Functional Programming

Functional programming is a style of programming that (as the name suggests) is based around functions.
A key part of functional programming is **higher-order functions**. We have seen this idea briefly in the previous lesson on functions as objects. Higher-order functions take other functions as arguments, or return them as results.
Example:

```
def apply_twice(func, arg):
  return func(func(arg))
def add_five(x):
  return x + 5
print(apply_twice(add_five, 10))
```

Result:

```
>>>
20
>>>
```

The function **apply_twice** takes another function as its argument, and calls it twice inside its body.

Ex:

What is the output of this code?

```
def test(func, arg):
 return func(func(arg))

def mult(x):
 return x * x

print(test(mult, 2))
```

Ans: 16

Pure Functions

Functional programming seeks to use **pure functions**. Pure functions have no side effects, and return a value that depends **only** on their arguments.
This is how functions in math work: for example, The cos(x) will, for the same value of x, always return the same result.
Below are examples of pure and impure functions.
Pure function:

```
def pure_function(x, y):
 temp = x + 2*y
 return temp / (2*x + y)
```

Impure function:

```
some_list = []

def impure(arg):
 some_list.append(arg)
```

The function above is not pure, because it changed the state of **some_list**.

Pure Functions

Using pure functions has both advantages and disadvantages.
Pure functions are:
- easier to reason about and test.
- more efficient. Once the function has been evaluated for an input, the result can be stored and referred to the next time the function of that input is needed, reducing the number of times the function is called. This is called **memoization**.
- easier to run in parallel.

The main disadvantage of using only pure functions is that they majorly complicate the otherwise simple task of I/O, since this appears to inherently require side effects.

They can also be more difficult to write in some situations.

Lambdas

Creating a function normally (using **def**) assigns it to a variable automatically.
This is different from the creation of other objects - such as strings and integers - which can be created on the fly, without assigning them to a variable.
The same is possible with functions, provided that they are created using **lambda** syntax. Functions created this way are known as **anonymous**.
This approach is most commonly used when passing a simple function as an argument to another function. The syntax is shown in the next example and consists of the **lambda** keyword followed by a list of arguments, a colon, and the expression to evaluate and return.

```
def my_func(f, arg):
 return f(arg)


my_func(lambda x: 2*x*x, 5)
```

Lambda functions get their name from **lambda calculus**, which is a model of computation invented by Alonzo Church.

Lambdas

Lambda functions aren't as powerful as named functions.
They can only do things that require a single expression - usually equivalent to a single line of code.
Example:

```
#named function

def polynomial(x):

  return x**2 + 5*x + 4

print(polynomial(-4))

#lambda

print((lambda x: x**2 + 5*x + 4) (-4))
```

Ans: >>>
0
0
>>>

Lambdas

Lambda functions can be assigned to variables, and used like normal functions.
Example:

```
double = lambda x: x * 2

print(double(7))

Ans:

>>>

14

>>>
```

However, there is rarely a good reason to do this - it is usually better to define a function with **def** instead.

map

The built-in functions **map** and **filter** are very useful higher-order functions that operate on lists (or similar objects called **iterables**).
The function **map** takes a function and an iterable as arguments, and returns a new iterable with the function applied to each argument.
Example:

```
def add_five(x):

 return x + 5

nums = [11, 22, 33, 44, 55]

result = list(map(add_five, nums))

print(result)
```

Result:

```
>>>

[16, 27, 38, 49, 60]

>>>
```

We could have achieved the same result more easily by using **lambda** syntax.

```
nums = [11, 22, 33, 44, 55]

result = list(map(lambda x: x+5, nums))

print(result)
```

To convert the result into a list, we used **list** explicitly.

filter

The function **filter** filters an iterable by removing items that don't match a predicate (a function that returns a Boolean).
Example:

```
nums = [11, 22, 33, 44, 55]

res = list(filter(lambda x: x%2==0, nums))

print(res)
```

Result:

```
>>>
[22, 44]
>>>
```

Like **map**, the result has to be explicitly converted to a list if you want to print it.

Generators

Generators are a type of iterable, like lists or tuples.
Unlike lists, they don't allow indexing with arbitrary indices, but they can still be iterated through with **for** loops.
They can be created using functions and the **yield** statement.
Example:

```
def countdown():

 i=5

 while i > 0

   yield i
```

```
  i -= 1

for i in countdown():

 print(i)
```

Result:

```
>>>

5

4

3

2

1
```

The ***yield*** statement is used to define a generator, replacing the return of a function to provide a result to its caller without destroying local variables.

Due to the fact that they **yield** one item at a time, generators don't have the memory restrictions of lists. In fact, they can be **infinite!**

```
def infinite_sevens():

 while True:

  yield 7

for i in infinite_sevens():

 print(i)
```

Result:

```
>>>

7

7

7

7

7

7

7

...
```

In short, **generators** allow you to declare a function that behaves like an iterator, i.e. it can be used in a **for** loop.

Fill in the blanks to create a prime number generator, that yields all prime numbers in a loop. (Consider having an is_prime function already defined):

```
get_primes():
 num = 2
 while True:
  if is_prime(num):
  yield num
  num += 1
```

Finite generators can be converted into lists by passing them as arguments to the **list** function.

```
def numbers(x):
 for i in range(x):
  if i % 2 == 0:
   yield i
print(list(numbers(11)))
```

Result:

```
>>>
[0, 2, 4, 6, 8, 10]
>>>
```

Using **generators** results in improved performance, which is the result of the lazy (on demand) generation of values, which translates to lower memory usage. Furthermore, we do not need to wait until all the elements have been generated before we start to use them.

Example:
What is the result of this code?

```
def make_word():
 word = ""
 for ch in "spam":
```

```
  word +=ch
  yield word

print(list(make_word()))
```

Ans: ['s', 'sp', 'spa', 'spam']

Decorators

Decorators provide a way to modify functions using other functions.
This is ideal when you need to extend the functionality of functions that you don't want to modify.
Example:

```
def decor(func):
 def wrap():
  print("============")
  func()
  print("============")
 return wrap
def print_text():
 print("Hello world!")
decorated = decor(print_text)
decorated()
```

We defined a function named **decor** that has a single parameter **func**. Inside **decor**, we defined a nested function named **wrap**. The **wrap** function will print a string, then call **func()**, and print another string. The **decor** function returns the **wrap** function as its result.
We could say that the variable **decorated** is a decorated version of **print_text** - it's **print_text** plus something.
In fact, if we wrote a useful decorator we might want to replace **print_text** with the decorated version altogether so we always got our "plus something" version of **print_text**.
This is done by re-assigning the variable that contains our function:

```
print_text = decor(print_text)
```

```
print_text()
```

Now **print_text** corresponds to our decorated version.

In our previous example, we decorated our function by replacing the variable containing the function with a wrapped version.

```
def print_text():
 print("Hello world!")
print_text = decor(print_text)
```

This pattern can be used at any time, to wrap any function.
Python provides support to wrap a function in a decorator by pre-pending the function definition with a decorator name and the @ symbol.
If we are defining a function we can "decorate" it with the @ symbol like:

```
@decor
def print_text():
 print("Hello world!")
```

This will have the same result as the above code.

A single function can have multiple decorators.

Recursion

Recursion is a very important concept in functional programming.
The fundamental part of recursion is self-reference - functions calling themselves. It is used to solve problems that can be broken up into easier sub-problems of the same type.

A classic example of a function that is implemented recursively is the **factorial** function, which finds the product of all positive integers below a specified number.
For example, 5! (5 factorial) is 5 * 4 * 3 * 2 * 1 (120). To implement this recursively, notice that 5! = 5 * 4!, 4! = 4 * 3!, 3! = 3 * 2!, and so on. Generally, n! = n * (n-1)!.
Furthermore, 1! = 1. This is known as the **base case**, as it can be calculated without performing any more factorials.
Below is a recursive implementation of the factorial function.

```
def factorial(x):
 if x == 1:
```

```
  return 1
 else:
  return x * factorial(x-1)
print(factorial(5))
```

Result:

```
>>>
120
>>>
```

The **base case** acts as the exit condition of the recursion.

What is the base case of a recursive function?

Ans: A case that doesn't involve any further calls to that function

Recursive functions can be infinite, just like infinite **while** loops. These often occur when you forget to implement the base case.

Below is an incorrect version of the factorial function. It has no base case, so it runs until the interpreter runs out of memory and crashes.

```
def factorial(x):
 return x * factorial(x-1)
print(factorial(5))
```

Result:

```
>>>
RuntimeError: maximum recursion depth exceeded
>>>
```

Recursion

Recursion can also be indirect. One function can call a second, which calls the first, which calls the second, and so on. This can occur with any number of functions.
Example:

```
def is_even(x):
```

```
if x == 0:
  return True
else:
  return is_odd(x-1)
def is_odd(x):
  return not is_even(x)
print(is_odd(17))
print(is_even(23))
```

Result:

```
>>>
True
False
>>>
```

Example:

```
def fib(x):
 if x == 0 or x == 1:
   return 1
 else:
   return fib(x-1) + fib(x-2)

print(fib(4))
```

Ans: 5

Sets

Sets are data structures, similar to lists or dictionaries. They are created using curly braces, or the **set** function. They share some functionality with lists, such as the use of **in** to check whether they contain a particular item.

```
num_set = {1, 2, 3, 4, 5}
```

```
word_set = set(["spam", "eggs", "sausage"])

print(3 in num_set)

print("spam" not in word_set)
```

Result:

```
>>>
True
False
>>>
```

To create an empty set, you must use **set()**, as **{}** creates an empty dictionary.

Sets differ from lists in several ways, but share several list operations such as **len**.
They are unordered, which means that they can't be indexed.
They **cannot** contain duplicate elements.
Due to the way they're stored, it's **faster** to check whether an item is part of a set, rather than part of a list.
Instead of using **append** to add to a set, use **add**.
The method **remove** removes a specific element from a set; **pop** removes an arbitrary element.

```
nums = {1, 2, 1, 3, 1, 4, 5, 6}

print(nums)

nums.add(-7)

nums.remove(3)

print(nums)
```

Result:

```
>>>
{1, 2, 3, 4, 5, 6}
{1, 2, 4, 5, 6, -7}
>>>
```

Basic uses of **sets** include membership testing and the elimination of duplicate entries.

Sets can be combined using mathematical operations.
The **union** operator **|** combines two sets to form a new one containing items in either.
The **intersection** operator **&** gets items only in both.
The **difference** operator **-** gets items in the first set but not in the second.
The **symmetric difference** operator **^** gets items in either set, but not both.

```
first = {1, 2, 3, 4, 5, 6}

second = {4, 5, 6, 7, 8, 9}

print(first | second)

print(first & second)

print(first - second)

print(second - first)

print(first ^ second)
```

Result:

```
>>>
{1, 2, 3, 4, 5, 6, 7, 8, 9}
{4, 5, 6}
{1, 2, 3}
{8, 9, 7}
{1, 2, 3, 7, 8, 9}
>>>
```

Data Structures

As we have seen in the previous lessons, Python supports the following data structures: **lists**, **dictionaries**, **tuples**, **sets**.

When to use a dictionary:
- When you need a logical association between a **key:value** pair.
- When you need fast lookup for your data, based on a custom key.
- When your data is being constantly modified. Remember, dictionaries are mutable.

When to use the other types:
- Use **lists** if you have a collection of data that does not need random access. Try to choose lists when you need a simple, iterable collection that is modified frequently.
- Use a **set** if you need uniqueness for the elements.
- Use **tuples** when your data cannot change.

Many times, a **tuple** is used in combination with a **dictionary**, for example, a **tuple** might represent a key, because it's immutable.

itertools

The module **itertools** is a standard library that contains several functions that are useful in functional programming.
One type of function it produces is infinite iterators.
The function **count** counts up infinitely from a value.
The function **cycle** infinitely iterates through an iterable (for instance a list or string).
The function **repeat** repeats an object, either infinitely or a specific number of times.
Example:

```
from itertools import count

for i in count(3):

 print(i)

 if i >=11:

   break
```

Result:

```
>>>

3

4

5

6

7

8

9

10

11

>>>
```

There are many functions in **itertools** that operate on iterables, in a similar way to **map** and **filter**.
Some examples:
takewhile - takes items from an iterable while a predicate function remains true;
chain - combines several iterables into one long one;
accumulate - returns a running total of values in an iterable.

```
from itertools import accumulate, takewhile
```

```
nums = list(accumulate(range(8)))

print(nums)

print(list(takewhile(lambda x: x<= 6, nums)))
```

Result:

```
>>>
[0, 1, 3, 6, 10, 15, 21, 28]
[0, 1, 3, 6]
>>>
```

There are also several combinatoric functions in **itertool**, such as **product** and **permutation**.
These are used when you want to accomplish a task with all possible combinations of some items.
Example:

```
from itertools import product, permutations

letters = ("A", "B")

print(list(product(letters, range(2))))

print(list(permutations(letters)))
```

Result:

```
>>>
[('A', 0), ('A', 1), ('B', 0), ('B', 1)]
[('A', 'B'), ('B', 'A')]
>>>
```

Example:

```
from itertools import product
a={1, 2}
print(len(list(product(range(3), a))))
```

Ans: 6

What is the result of this code?

```
def power(x, y):
 if y == 0:
```

```
    return 1
  else:
    return x * power(x, y-1)

print(power(2, 3))
Ans: 8
```