8-0: Functional Programming

- Functional programming focuses on the development of functions
  - Black boxes that can take inputs and return an output
  - No alteration of inputs
  - No side effects
- Functions are “first class” objects
- Purely functional languages:
  - Lisp, Scheme, ML
  - Python has a great deal of support for functional programming

8-1: Imperative Languages

- Recall that C, Pascal, Java are what’s called imperative languages
  - Load values into variables
  - Compute a result
  - Store the result in a new variable
- Language closely mirrors the way that CPUs are constructed.

8-2: Functional Programming

- Functional programming languages abstract away from the underlying architecture.
- Focus is on functions (in the mathematical sense) that map elements of a domain onto elements of a range.
- Concern is with expressions; complex operations are built by composing simpler functions.

8-3: Example: Reversing a list

- Imperative approach:
  ```python
  def reverse (l) :
      temp = []
      for item in l :
          temp = [item] + temp
      return temp
  ```
  Emphasis is on building up a result and returning it

- Functional approach:
  ```python
  def reverse (l) :
      if l :
          return reverse(l[1:]) + [l[0]]
      else :
          return []
  ```
  No assignment statements
  Problem is solved via divide-and-conquer
**8-5: Operating on lists**

- Functional programming often works by applying a function to a sequence of objects.
- This is sometimes called applicative programming.
- For example, let’s take a list of words and convert them all to lower case.

**8-6: Converting a list of words**

- Imperative style
  ```python
  def lowercase(wordlist):
      returnlist = []
      for item in wordlist:
          returnlist.append(item.lower())
  return returnlist
  ```
- There’s nothing wrong with this, but we have to explicitly manage how the new list is built up.
- The transformation that’s occurring is ‘buried’ inside the function.

**8-7: Converting a list of words**

- Functional style:
  ```python
  def lowercase(word):
      return word.lower()
  map(lowercase, wordlist)
  ```
- map repeatedly applies a function to a list.
- Returns the result of applying this function to everything in a list.

**8-8: Mapping multiple sequences**

- Map can also work with multiple sequences.
- Number of sequences must match number of function args.
  ```python
  def average(x, y):
      return x + y
  map(average, [1, 2, 3, 4], [2, 3, 4, 5])
  ```
- If sequences are of different length, your function should use default arguments.

**8-9: Filtering a sequence**

- filter can be used to find all elements of a sequence that pass a test.
- Design a function that returns true or false.
  ```python
  def isEven(x):
      return x % 2 == 0
  filter(isEven, [1, 2, 3, 4, 5])
  ```
- filter can also be used to test if any member of a sequence passes some test. (how?)

**8-10: Reducing a list**

- A common programming task is to apply some function repeatedly to a list of data to compute summary information.
- For example, average a bit list of numbers.
- Imperative approach:
  ```python
  nlist = [1, 2, 3, 4, 5, 6]
  ave = 0
  for item in nlist:
      ave += item
  ave = ave / len(nlist)
  ```
- Note that we have to manage the accumulator ourselves.
8-11: Reducing a list

- Functional approach:
  - Reduce applies a function repeatedly to a list and returns a scalar.
  ```python
nlist = [1,2,3,4,5,6]
def add(x,y):
    return x + y
reduce(add, nlist) / len(nlist)
```
  - We can let Python manage the iteration and accumulation.

8-12: Functions as first-class objects

- What is allowing us to do this?
  - Python allows functions to be treated as *first-class objects*
    - A function can be assigned to a variable
    - A function can be provided as input to another function
    - A function can return another function

8-13: Returning a function from a function

- A simple example:
  ```python
def add(x,y):
    return x + y
def sub(x,y):
    return x - y
def mult(x,y):
    return x * y

def get_op(input):
    if input == 'plus':
        return add
    elif input == 'minus':
        return sub
    elif input == 'times':
        return mult

op = get_op(input('What operation would you like?(plus/minus/times)'))
result = op(input('first operand?'), input('second operand?'))
```

8-14: Returning a function from a function

- Notice the 'inside-out' style: this is a hallmark of functional programming.
  - The result of one function is used as the input of the next function.
    - This is great if a result is only needed once.

8-15: Helper functions

- Many of the examples have used a helper function, such as 'add' or 'lower'
- This is to deal with syntactic issues when we want a function, but Python provides a method or operator to provide that effect.
- But having lots of helper functions can clutter up the namespace.
- What to do?

8-16: Local functions

- We can define helper functions locally.
  ```python
def lowerlist(wordlist):
    def lowercase(word):
        return word.lower()
    return map(lowercase, wordlist)
```
  - lowercase only exists inside of the lowerlist function
**8-17: Lambda**

- Lambda lets us define an *anonymous* function.
- Just as \( x + y \) says return the result of \( x + y \), \( \lambda x,y : x + y \) says to return a function that computes \( x + y \).
- \( \text{foo} = \lambda x,y : x + y \) is the same as \( \text{def foo}(x,y) : \text{return } x + y \).
- Note that lambda does not use return.

**8-18: Using Lambda**

- Lambda is very nice for “one-off” functions.
  
  \[
  \begin{align*}
  \text{filter}(\lambda x : x \% 2 == 0, [1, 2, 3, 4, 5]) \\
  \text{map}(\lambda x : x.\text{lower}(), ['HELLO', 'WORLD', 'HOW', 'ARE', 'YOU'])
  \end{align*}
  \]
- Very useful for places where you need a function syntactically, but don’t want to deal with creating a function.

**8-19: List comprehensions**

- Lambda and map/filter are great for simple problems, but it can be hard to combine them.
- Also, Lambda can get hard to read when the function body is large.
- Example: Compute cube(x) for every item in a list of the item is greater than zero.
  
  \[
  \begin{align*}
  \text{map}(\lambda x : x * x * x, \\
  \quad \text{filter}(\lambda y : y > 0, [-1, 0, 1, 2, 3]))
  \end{align*}
  \]
- Python provides another form of applicative programming that’s more readable: the list comprehension.

**8-20: List comprehensions**

- Basic idea: specify an operation that should take place for every element in a list.
- Returns a new list containing the results.
  
  \[
  \begin{align*}
  \{x**2 \text{ for } x \text{ in } [1, 2, 3, 4, 5]\} \\
  \{\text{item.}\text{lower()} \text{ for } \text{item in ['HELLO', 'WORLD', 'HOW', 'ARE', 'YOU']}\}
  \end{align*}
  \]
- Cleaner, more readable and English-looking

**8-21: List comprehensions**

- We can also add filters to this with if:
  
  \[
  \begin{align*}
  \{x**2 \text{ for } x \text{ in } [1,2,3,4,5] \text{ if } x \% 2 == 0\} \\
  \{\text{item.}\text{lower()} \text{ for } \text{item in ['HELLO', 'WORLD', 'HOW', 'ARE', 'YOU']} \text{ if } \text{len(item)} > 3\}
  \end{align*}
  \]
- We can also operate on multiple lists at once - to compute dot-product:
  
  \[
  \{x * y \text{ for } x \text{ in } [1,2,3,4,5] \text{ for } y \text{ in } [3,4,5,6,7]\}
  \]

**8-22: List comprehensions**

- Some cool list comprehension tricks:
  
  \[
  \begin{align*}
  \{\text{x=y} \text{ for } \text{x,y in zip([1,2,3,4,5],[5,6,7,8])}\}
  \end{align*}
  \]
- List comprehensions give you most of the things you’d like to do with map and filter, but in a more compact, readable form.
8-23: Summary

- Functional programming focuses on expressions that return results.
- These results are the inputs to other expressions.
- Filter, map and reduce let you apply a function to one or more sequences.
- Lambda lets you define an anonymous function.
- List comprehensions provide a cleaner syntax for lambda/map operations.
- Next time: Exploiting the idea that functions are objects: code is data.