Functional Programming

Functional Programming is Declarative Review

Functional programming is a programming paradigm that adheres to the *declarative* style of programming.

Declarative Programming Review Card

In the declarative style of programming, the programmer describes *what* must be done as opposed to *how* it must be done.

```
nums = [9, 6, 5, 2, 3]

nums.sort() # Sorting the list declaratively using the
sorting algorithm provided by the Python langauge

def custom_sort(list):
  # Custom sorting algorithm defined here

custom_sort(nums) # Sorting the list non-declarativley by
using a custom built sorting algorithm
```

Functions Should Have no Side Effects

In functional programming, a function is expected to be “pure”, meaning it should have no side effects!
A side effect is when a function alters the state of an external variable.
A function can read an external variable, but it should not change it!

```
nums = [1, 3, 5, 9]

# This function has side effects because it alters the nums list!
def square1():
  for i in range(len(nums)):
    nums[i] = nums[i]**2

# This function does not have side effects because it does not alter the nums list!
def square2(lst):
  new_list = []
  for i in lst:
    new_list.append(i)
  return i

# Note: squre2 should loop using recursion. The for-loop is used instead for simplicity!
```

Using namedtuple

When storing data that contains multiple properties, using a `namedtuple` is more efficient than using a regular tuple. It allows you to store and reference the properties of a data entry by their name.
The accompanying code shows a data entry for a student which contains information about the student's age, eye color, and gender.

```
from collections import namedtuple as namedtuple

# Record for student Peter stored in a regular tuple
peter = (16, blue, "male") # This is error-prone because you are forced to remember what each entry means.

student = namedtuple("student", ["age", "eye_color", "gender"])

peter = student(16, "blue", "male")

# This is more efficient and less error-prone as the student's data can be accssed like so: peter.age, peter, eye_color, peter.gender
```

Lazy Iteration

We use lazy iteration in functional programming to be more efficient with memory. With lazy iteration, the iterator is triggered only when the next value is needed.
In the example given, `evens` is intended to be a collection of the even numbers in `nums`.
The next even number in `evens` will be obtained by calling `next(evens)` and should only be done when the next even number is needed!

```
nums = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

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

print(evens) # This will not output a tuple of the even numbers in nums because the iterator has not yet been triggered!

print(next(evens)) # This will output 2
```

Using Higher-order Functions Together

The higher-order functions `map()`, `filter()` and `reduce()` can be used together to execute a task that would otherwise require many loops neatly.

```
from functools import reduce

nums = (2, 4, 6, 8, 10, 12, 14, 16)

# The following adds 1 to all numbers greater than 8 (in nums) and sums them all up

sum = reduce(lambda x, y: x + y, map(lambda x: x+1, filter(lambda x: x > 8, nums)))
```

Working With Large Data Sets

Functional programming is widely used to process data stored in CSV files or JSON files. Since the files could contain a large amount of data, lazy iteration plays an essential role. Data is imported when needed instead of occupying too much memory by loading it all at once.

Storing Data From CSV Files in a namedtuple

We can represent data records stored in CSV files using a `namedtuple`. In the code block shown, the `map()` function is used to read in a record of data and represent it as a `namedtuple`.

```
import csv
from collections import namedtuple
from functools import reduce

tree = namedtuple("tree", ["index", "girth", "height",
"volume"])

with open('trees.csv', newline = '') as csvfile:
    reader = csv.reader(csvfile, delimiter=',',
quotechar='|')
    fields = next(reader)

    # This will return an iterator that will be triggerd when
the next tree is needed!
    trees = map(lambda x: tree(x[0], x[1], x[2], x[3]),
reader)
```