Sumy State University

Lab №3
1. Functional Programming in Python

2. Exercises

2.1 Lambdas

Recall that lambda functions are anonymous, unnamed function objects created on the fly, usually to accomplish a small transformation. For example,

```
(lambda val: val ** 2)(5)  # => 25
(lambda x, y: x * y)(3, 8)  # => 24
(lambda s: s.strip().lower()[:2])('  PyTHon')  # => 'py'
```

On their own, `lambda`s aren't particularly useful, as demonstrated above. Usually, `lambda`s are used to avoid creating a formal function definiton for small throwaway functions, not only because they involves less typing (no `def` or `return` statement needed) but also, and perhaps more imporatntly, because these small functions won't pollute the namespace.
Lambdas are also frequently used as arguments to or return values from higher-order functions, such as `map` and `filter`.

2.2 Map

Recall from class that `map(func, iterable)` applies a function over elements of an iterable. For each of the following rows, write a single statement using `map` that converts the left column into the right column:

From	To
`['12', '-2', '0']`	`[12, -2, 0]`
`['hello', 'world']`	`[5, 5]`
`['hello', 'world']`	`['olleh', 'dlrow']`
`range(2, 6)`	`[(2, 4, 8), (3, 9, 27), (4, 16, 64), (5, 25, 125)]`
`zip(range(2, 5), range(3, 9, 2))`	`[6, 15, 28]`

Hint: you may need to wrap the output in a `list()` constructor to see it printed to console - that is, `list(map(..., ...))`

The `map` function can accept a variable number of iterables as arguments. Thus, `map(func, iterA, iterB, iterC)` is equivalent to `map(func, zip(iterA, iterB, iterC))`. This can be used as follows:

```
map(int, ('10110', '0xCAFE', '42'), (2, 16, 10))  # generates 22, 51966, 42
```

This works because `int` *takes an optional second argument specifying the conversion base.*

2.3 Filter

Recall from class that `filter(pred, iterable)` keeps only those elements from an iterable that satisfy a predicate function.
Write statements using `filter` that convert the following sequences from the left column to the right column:

From	To
`['12', '-2', '0']`	`['12', '0']`
`['hello', 'world']`	`['world']`
`['Stanford', 'Cal', 'UCLA']`	`['Stanford']`
`range(20)`	`[0, 3, 5, 6, 9, 10, 12, 15, 18]`

2.4 Other Useful Tools

Module: ***`functools`***

There is another utility in the `functools` module called `reduce`. This function is well-explained by the official documentation:

`functools.reduce(function, iterable[, initializer])`

Apply `function` of two arguments cumulatively to the items of `iterable`, from left to right, so as to reduce the iterable to a single value. For example, `functools.reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])` calculates `((((1 + 2) + 3) + 4) + 5)`. The left argument, `x`, is the accumulated value and the right argument, `y`, is the update value from the sequence. If the optional `initializer` is present, it is placed before the items of the sequence in the calculation, and serves as a default when the iterable is empty. If `initializer` is not given and `iterable` contains only one item, the first item is returned.

Use the `reduce` function to find the least common multiple of a arbitrary list of arguments. This can be accomplished in one line of Python.

```
import operator
from functools import reduce

def gcd(a, b):
    """Reference implementation of finding the
    greatest common denominator of two numbers"""
    while b != 0:
        a, b = b, a % b
    return a
```

```
def lcm(*args):
    pass
    # Your implementation here: Use reduce and use only one line!
```

.

Module: operator

Frequently, you might find yourself writing anonymous functions similar to `lambda x, y: x + y`. This feels a little redundant, since Python already knows how to add two values together. Unfortunately, we can't just refer to `+` as a function - it's a builtin syntax element. To solve this problem, The `operator` module exports callable functions for each builtin operation. These operators can simplify some common uses of lambdas, and should be used wherever possible.

```
import operator
operator.add(1, 2)  # => 3
operator.mul(3, 10)  # => 30
operator.pow(2, 3)  # => 8
operator.itemgetter(1)([1, 2, 3]) # => 2
```

Take a moment to skim over the official documentation for the `operator` module.

Use `reduce` in conjunction with a function from the `operator` module to compute factorials in one line of Python:

```
import operator
from functools import reduce

def fact(n):
    # Your implementation here: Use reduce, an operator, and only one line!

fact(3)  # => 6
fact(7)  # => 5040
```

Custom comparison for `sort`, `max`, and `min`

Python defaults to ordering sequences by a default ordering. For instances, strings sort alphabetically, and tuples sort lexicographically. Sometimes, however, we need to sort based on a custom key value. In Python, we can supply an optional `key` argument to `sorted(seq)`, `max(seq)`, `min(seq)`, and `seq.sort()` to determine the values used for ordering elements in a sequence. In Python, these default sorting tools are guaranteed to be stable. For example:

```
words = ['pear', 'cabbage', 'apple', 'bananas']
min(words)  # => 'apple'
words.sort(key=lambda s: s[-1])  # Alternatively, key=operator.itemgetter(-1)
words  # => ['cabbage', 'apple', 'pear', 'bananas'] ... Why 'cabbage' > 'apple'?
max(words, key=len)  # 'cabbage' ... Why not 'bananas'?
min(words, key=lambda s: s[1::2])  # What will this value be?
```

Write a function to return the two words with the highest alphanumeric score of uppercase letters:

```
def alpha_score(upper_letters):
    """Computers the alphanumeric sum of letters in a string.
    Prerequisite: upper_letters is composed entirely of capital letters.
    """
    return sum(map(lambda l: 1 + ord(l) - ord('A'), upper_letters))

alpha_score('ABC')  # => 6 = 1 ('A') + 2 ('B') + 3 ('C')
```

```
def two_best(words):
    pass  # Your implementation here

two_best(['hEllO', 'wOrLD', 'i', 'aM', 'PyThOn'])
```

You may want to use `filter` too.

Custom comparison for `sort`, `max`, and `min`

Python defaults to ordering sequences by a default ordering. For instances, strings sort alphabetically, and tuples sort lexicographically. Sometimes, however, we need to sort based on a custom key value. In Python, we can supply an optional `key` argument to `sorted(seq)`, `max(seq)`, `min(seq)`, and `seq.sort()` to determine the values used for ordering elements in a sequence. In Python, these default sorting tools are guaranteed to be stable. For example:

```
words = ['pear', 'cabbage', 'apple', 'bananas']
min(words)  # => 'apple'
words.sort(key=lambda s: s[-1])  # Alternatively, key=operator.itemgetter(-1)
words  # => ['cabbage', 'apple', 'pear', 'bananas'] ... Why 'cabbage' > 'apple'?
max(words, key=len)  # 'cabbage' ... Why not 'bananas'?
min(words, key=lambda s: s[1::2])  # What will this value be?
```

Write a function to return the two words with the highest alphanumeric score of uppercase letters:

```
def alpha_score(upper_letters):
    """Computers the alphanumeric sum of letters in a string.
    Prerequisite: upper_letters is composed entirely of capital letters.
    """
    return sum(map(lambda l: 1 + ord(l) - ord('A'), upper_letters))

alpha_score('ABC')  # => 6 = 1 ('A') + 2 ('B') + 3 ('C')

def two_best(words):
    pass  # Your implementation here

two_best(['hEllO', 'wOrLD', 'i', 'aM', 'PyThOn'])
```

You may want to use `filter` too.

2.6 Purely Functional Programming

As a solely academic thought exercise, let's investigate how we would use Python in a purely functional programming paradigm. Ultimately, we will try to remove statements and replace them with expressions.

Replacing Control Flow

The first thing that needs to go are control flow statements - `if/elif/else`. Luckily, Python, like many other languages, short circuits boolean expressions. This means that we can rewrite

```
if <cond1>:   func1()
elif <cond2>: func2()
else:         func3()
```

as the equivalent expression

```
(<cond1> and func1()) or (<cond2> and func2()) or (func3())
```

Rewrite the following code block without using `if/elif/else`:

```
if score == 1:
    return "Winner"
elif score == -1:
    return "Loser"
else:
    return "Tied"
```

Replacing Returns

However, we would still need return values to do anything useful. Since lambdas implicitly return their expression, we will use lambdas to eliminate return statements. We can bind these temporary conditional conjunctive normal form expressions to a lambda function.

```
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"))
```

Replacing Loops

Getting rid of loops is easy! We can `map` over a sequence instead of looping over the sequence. For example:

```
for e in lst:
    func(e)
```

becomes

```
map(func, lst)
```

Replacing Action Sequence

Most programs take the form a sequence of steps, written out line by line. By using a `just_do_it` function and `map`, we can replicate a sequence of function calls.

```
just_do_it = lambda f: f()

# Suppose f1, f2, f3 are actions
map(just_do_it, [f1, f2, f3])
```

Our main program execution can then be a single call to such a map expression.

Note

In fact, Python has `eval` and `exec` functions builtin. Don't use them! They are dangerous.

.

2.7 Iterators

Recall from class than an iterator is an object that represents a stream of data returned one value at a time.

Iterator Consumption

Suppose the following two lines of code have been run:

```
it = iter(range(100))
67 in it  # => True
```

What is the output of each of the following lines of code?

```
next(it)  # => ??
37 in it  # => ??
next(it)  # => ??
```

With a partner, discuss why we see these results.

Module: itertools

Python ships with a spectacular module for manipulating iterators called `itertools`. Take a moment to read through the documentation page for itertools.
Predict the output of the following pieces of code:

```
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?

itertools.starmap(operator.mul, itertools.zip_longest([3,5,7],[2,3], fillvalue=1))
```

(Challenge) Linear Algebra

These challenge problems test your ability to write compact Python functions using the tools of functional programming and some good old-fashioned cleverness. As always, challenge problems are optional. These challenge problems focus heavily on linear algebra.

Dot Product

Write a one-liner in Python that takes the dot product of two lists `u` and `v`. You can assume that the lists are the same size, and are standard Python lists (not anything special, like numpy arrays). You should use

```
def dot_product(u, v):
    pass
```

For example, `dot_product([1, 3, 5], [2, 4, 6])` should return `44` (since `1 * 2 + 3 * 4 + 5 * 6 = 44`).

Matrix Transposition

Write a one-liner in Python to transpose a matrix. Assume that the input matrix is a tuple-of-tuples that represents a valid matrix, not necessarily square. Again, do not use numpy or any other libraries - just raw data structure manipulation.

```
def transpose(m):
    pass
```

Not only can you do this in one line - you can do it in 14 characters!

For example,

```
matrix = (
    (1, 2, 3, 4),
    (5, 6, 7, 8),
    (9,10,11,12)
)
transpose(matrix)
# returns
# (
#     (1, 5, 9),
#     (2, 6, 10),
#     (3, 7, 11),
#     (4, 8, 12)
# )
```

Matrix Multiplication

Write another one-liner in Python to take the product of two matrices `m1` and `m2`. You can use the `dot_product` and `transpose` functions you already wrote.

```
def matmul(m1, m2):
    pass
```

Lazy Generation

Rewrite your `transpose` and `matmul` functions above so that they are lazily evaluated.

2.8 Generator Expressions

Recall that generator expressions are a way to lazily compute values on the fly, without buffering the entire contents of the list in place.

For each of the following scenarios, discuss whether it would be more appropriate to use a generator expression or a list comprehension:

1. Searching for a given entity in the entries of a 1TB database.
2. Calculate cheap airfare using journey-to-destination flight information.
3. Finding the first palindromic Fibonacci number greater than 1,000,000.

4. Determine all multi-word anagrams of user-supplied 1000-character-or-more strings (very expensive to do).
5. Generate a list of names of Stanford students whose SUNet ID numbers are less than 5000000.
6. Return a list of all startups within 50 miles of Stanford.

Generators

Write a infinite generator that successively yields the triangle numbers `0, 1, 3, 6, 10, ...`

```
def generate_triangles():
    pass  # Your implementation here
```

Use your generator to write a function `triangles_under(n)` that prints out all triangle numbers strictly less than the parameter `n`.

```
def triangles_under(n):
    pass
```

Functions in Data Structures

In class, we quickly showed a highly unusual way to generate primes. Take some time to read through it again, and talk with a partner about how and why this successfully generates prime numbers.

```
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
```

How would you modify the code above to yield all composite numbers, rather than all prime numbers? Test your solution. What is the 1000th composite number?

Nested Functions and Closures

In class, we saw that functions can be defined within the scope of another function (recall from Week 3 that functions introduce new scopes via a new local symbol table). An inner function is only in scope inside of the outer function, so this type of function definition is usually only used when the inner function is being returned to the outside world.

```
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
```

Why are the memory addresses different for `f` and `f2`? Discuss with a partner.

Closure

As we saw above, the definition of the inner function occurs during the execution of the outer function. This implies that a nested function has access to the environment in which it was defined. Therefore, it is possible to return an inner function that remembers the state of the outer function, even after the outer function has completed execution. This model is referred to as a 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>
add1(4)  # => 4
add1(5)  # => 6
add2 = make_adder(2)
print(add2)  # <function make_adder.<locals>.add_n at 0x103ecbf28>
add2(4)  # => 6
add2(5)  # => 7
```

The information in a closure is available in the function's `__closure__` attribute. For example:

```
closure = add1.__closure__
cell0 = closure[0]
cell0.cell_contents  # => 1 (this is the n = 1 passed into make_adder)
```

As another example, consider the function:

```
def foo(a, b, c=-1, *d, e=-2, f=-3, **g):
    def wraps():
        print(a, c, e, g)
```

The `print` call induces a closure of `wraps` over `a`, `c`, `e`, `g` from the enclosing scope of `foo`. Or, you can imagine that wraps "knows" that it will need `a`, `c`, `e`, and `g` from the enclosing scope, so at the time `wraps` is defined, Python takes a "screenshot" of these variables from the enclosing scope and stores references to the underlying objects in the `__closure__`attribute of the `wraps` function.

```
w = foo(1, 2, 3, 4, 5, e=6, f=7, y=2, z=3)
list(map(lambda cell: cell.cell_contents, w.__closure__))
# = > [1, 3, 6, {'y': 2, 'z': 3}]
```

What happens in the following situation? Why?

```
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))  # => ??
```

2.9 Building Decorators

Recall that a decorator is a special type of function that accepts a function as an argument and (usually) returns a modified version of that function. In class, we saw the `debug` decorator - review the slides if you still feel uncomfortable with the idea of a decorator.
Furthermore, recall that the `@decorator` syntax is syntactic sugar.

```
@decorator
def fn():
    pass
```

is equivalent to

```
def fn():
    pass
fn = decorator(fn)
```

print_args

The `debug` decorator we wrote in class isn't very good. It doesn't tell us which function is being called, and it just gives us a tuple of positional arguments and a dictionary of keyword arguments - it doesn't even know what the names of the positional arguments are! If the default arguments aren't overridden, it won't show us their value either.
Use function attributes to improve our `debug` decorator into a `print_args` decorator that is as good as you can make it.

```
def print_args(function):
    def wrapper(*args, **kwargs):
        # (1) You could do something here
        retval = function(*args, **kwargs)
        # (2) You could also do something here
        return retval
    return wrapper
```

Hint: Consider using the attributes `fn.__name__` and `foo.__code__`. You'll have to investigate these attributes, but I will say that the last one is `.__code__` code object which contains a number of useful attributes - for instance, `fn.__code__.co_varnames`. Check it out!

Note

There are a lot of subtleties to this function, since functions can be called in a number of different ways. How does your `print_args` handle keyword arguments or even keyword-only arguments? Variadic positional arguments? Variadic keyword arguments? For more customization, look at `fn.__defaults__`, `fn.__kwdefaults__`, as well as other attributes of `fn.__code__`.

Automatic Caching

Write a decorator `cache` that will automatically cache any calls to the decorated function. You can assume that all arguments passed to the decorated function will always be hashable types.

```
def cache(function):
    pass  # Your implementation here
```

For example:

```
@cache
def fib(n):
    return fib(n-1) + fib(n-2) if n > 2 else 1

fib(10)  # 55 (takes a moment to execute)
fib(10)  # 55 (returns immediately)
fib(100) # doesn't take forever
fib(400) # doesn't raise RuntimeError
```

Hint: You can set arbitrary attributes on a function (e.g. `fn._cache`). When you do so, the attribute-value pair also gets inserted into `fn.__dict__`. Take a look for yourself. Are the extra attributes and `.__dict__` always in sync?

Challenge: Cache Options

Add `max_size` and `eviction_policy` keyword arguments, with reasonable defaults (perhaps `max_size=None` as a sentinel), to your `cache` decorator. `eviction_policy` should be `'LRU'`, `'MRU'`, or `'random'`.

Note

This is actually implemented as part of the language in `functools.lru_cache`

Static Type Checker

Recall that functions in Python can be optionally annotated by semantically-useless but structurally-valuable type annotations. For example:

```
def foo(a: int, b: str) -> bool:
    return b[a] == 'X'

foo.__annotations__  # => {'a': int, 'b': str, 'return': bool}
```

Write a static type checker, implemented as a decorator, that enforces the parameter types and return type of Python objects.

```
def enforce_types(function):
    pass  # Your implementation here
```

For example:

```python
@enforce_types
def foo(a: int, b: str) -> bool:
    if a == -1:
        return 'Gotcha!'
    return b[a] == 'X'

foo(3, 'abcXde')  # => True
foo(2, 'python')  # => False
foo(1, 4)  # prints "Invalid argument type for b: expected str, received int
foo(-1, '')  # prints "Invalid return type: expected bool, received str
```

There are lots of nuances to this function. What happens if some annotations are missing? How are keyword arguments and variadic arguments handled? What happens if the expected type of a parameter is not a primitive type? Can you annotate a function to describe that a parameter should be a list of strings? A tuple of (str, bool) pairs? A dictionary mapping strings to lists of ints?

If you think you're done, show your decorator to us.

Bonus: Optional Debug Argument

Warning! This extension is very hard

Extend the `enforce_types` decorator to accept a keyword argument `severity` which modifies the extent of the enforcement. If `severity == 0`, disable type checking. If `severity == 1` (which is the default), just print a message if there are any type violations. If `severity == 2`, raise an Exception if there are any type violations.
For example:

```python
@enforce_types(severity=2)
def bar(a: list, b: str) -> int:
    return 0

@enforce_types()  # Note that there are parentheses
def baz(a: bool, b: str) -> str:
    return ''
```

APPENDIX

Procedural

```
def a(i):
    if i == 1:
        return "One"
    elif i == 2:
        return "Two"
    else:
        return "Three"

print a(1) # One
print a(2) # Two
print a(3) # Three
```

Functional

```
a = lambda x: x

b = lambda x: (x == 1 and a("One")) \
        or (x == 2 and a("Two")) \
        or (a("Three"))

print b(1) # One
print b(2) # Two
print b(3) # Three
```

Procedural

```
grocery_list = ['apples', 'bananas',
    'oranges', 'milk']

for grocery in grocery_list:
    print grocery
```

Functional

```
grocery_list = ['apples', 'bananas',
    'oranges', 'milk']

def grocery(list):
    print list

map(grocery, grocery_list)
```

Procedural

```
count = 0
while count < 5:
    print 'The count is:', count
    count = count + 1
```

Functional

```
def func(n):
    print 'The count is:', n

a = lambda count: [func(i) for i in range(count)]
a(5)
```

Procedural	Functional
def f(x): return x*x a = f(4) print a	a = lambda x: x * x print a(4)