COMP1531

6.2 Pythonic Code

Being Pythonic

Being "Pythonic" means that your code generally follows a set of idioms agreed upon by the broader python community.

"When a veteran Python developer calls portions of code not “Pythonic”, they usually mean that these lines of code do not follow the common guidelines and fail to express its intent in what is considered the most readable way. On some border cases, no best way has been agreed upon on how to express an intent in Python code, but these cases are rare."

Hitchhiker's guide to python (read more on this)

Examples

- Docstrings
- Map, reduce, filter, lambdas
- Exceptions > Early returns
- Destructuring, ignored variables
- Enumerate
- Multi line strings

Docstrings

Docstrings are an important way to document code and make clear to other programmers the intent and meaning behind what you're writing. We are somewhat different on the formatting, but we want it to include 1) Description, 2) Parameters, 3) Returns

docstring.py

```
1  def string_find(str1, str2):
2      """ Returns whether str2 can be found within str1
3
4      Parameters:
5          str1 (str): The haystack
6          str2 (str): The needle
7
8      Returns:
9          (bool): Whether or not str2 could be found in str1
10
11     """
```

Map, Reduce, Filter

- **Map**: creates a new list with the results of calling a provided function on every element in the given list

- **Reduce**: executes a **reducer** function (that you provide) on each member of the array resulting in a single output value

- **Filter**: creates a new array with all elements that pass the test implemented by the provided function

Map

Map: creates a new array with the results of calling a provided function on every element in the calling array

map.py

```
1 def shout(string):
2     return string.upper() + "!!!!"
3
4 if __name__ == '__main__':
5     tutors = ['Simon', 'Teresa', 'Kaiqi', 'Michelle']
6     angry_tutors = list(map(shout, tutors))
7     print(angry_tutors)
```

Reduce

Reduce: executes a **reducer** function (that you provide) on each member of the array resulting in a single output value

reduce.py

```
1 from functools import reduce
2
3 def custom_sum(first, second):
4     return first + second
5
6 if __name__ == '__main__':
7     studentMarks = [ 55, 43, 34, 23, 22, 10, 44 ]
8     total = reduce(lambda a, b: a + b, studentMarks)
9     print(total)
```

Filter

Filter: creates a new array with all elements that pass the test implemented by the provided function

filter.py

```
1 from functools import reduce
2
3 if __name__ == '__main__':
4     marks = [ 65, 72, 81, 40, 56 ]
5     passing_marks = list(filter(lambda m: m >= 50, marks))
6     total = reduce(lambda a, b: a + b, passing_marks)
7     average = total/len(passing_marks)
8     print(average)
```

Combined

allthree.py

```
1 from functools import reduce
2
3 if __name__ == '__main__':
4     marks = [ 39, 43.2, 48.6, 24, 33.6 ] # Marks out of 60
5     normalised_marks = map(lambda m: 100*m/60, marks)
6     passing_marks = list(filter(lambda m: m >= 50, normalised_marks))
7     total = reduce(lambda a, b: a + b, passing_marks)
8     average = total/len(passing_marks)
9     print(average)
```

Exceptions > Early Returns

You might be quite familiar with early returns:

early1.py

```
1 def sqrt(num):
2     if num < 0:
3         return None
4     return num ** 0.5
5
6 myNum = int(input())
7 if sqrt(myNum) is not None:
8     print(sqrt(myNum))
```

The problems though are:

- Often we can only use "None" or some arbitrary return (-1) to signify that it didn't work
- It's harder to check for a client using it

Exceptions > Early Returns

So we use exceptions. And we can make our own.

early.py

```
1  class SqrtException(Exception):
2      pass
3
4  def sqrt(num):
5      if num < 0:
6          raise SqrtException("Number cannot be < 0")
7      return num ** 0.5
8
9  try:
10     print(sqrt(int(input())))
11 except SqrtException as e:
12     print(e)
```

Destructuring

Being able to make tuples and destructure them is very powerful. If you don't want all tuples you can use blanks to ignore them.

destructure.py

```
1  import math
2
3  def convert(x, y):
4      return (math.sqrt(x**2 + y**2), math.degrees(math.atan2(y,x)))
5
6  if __name__ == '__main__':
7      print("Enter x coord: ", end='')
8      x = int(input())
9      print("Enter y coord: ", end='')
10     y = int(input())
11
12     mag, dir = convert(x, y)
13     print(mag, dir)
14
15     mag2, _ = convert(x, y)
16     print(mag2)
```

Enumerate

Sometimes we want to iterate cleanly over the values in a list, but also know what index we're up to. In these situations the enumerate built-in is useful.

enumerate.py

```
1 tutors = ['Vivian', 'Rob', 'Rudra', 'Michelle']
2
3 for idx, name in enumerate(tutors):
4     print(f"{idx + 1}: {name}")
```

Multi-line strings

Someones strings need to exist over multiple lines, there are two good approaches for this

multiline.py

```
1  if __name__ == '__main__':
2      text1 = """hi
3
4      this has lots of space
5
6      between chunks"""
7
8      text2 = (
9          "This is how you can break strings "
10         "into multiple lines "
11         "without needing to combine them manually"
12     )
13
14     print(text1)
15     print(text2)
```