Functional Programming

Python Functions

- functions are "first class" objects, i.e., a program entity that can be created at runtime
  - assigned to a variable or element in a data structure
  - passed as an argument to a function
  - returned as the result of a function

In [1]:
```
def fact(n):
    '''returns n!
    '''
    if n < 2:
        return 1
    else:
        return n * fact(n - 1)

fact(3), fact(52)
```

Out[1]:
```
(6, 80658175170943878571660636856403766975289505440883277824000000000000)
```

In [2]:
```
help(fact)
```

```
Help on function fact in module __main__:

fact(n)
    returns n!
```

In [3]:
```
fact.__doc__
```

Out[3]:
```
'returns n!\n    '
```

In [4]:
```
type(fact)
```

```
function
```

Out[4]:

In [5]:
```
f = fact # let's take a look at www.pythontutor.com
f
```

Out[5]:
```
<function __main__.fact(n)>
```

In [6]:
```
f(8)
```

Out[6]:
```
40320
```

Lambda Functions

- the **`lambda`** keyword creates an *anonymous* function within a Python expression
- body of **`lambda`** functions limited to pure expressions, i.e.,
  - no assignments
  - no Python statements such as **`while`** , **`try`** , etc.
- best use of **`lambda`** is in the context of an argument list

In [7]:
```
fruits = ['strawberry', 'banana', 'fig', 'apple', 'cherry','kiwi']
fruits
```

Out[7]:
```
['strawberry', 'banana', 'fig', 'apple', 'cherry', 'kiwi']
```

In [8]:
```
def backwards(word):
    return word[::-1]

backwards('supercalifragilisticexpialidocious')
```

Out[8]:
```
'suoicodilaipxecitsiligarfilacrepus'
```

In [9]:
```
sorted(fruits, key=backwards)
```

Out[9]:
```
['banana', 'apple', 'fig', 'kiwi', 'strawberry', 'cherry']
```

In [10]:
```
sorted(fruits, key=lambda word: word[::-1])
```

Out[10]:
```
['banana', 'apple', 'fig', 'kiwi', 'strawberry', 'cherry']
```

In [11]:
```
# how about sorting the list of fruits by the slice
# (no pun intended) which discards the first and last characters,
# e.g., 'anan', 'ppl', etc.

sorted(fruits, key=lambda w: w[1:-1])
```

Out[11]:
```
['banana', 'cherry', 'fig', 'kiwi', 'apple', 'strawberry']
```

map()

- takes a function as its first argument returns an iterable where each item is the result of applying the function to successive elements of the second argument (an iterable)

In [12]:
```
map(fact, range(9))
```

Out[12]:
```
<map at 0x7fd31c2f0c40>
```

In [13]:
```
list(map(fact, range(9)))
```

Out[13]:
```
[1, 1, 2, 6, 24, 120, 720, 5040, 40320]
```

In [14]:
```
# how about mapping '*' to a string?
# or mapping '**' to numbers?
list(map(lambda x: x * 2, 'fiduciary'))
```

Out[14]:
```
['ff', 'ii', 'dd', 'uu', 'cc', 'ii', 'aa', 'rr', 'yy']
```

In [15]:
```
list(map(lambda x: x ** 3, range(1, 10)))
# [x ** 3 for x in range(1, 10)]
```

Out[15]:
```
[1, 8, 27, 64, 125, 216, 343, 512, 729]
```

Higher-Order Functions

- a function that takes another function as an argument or returns a function as a result
  - **`map()`** (as well as **`filter()`** and **`reduce()`**)
  - **`sorted()`** –takes an optional key arg which lets you provide a function which is applied to each item for sorting

In [16]:
```
fruits = ['strawberry', 'banana', 'fig', 'apple', 'cherry', 'kiwi']
sorted(fruits)
```

Out[16]:
```
['apple', 'banana', 'cherry', 'fig', 'kiwi', 'strawberry']
```

In [17]:
```
print(id(len))
sorted(fruits, key=len, reverse=True)
```

Out[17]:
```
140544803338656
['strawberry', 'banana', 'cherry', 'apple', 'kiwi', 'fig']
```

filter

- applies its first arg, a function, to its second argument

In [18]:
```
list(range(6))
```

Out[18]:
```
[0, 1, 2, 3, 4, 5]
```

In [19]:
```
def odd(num):
    return num % 2

list(filter(odd, range(20)))
```

Out[19]:
```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

In [20]:
```
list(filter(lambda num: num % 2, range(20)))
```

Out[20]:
```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

In [21]:
```
# using filter and lambda, pull out all numbers
# divisible by 3 from a list of random numbers
mylist = [33, 35, -3, 20, 6, 9, 20]
list(filter(lambda num: num % 3 == 0, mylist))
```

Out[21]:
```
[33, -3, 6, 9]
```

Lab: filter

- use **`filter()`** to identify all the words in a list which begin with a vowel
- modify your code which displays the last 10 lines of a file using a **`deque`** such that you only display the lines which contain a specific string, e.g., 'salesforce', or match a certain regex pattern, e.g., 'error.*5[012]'

We can further combine functions...

In [22]:
```
list(map(fact, filter(odd, range(12))))
```

Out[22]:
```
[1, 6, 120, 5040, 362880, 39916800]
```

The preceding would normally be done with a list comprehension...

In [23]:
```
[fact(num) for num in range(1, 12, 2)]
```

Out[23]:
```
[1, 6, 120, 5040, 362880, 39916800]
```

...but you may run into stuff like the above in legacy code

reduce()

- produces a single aggregate result from a sequence of any finite iterable object
- was built in to Python 2, but "demoted" to the **`functools`** module in Python 3
- most common use of **`reduce()`**, summation, is better served by the **`sum()`** builtin

- many examples of **`reduce()`** are clearer when written as **`for`** loops

In [24]:
```
from operator import add
help(add)
```

```
Help on built-in function add in module _operator:

add(a, b, /)
    Same as a + b.
```

In [25]:
```
from functools import reduce # no need to import in Python 2
from operator import add
reduce(add, range(101))
```

Out[25]:
```
5050
```

In [26]:
```
sum(range(101))
```

Out[26]:
```
5050
```

In [27]:
```
%%python2
print(range(101))
# range(), xrange()
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 3
2, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 6
2, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 9
2, 93, 94, 95, 96, 97, 98, 99, 100]
```

In [28]:
```
print(range(101))
```

```
range(0, 101)
```

Python's `functools` module

- contains tools which act on *higher-order functions*

If you have a function which needs to remember its results, rather than compute them each time...

In [29]:
```
from functools import lru_cache

@lru_cache(maxsize=None)
def fact(n):
    '''returns n!
    '''
    if n < 2:
        return 1
    else:
        return n * fact(n - 1)

fact_list = [fact(n) for n in range (25)]
fact.cache_info()
```

Out[29]:
```
CacheInfo(hits=23, misses=25, maxsize=None, currsize=25)
```

Or what if you want to *freeze* some of a functions arguments in order to make a simplified version...

In [30]:
```
from functools import partial

basetwo = partial(int, base=2)
basetwo.__doc__ = 'Convert base 2 string to an int.'
basetwo('10010')
```

Out[30]:
```
18
```

In [31]:
```
basetwo.__doc__
```

Out[31]:
```
'Convert base 2 string to an int.'
```

Lab: Partials

- create a **`print_no_nl()`** function which allows you to print something without a trailing newline, without having to specify **`end=''`**

- also make a **`print_no_sp()`** without having to specify **`sep=''`**
- how about a **`sorted_r()`** function for reverse sorting?