

# 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:

```
1 >>> help(sorted)
2 ...
3 sorted(iterable, key=None, reverse=False)
4     Return a new list containing all items from the iterable in ascending
5     order.
6
7     A custom key function can be supplied to customise the sort order, and the
8     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)*:

```
1 >>> from pprint import pprint
2 >>> studs = [("Stan", 2.5, "ISyE"), ("Kyle", 2.2, "CS"),
3 ...         ("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:

```
1 >>> pprint(sorted(studs))
2 [('Cartman', 2.4, 'CmpE'),
3  ('Kenny', 4.0, 'ME'),
4  ('Kyle', 2.2, 'CS'),
5  ('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.

```
1 >>> def by_gpa(stud):
2 ...     return stud[1]
3 ...
4 >>> pprint(sorted(studs, key=by_gpa))
5 [('Kyle', 2.2, 'CS'),
6  ('Cartman', 2.4, 'CompE'),
7  ('Stan', 2.5, 'ISyE'),
8  ('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*:

```
1 >>> pprint(sorted(studs, key=lambda t: t[1]))
2 [('Kyle', 2.2, 'CS'),
3  ('Cartman', 2.4, 'CmpE'),
4  ('Stan', 2.5, 'ISyE'),
5  ('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:

```
1 >>> houses = ["Stark", "Lannister", "Targaryen"]
2 >>> shout = []
3 >>> for house in houses:
4 ...     shout.append(house.upper())
5 ...
6 >>> shout
7 ['STARK', 'LANNISTER', 'TARGARYEN']
```

Heres' the functional approach:

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

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

## filter

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

```
1 >>> grades = [100, 90, 0, 80]
2 >>> [x for x in grades]
3 [100, 90, 0, 80]
4 >>> [x + 10 for x in grades]
5 [110, 100, 10, 90]
```

We can also filter in a comprehension:

```
1 >>> [x + 50 for x in grades if x < 50]
2 [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:

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

Dictionary comprehension using tuple unpacking:

```
1 >>> house2words = {house: words for house, words in zip(houses, words)}
2 >>> house2words
3 {'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.

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

## reduce

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

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

```
1 >>> for i in myrange(0, 4):  
2     ...     print(i)  
3     ...  
4     0  
5     1  
6     2  
7     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!