Functional Programming in Python

Functional Features in Python

Functions are first class, meaning they can be

- stored in variables and data structures
- passed as arguments to functions
- returned from functions

Higher-Order Functions

A higher order function is a function that takes another function as a parameter or returns a function as a value. We've already used one:

```
>>> help(sorted)
...
sorted(iterable, key=None, reverse=False)
    Return a new list containing all items from the iterable in ascending
    order.

    A custom key function can be supplied to customise the sort order, and the
    reverse flag can be set to request the result in descending order.
```

The second parameter, `key`, is a function. In general, a *sort key* is the part of an object on which comparisons are made in a sorting algorithm.

Sorting without a `key`

Say we have a list of tuples, *(name, gpa, major)*:

```
>>> from pprint import pprint
>>> studs = [("Stan", 2.5, "ISyE"), ("Kyle", 2.2, "CS"),
...          ("Cartman", 2.4, "CmpE"), ("Kenny", 4.0, "ME")]
```

The default sort order is simply elementwise by the default order for each type in the tuple:

```
>>> pprint(sorted(studs))
[('Cartman', 2.4, 'CmpE'),
 ('Kenny', 4.0, 'ME'),
 ('Kyle', 2.2, 'CS'),
 ('Stan', 2.5, 'ISyE')]
```

Active Review

- What if two students had the same name?

Sorting with a `key`

If we want a different sort order, we can define a function that extracts the part of a tuple by which we want to sort.

```
>>> def by_gpa(stud):
...     return stud[1]
...
>>> pprint(sorted(studs, key=by_gpa))
[('Kyle', 2.2, 'CS'),
 ('Cartman', 2.4, 'CmpE'),
 ('Stan', 2.5, 'ISyE'),
 ('Kenny', 4.0, 'ME')]
```

`sorted` is a *higher-order function* because it takes a function as an argument.

Active Review

- Write a function that sorts students by major, then GPA, then name.

Lambda Functions

The by_gpa function is pretty simple. Instead of defining a named function, we can define it inline with an anonymous function, a.k.a., a *lambda function*:

```
>>> pprint(sorted(studs, key=lambda t: t[1]))
[('Kyle', 2.2, 'CS'),
 ('Cartman', 2.4, 'CmpE'),
 ('Stan', 2.5, 'ISyE'),
 ('Kenny', 4.0, 'ME')]
```

The general form is lambda <parameter_list>: <expression>

The body of a lambda function is limited to a single expression, which is implicitly returned.

map

Common task: build a sequence out of transformations of elements of an existing sequence. Here's the imperative approach:

```
>>> houses = ["Stark", "Lannister", "Targaryen"]
>>> shout = []
>>> for house in houses:
...     shout.append(house.upper())
...
>>> shout
['STARK', 'LANNISTER', 'TARGARYEN']
```

Heres' the functional approach:

```
>>> list(map(lambda house: house.upper(), houses))
['STARK', 'LANNISTER', 'TARGARYEN']
```

`map` returns an iterator, which we pass to the `list` constructor to create a list.

filter

```
>>> nums = [0,1,2,3,4,5,6,7,8,9]
>>> filter(lambda x: x % 2 == 0, nums)
<filter object at 0x1013e87f0>
>>> list(filter(lambda x: x % 2 == 0, nums))
[0, 2, 4, 6, 8]
```

List Comprehensions

A list comprehension iterates over a (optionally filtered) sequence, applies an operation to each element, and collects the results of these operations in a new list, just like `map`.

```
>>> grades = [100, 90, 0, 80]
>>> [x for x in grades]
[100, 90, 0, 80]
>>> [x + 10 for x in grades]
[110, 100, 10, 90]
```

We can also filter in a comprehension:

```
>>> [x + 50 for x in grades if x < 50]
[50]
```

Comprehensions are more Pythonic than using `map` and `filter` directly.

Active Review

- Write a list comprehension that returns the perfect squares from a list of numbers.

Dictionary Comprehensions

First, zip:

```
words = ["Winter is coming", "Hear me roar", "Fire and blood"]
>>> list(zip(houses, words))
[('Stark', 'Winter is coming'), ('Lannister', 'Hear me roar'), ('Targaryen', 'Fire and blood')]
```

Dictionary comprehension using tuple unpacking:

```
>>> house2words = {house: words for house, words in zip(houses, words)}
>>> house2words
{'Lannister': 'Hear me roar', 'Stark': 'Winter is coming', 'Targaryen': 'Fire and blood'}
```

Of course, we could just use the `dict` constructor on the `zip` object.

```
>>> dict(zip(houses, words))
{'Lannister': 'Hear me roar', 'Stark': 'Winter is coming', 'Targaryen': 'Fire and blood'}
```

reduce

```
>>> import functools
>>> functools.reduce(lambda x, y: x + y, [0,1,2,3,4,5,6,7,8,9])
45
```

Confirm this using the standard sum $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Active Review

- Write the factorial function using `reduce`.
  - `factorial(0)== 1`, and
  - for $n > 0, n \in \mathbf{Z}$, `factorial(n)=`

$$\prod_{i=1}^{n} i$$

Generator Functions

Generator functions are an easy functional way to create iterators.

```
def myrange(start: int, end: int) -> int:
    while start < end:
        yield start
        start += 1
```

```
>>> for i in myrange(0, 4):
...     print(i)
...
0
1
2
3
```

Active Review

- Modify the `myrange` generator function above to include a step just like Python's built-in `range` object.

Conclusion

- Because functions are first-class objects in Python, programming in a functional style is possible.
- Remember from the functions lesson that Python does not do tail-call optimization and therefore is not suitable for general purely functional programming.
- Python provides the more useful and ergonomic functional features, like map, filter, and reduce.
- Favor comprehension expressions over using `map` and `filter` directly.
- Simple loop-based transformations should be done with comprehension expressions, but more complex transformations can result in hard-to-read comprehension expressions – always favor readability!