Filip Konior sprawozdanie nr4

import functools  
import inspect  
  
print((lambda val: val \*\* 2)(5)) # => 25  
print((lambda x, y: x \* y)(3, 8)) # => 24  
print((lambda s: s.strip().lower()[:2])(' PyTHon')) # => 'py'  
  
#Map  
# ['12', '-2', '0'] => [12, -2, 0]  
print(list(map(int,['12', '-2', '0'] )))  
  
#['hello', 'world'] => [5, 5]  
print(list(map(len,['hello', 'world'] )))  
  
#['hello', 'world'] => ['olleh', 'dlrow']  
print(list(map(lambda x: x[::-1],['hello', 'world'] )))  
  
#range(2, 6) => [(2, 4, 8), (3, 9, 27), (4, 16, 64), (5, 25, 125)]  
print(list(map(lambda x: (x, x\*\*2, x\*\*3),range(2, 6) )))  
  
#zip(range(2, 5), range(3, 9, 2)) => [6, 15, 28]  
print(list(map(lambda args: args[0]\*args[1],zip(range(2, 5), range(3, 9, 2)))))  
  
print(list(map(int, ('10110', '0xCAFE', '42'), (2, 16, 10)))) # generates 22, 51966, 42  
  
#  
print()  
#Filter  
  
#['12', '-2', '0'] => ['12', '0']  
print(list(filter(lambda x: int(x)>=0,['12', '0'])))  
  
#['hello', 'world'] => ['world']  
print(list(filter(lambda x: x.startswith('w'),['hello', 'world'])))  
  
#['Stanford', 'Cal', 'UCLA'] => ['Stanford']  
print(list(filter(lambda x: x.startswith('S'),['Stanford', 'Cal', 'UCLA'])))  
  
#range(20) => [0, 3, 5, 6, 9, 10, 12, 15, 18]  
print(list(filter(lambda x: x%3 == 0 or x%5 == 0,range(20))))  
print()

from functools import reduce  
from math import gcd  
  
def lcm(\*nums):  
 return reduce(lambda x, y: (x\*y)//gcd(x,y), nums)  
  
print(lcm(3,4))  
print(lcm(65,10,5))  
  
#  
print()  
#Module: operator  
import operator  
print(operator.add(1, 2)) # => 3  
print(operator.mul(3, 10)) # => 30  
print(operator.pow(2, 3)) # => 8  
print(operator.itemgetter(1)([1, 2, 3])) # => 2  
  
def fact(n):  
 return 1 if n ==0 else reduce(operator.mul,range(1,n+1))  
  
  
print(fact(3)) # => 6  
print(fact(7)) # => 5040  
print(fact(1)) # => 1  
print(fact(0)) # => 1  
  
#  
print()  
#Custom comparison for sort, max, and min  
words = ['pear', 'cabbage', 'apple', 'bananas']  
print(min(words)) # => 'apple'  
words.sort(key=lambda s: s[-1]) # Alternatively, key=operator.itemgetter(-1)  
print(words) # => ['cabbage', 'apple', 'pear', 'bananas'] ... Why 'cabbage' > 'apple'? Cuz g < l  
print(max(words, key=len)) # 'cabbage' ... Why not 'bananas'?  
print(min(words, key=lambda s: s[1::2])) # What will this value be? bananas cuz aaa < abg

def alpha\_score(upper\_letters):  
  
 return sum(map(lambda l: 1 + ord(l) - ord('A'), upper\_letters))  
  
print(alpha\_score('ABC')) # => 6 = 1 ('A') + 2 ('B') + 3 ('C')  
  
def two\_best(words):  
 words.sort(key=lambda word: alpha\_score(filter(str.isupper, word)), reverse=True)  
 return words[:2]  
  
print(two\_best(['hEllO', 'wOrLD', 'i', 'aM', 'PyThOn']))

def control\_flow(score):  
 if score == 1:  
 return "Winner"  
 elif score == -1:  
 return "Loser"  
 else:  
 return"Tied"  
  
def cf(score):  
 return (score == 1 and "Winner") or (score == -1 and "Loser") or "Tied"  
  
print(control\_flow(1))  
print(cf(-1))

echo = lambda arg: arg # In practice, you should never bind lambdas to local names  
cond\_fn = lambda x: (x==1 and echo("one")) \  
 or (x==2 and echo("two")) \  
 or (echo("other"))  
print(cond\_fn(1))

#Replacing Loops  
# For example:  
#  
# for e in lst:  
# func(e)  
#  
# becomes  
#  
# map(func, lst)  
  
#Replacing Action Sequence  
def f1(x):  
 return x+1  
def f2(x):  
 return x\*x  
def f3(x):  
 return x\*\*3  
  
just\_do\_it = lambda f: f(3)  
  
# Suppose f1, f2, f3 are actions  
print(list(map(just\_do\_it, [f1, f2, f3])))  
  
#  
print()  
#Iterators  
#Iterator Consumption  
  
it = iter(range(100))  
print(67 in it) # => True -> stops at 67  
print(next(it)) # => 68  
print(37 in it) # => False  
#print(next(it)) # StopIteration Exception - there are no further values  
  
#Module: itertools  
import itertools  
import operator  
  
for el in itertools.permutations('XKCD', 2):  
 print(el, end=', ')  
  
# for el in itertools.cycle('LO'):  
# print(el, end='') # Don't run this one. Why not? It wont end  
print()  
print(list(itertools.starmap(operator.mul, itertools.zip\_longest([3,5,7],[2,3], fillvalue=1))))  
  
#(Challenge) Linear Algebra  
#Dot Product  
def dot\_product(u, v):  
 return sum((itertools.starmap(operator.mul, itertools.zip\_longest(u,v, fillvalue=1))))  
  
print(dot\_product([1,3,5],[2,4,6]))  
  
#  
print()  
#Matrix Transposition  
def transpose(m):  
 return tuple(zip(\*m))  
matrix = (  
 (1, 2, 3, 4),  
 (5, 6, 7, 8),  
 (9,10,11,12)  
)  
print(transpose(matrix))  
  
#  
  
print()  
#Matrix Multiplication  
m1 = (  
 (1, 2),  
 (3, 5)  
)  
m2 = (  
 (2, 4),  
 (6, 10)  
)  
  
def matmul(m1, m2):  
 return tuple(map(lambda row: tuple(dot\_product(row, col) for col in transpose(m2)), m1))  
  
print(matmul(m1,m2))  
  
#Generator Expressions  
def generate\_triangles():  
 n = 0  
 total = 0  
 while True:  
 total += n  
 n += 1  
 yield total  
  
def triangles\_under(n):  
 for triangle in generate\_triangles():  
 if triangle >= n:  
 break  
 print(triangle)  
  
triangles\_under(15)  
  
#Functions in Data Structures  
def make\_divisibility\_test(n):  
 return lambda m: m % n == 0  
  
def primes\_under(n):  
 tests = []  
 for i in range(2, n):  
 if not any(map(lambda test: test(i), tests)):  
 tests.append(make\_divisibility\_test(i))  
 yield i  
  
def composite\_gen():  
 tests = []  
 i = 2  
 while True:  
 if not any(map(lambda test: test(i), tests)):  
 tests.append(make\_divisibility\_test(i))  
 else:  
 yield i  
 i+=1  
  
def nth\_composite(n):  
 g = composite\_gen()  
 for i in range(n - 1):  
 next(g)  
 return next(g)  
  
print(list(primes\_under(5)))  
print(nth\_composite(4))  
  
#Nested Functions and Closures  
def outer():  
 def inner(a):  
 return a  
 return inner  
  
f = outer()  
print(f) # <function outer.<locals>.inner at 0x1044b61e0>  
f(10) # => 10  
  
f2 = outer()  
print(f2) # <function outer.<locals>.inner at 0x1044b6268> (Different from above!)  
f2(11) # => 11  
  
#Closure  
def make\_adder(n):  
 def add\_n(m): # Captures the outer variable `n` in a closure  
 return m + n  
 return add\_n  
  
add1 = make\_adder(1)  
print(add1) # <function make\_adder.<locals>.add\_n at 0x103edf8c8>  
print(add1(4)) # => 4  
print(add1(5)) # => 6  
add2 = make\_adder(2)  
print(add2) # <function make\_adder.<locals>.add\_n at 0x103ecbf28>  
print(add2(4)) # => 6  
print(add2(5)) # => 7  
  
add7 = make\_adder(7)  
print(add7(3))  
  
closure = add1.\_\_closure\_\_  
cell0 = closure[0]  
print(cell0.cell\_contents) # => 1 (this is the n = 1 passed into make\_adder)  
  
def foo(a, b, c=-1, \*d, e=-2, f=-3, \*\*g):  
 def wraps():  
 print(a, c, e, g)  
  
w = foo(1, 2, 3, 4, 5, e=6, f=7, y=2, z=3)  
#print(list(map(lambda cell: cell.cell\_contents, w.\_\_closure\_\_)))  
# = > [1, 3, 6, {'y': 2, 'z': 3}]  
  
def outer(l):  
 def inner(n):  
 return l \* n  
 return inner  
  
  
l = [1, 2, 3]  
f = outer(l)  
print(f(3))  
  
l.append(4)  
print(f(3))  
  
print()  
  
#Building Decorators  
  
def debug(function):  
 def wrapper(\*args, \*\*kwargs):  
 print("Arguments:", args, kwargs)  
 return function(\*args, \*\*kwargs)  
 return wrapper  
  
def bind\_args(function, \*args, \*\*kwargs):  
 sig = inspect.Signature.from\_callable(function)  
 ba = sig.bind(\*args, \*\*kwargs)  
 return ba.arguments  
  
def cache(function):  
 function.\_cache = {}  
 @functools.wraps(function)  
 def wrapper(\*args, \*\*kwargs):  
 key = (args, tuple(kwargs.items()))  
 if key in function.\_cache:  
 return function.\_cache[key]  
 retval = function(\*args, \*\*kwargs)  
 function.\_cache[key] = retval  
 return retval  
 return wrapper  
  
@cache  
def fib(n):  
 return fib(n-1) + fib(n-2) if n > 2 else 1  
  
print(fib(10)) # 55 (takes a moment to execute)  
print(fib(10)) # 55 (returns immediately)  
print(fib(100)) # doesn't take forever  
print(fib(400)) # doesn't raise RuntimeError  
  
#  
print()  
#Dynamic Type Checker  
def foo(a: int, b: str) -> bool:  
 return b[a] == 'X'  
  
print(foo.\_\_annotations\_\_) # => {'a': int, 'b': str, 'return': bool}  
  
def enforce\_types(function):  
 expected = function.\_\_annotations\_\_  
 if not expected:  
 return function  
 assert(all(map(lambda exp: type(exp) == type, expected.values())))  
 @functools.wraps(function)  
 def wrapper(\*args, \*\*kwargs):  
 bound\_arguments = bind\_args(function, \*args, \*\*kwargs)  
 for arg, val in bound\_arguments.items():  
 if arg in expected and not isinstance(val, expected[arg]):  
 print("(Bad Argument Type!) argument '{arg}={val}': expected {exp}, received {r}".format(  
 arg=arg,  
 val=val,  
 exp=expected[arg],  
 r=type(val)  
 ))  
  
 retval = function(\*args, \*\*kwargs)  
  
 # Check the return value  
 if 'return' in expected and not isinstance(retval, expected['return']):  
 print("(Bad Return Value!) return '{ret}': expected {exp}, received {r}".format(  
 ret=retval,  
 exp=expected['return'],  
 r=type(retval)  
 ))  
 return retval  
 return wrapper  
  
@enforce\_types  
def foo(a: int, b: str) -> bool:  
 if a == -1:  
 return 'Gotcha!'  
 return b[a] == 'X'  
  
print(foo(3, 'abcXde')) # => True  
print(foo(2, 'python')) # => False  
#print(foo(1, 4)) # prints "Invalid argument type for b: expected str, received int  
# print(foo(-1, '')) # prints "Invalid return type: expected bool, received str