals the sum of its divisors

def divisors(n):
    '''
    divisors(integer) -> list of integers
    Return the proper divisors of n (numbers less than n that divide evenly into n).
    '''
    return [div for div in range(1,n) if n % div == 0]
```

Notes

- Functions that return a boolean are named with `is_` as a prefix.
- Use `"'` after the function definition to create function documentation
- in `is_perfect` we can return the condition value instead of using the `if-else` clause:

```
def is_perfect(n):
    '''
    is_perfect(integer) -> bool
    Return True iff n equals the sum of its divisors
    '''
    return n == sum(divisors(n))

print("perfect numbers in range(1,1000)\n")
print([n for n in range(1,1001) if is_perfect(n)])
```

```
perfect numbers in range(1,1000)

[6, 28, 496]
```

Complexity We can write another version of `divisors` that is more efficient by iterating only on numbers between 1 and $\sqrt{n}$, but this function is more complex and bugs are crawling in it:

```
def divisors2(n):
    divs = []
    for m in range(2,round(n ** 0.5)):#1 and n**0.5 will be handled separately. why?
        if n % m == 0:
            divs += [m, n // m]
    divs += [1] # 1 surely divides n
    if round(n ** 0.5) ** 2 == n: # if n is the square of an integer
        divs += [round(n ** 0.5)]
    return divs
```

Fin

This notebook is part of the Extended introduction to computer science course at Tel-Aviv University.

The notebook was written using Python 3.2 and IPython 0.13.1.

The code is available at https://raw.github.com/yoavram/CS1001.py/master/recitation2.ipynb.

The notebook can be viewed online at http://nbviewer.ipython.org/urls/raw.github.com/yoavram/CS1001.py/master/recitation2.ipynb.

The notebooks is also available as a PDF at https://github.com/yoavram/CS1001.py/blob/master/recitation2.pdf?raw=true.