Python Programming with First-Class Functions

Imperative vs. Declarative Programming

**Imperative**

Expresses how computation is performed by explicit statements that change the program's state.

Programs use loops, conditionals, variables and data structures (lists, dicts, sets) that change during the program flow.

**Declarative**

Expresses what the computation accomplishes without showing how, leaving the details to the language.

Programs are written by using certain building blocks that (mostly) don’t have side-effects on data. When the building blocks are functions, we get functional programming.

Python is a programming language that can express both of these paradigms and mix them as needed.

---

```python
import matplotlib.pyplot as plt
from numpy import arange
from math import *

def plotSin(xLo, xHi, step, style):
    """plots values of sin from xLo to xHi using given step and style""
    inputs = arange(xLo, xHi, step)
    outputs = [sin(inp) for inp in inputs]
    plt.plot(inputs, outputs, style)
    plt.show()

plotSin(-3, 3, 0.1, 'b-')
plotSin(-10, 10, 0.3, 'ro')
```

---

```python
def plotCube(xLo, xHi, step, style):
    """plots values of cube from xLo to xHi using given step and style""
    inputs = arange(xLo, xHi, step)
    outputs = [inp**3 for inp in inputs]
    plt.plot(inputs, outputs, style)
    plt.show()

plotCube(-3, 5, 0.2, 'm+')
plotCube(-2, 2, 0.1, 'g-')
```
**plotSin vs. plotCube**

```python
def plotSin(xLo, xHi, step, style):
    """plots values of sin from xLo to xHi using given step and style""
    inputs = arange(xLo, xHi, step)
    outputs = [sin(inp) for inp in inputs]
    plt.plot(inputs, outputs, style)
    plt.show()

def plotCube(xLo, xHi, step, style):
    """plots values of cube from xLo to xHi using given step and style""
    inputs = arange(xLo, xHi, step)
    outputs = [inp**3 for inp in inputs]
    plt.plot(inputs, outputs, style)
    plt.show()
```

**Abstracting over the difference**

```python
def plotFun(fun, xLo, xHi, step, style):
    """plots values of given fcn from xLo to xHi using given step and style""
    inputs = arange(xLo, xHi, step)
    outputs = [fun(inp) for inp in inputs]
    plt.plot(inputs, outputs, style)
    plt.show()()

def cube(x):
    return x**3

plotFun(sin, -10, 10, 0.3, 'r-')
plotFun(cube, -2, 2, 0.1, 'm-')
```

**Functional Programming**

**sin and cube are First-class Function Objects**

```
In [2]: cube
Out[2]: <function __main__.cube>

In [3]: sin
Out[3]: <function math.sin> # Function from math module

In [4]: type(cube)
Out[4]: function # implemented by us

In [5]: type(sin)
Out[5]: builtin_function_or_method # implemented by Python

In [6]: import inspect
In [7]: inspect.getargspec(plotSin)
Out[7]: ArgSpec(args=['xLo', 'xHi', 'step', 'style'],
                  varargs=None, keywords=None, defaults=None)

In [8]: inspect.getargspec(cube)
Out[8]: ArgSpec(args=['n'], varargs=None,
                  keywords=None, defaults=None)
```

**First-Class Property #1: Can be Named by a Variable**

```
In [9]: myFun = cube

In [10]: myFun(10)
Out[10]: 1000

In [11]: myFun
Out[11]: <function __main__.cube>

In [12]: myFun = parabola

In [13]: parabola(10)
Out[13]: -90

In [14]: double(10)
Out[14]: 20
```

```python
def double(z):
    return z*2

def parabola(y):
    return 10 - y*y
```
First-Class Property #2: Can be Stored in a Collection

```python
In [16]: functionList = [cube, parabola, double]
In [17]: functionList
Out[17]: [function __main__.cube,
        function __main__.parabola,
        function __main__.double]
In [18]: [fcn(10) for fcn in functionList]
Out[18]: [1000, -90, 20]
```

First-Class Property #3: Can be Passed as an Argument to Another Function

```python
def testOnRange(fun, lo, hi):
    return [(i, fun(i)) for i in range(lo, hi+1)]
```

```python
In [25]: testOnRange(cube, 0, 10)
Out[25]: [(0, 0), (1, 1), (2, 8), (3, 27), (4, 64), (5, 125), (6, 216), (7, 343), (8, 512), (9, 729), (10, 1000)]
In [26]: testOnRange(double, 3, 7)
Out[26]: [(3, 6), (4, 8), (5, 10), (6, 12), (7, 14)]
```

```python
In [27]: testOnRange('double', 3, 7)  # Pass function object, not name of function
TypeError: 'str' object is not callable
```

```python
def testOnRange(fun, lo, hi):
    return [(i, fun(i)) for i in range(lo, hi+1)]
```

```python
In [28]: getFunction(2)(6)
Out[28]: 12
In [29]: testOnRange(getFunction(0), 2, 6)
Out[29]: [(2, 8), (3, 27), (4, 64), (5, 125), (6, 216)]
```

First-Class Property #4: Can be Returned as Result from another Function [1]

```python
functionList = [cube, parabola, double]
def getFunction(index):
    return functionList[index]
```

```python
In [28]: getFunction(2)(6)
Out[28]: 12
In [29]: testOnRange(getFunction(0), 2, 6)
Out[29]: [(2, 8), (3, 27), (4, 64), (5, 125), (6, 216)]
```

```python
import random
def randomFunction():
    return functionList[random.randint(0, 2)]
```

```python
In [30]: [randomFunction()(i) for i in range(1, 6)]
Out[30]: [1, 6, 27, -6, 10] # One of many possible answers
```

First-Class Property #4: Can be Returned as Result from another Function [2]

Recall that inner function definitions may be nested within outer ones and can refer to parameter of outer definitions.

```python
def linear(a, b):
    def innerFunction(x):
        return a*x + b  # inner functions can refer to 
        # parameters of outer functions
    return innerFunction
```

```python
In [31]: lin1 = linear(10, -7)
In [32]: testOnRange(lin1, -3, 3)
Out[32]: [(-3, -37), (-2, -27), (-1, -17), (0, -7), 
        (1, 3), (2, 13), (3, 23)]
In [33]: lin2 = linear(0.3, 5)
In [34]: testOnRange(lin3, -3, 3)
Out[34]: [(-3, 4.1), (-2, 4.4), (-1, 4.7), (0, 5.0), 
        (1, 5.3), (2, 5.6), (3, 5.9)]
```
Modeling Nested Functions: linear

```python
def linear(a, b):
    def innerFunction(x):
        return a*x + b
    return innerFunction
```

The value of `innerFunction` returned by `linear(10,-7)` is a so-called closure that has two parts:

1. The function definition
2. An environment = a dictionary of local variables other than the parameters used in the body of the function definition

When the closure is called, all the key/value pairs in the environment are added to the variable section of the function frame, along with the parameters.

Higher Order Functions

Higher-order functions (HOFs) are:
- functions that take at least some arguments that are themselves functions
- functions that return functions as results

Examples of HOFs we have seen so far are:
- `plotFun, testOnRange` (take a function as argument)
- `getFunction, randomFunction, linear` (return a function as result)

Now we study some built-in HOFs in Python:
- `sorted`
- `reduce`
- `map`
- `filter`

Sorting tuples: problem statement

```python
people = [('Mary Beth Johnson', 18), ('Ed Smith', 17), ('Janet Doe', 25), ('Bob Miller', 31)]
def firstName(personTuple):
    return personTuple[0].split()[0]
def lastName(personTuple):
    return personTuple[0].split()[-1]
def age(personTuple):
    return personTuple[1]
```

How did we sort the `people` tuples by:
- the full name?
- the age?
- the last name?

Sorting tuples: Decorate/Sort/Undecorate (DSU)

By default, `sorted()` looks at index 0; only looks at other indices to break ties. This is the basis of the Decorate, Sort, Undecorate (DSU) strategy for sorting (seen in PS06 Titanic and PS09 Google Book Search)

For example, to sort people by age:
- **Decorate**: create new list of pairs with age at index 0 and person at index 1
  ```python
  [(18, ('Mary Beth Johnson', 18)),
   (17, ('Ed Smith', 17)),
   (25, ('Janet Doe', 25)),
   (31, ('Bob Miller', 31))]
  ```
- **Sort** this list:
  ```python
  [(17, ('Ed Smith', 17)),
   (18, ('Mary Beth Johnson', 18)),
   (25, ('Janet Doe', 25)),
   (31, ('Bob Miller', 31))]
  ```
- **Undecorate**: map the list back to person
  ```python
  ['Ed Smith', 17],
  ('Mary Beth Johnson', 18),
  ('Janet Doe', 25),
  ('Bob Miller', 31)]
  ```
Sorting with key argument

`sorted` takes an optional argument named `key` that specifies a function that for each element determines how it should be compared to other elements.

```
In [35]: sorted(people, key=age)
Out[35]: [('Ed Smith', 17), ('Mary Beth Johnson', 18), ('Janet Doe', 25), ('Bob Miller', 31)]
```

Exercise: Sorting with key

Use the `key` argument of `sorted` to sort the elements of people in ascending order:
- By their last name
- By the length of their first name

```
In [60]: sorted(people, key=lastName)
Out[61]: [('Janet Doe', 25), ('Mary Beth Johnson', 18), ('Bob Miller', 31), ('Ed Smith', 17)]

In [60]: sorted(people, key=firstNameLength)
Out[61]: [('Ed Smith', 17), ('Bob Miller', 31), ('Mary Beth Johnson', 18), ('Janet Doe', 25)]
```

def firstNameLength(personTuple):
    return len(firstName(personTuple))

Exercise with tie-breakers

Define a `key` function so that `people1 + people2` is sorted in ascending order first by first name, then by length of last name, then by age.

```
def firstLengthLastAge(person):
    return (firstName(person), len(lastName(person)), age(person))

In [37]: sorted(people1 + people2, key=firstLengthLastAge)
```

Breaking ties with key functions

The `people2` list has many ambiguities due to first names, last names, and ages that are the same:

```
people2 = [('Ed Jones', 18), ('Ana Doe', 25), ('Ed Doe', 18), ('Bob Doe', 25), ('Ana Jones', 18)]
```

We define `ageLastFirst` to be a key function that will first sort by age, then by last name (if ages are equal), then by first name (if age and last name are equal.)

```
def ageLastFirst(person):
    return (age(person), lastName(person), firstName(person))
```

```
In [36]: sorted(people2, key=ageLastFirst)
Out[36]: [('Ed Doe', 18), ('Ana Jones', 18), ('Ed Jones', 18), ('Ana Doe', 25), ('Bob Doe', 25)]
```
**Lambda notation creates anonymous functions**

It is often inconvenient to define a named function just in order to pass it as the functional argument to higher-order functions like `plotFun`, `sorted`, `reduce`, `map`, and `filter`.

Python provides **lambda notation** for creating an anonymous function, which can be used directly with HOFs without introducing named functions.

```
plotFcn(lambda n : n*n, -3, 3, 0.1, 'r-')
plotFcn(lambda x : x*sin(x), -30, 30, 0.1, 'g-')
```

**Lambda creates anonymous functions**

A function defined with `def`

```
def cube(x):
    return x*x*x
```

In [38]: `cube(4)`
Out[38]: 64

In [39]: `cube.__name__`
Out[39]: 'cube'

A function defined with `lambda`

```
In [40]: (lambda x : x*x)(4)
Out[40]: 16
```

```
In [41]: sq = lambda x : x*x
In [42]: sq(5)
Out[42]: 25
```

```
In [43]: sq.__name__
Out[43]: '<lambda>'
```

**Anatomy of a lambda expression**

A **lambda expression** has the form:

```
lambda param: bodyExpression
```

Keyword meaning

<table>
<thead>
<tr>
<th>parameter name of this function</th>
<th>expression for result of this function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>“I am a function”</td>
<td>It does not use an explicit <code>return</code>!</td>
</tr>
</tbody>
</table>

- **Lambda num**: `num*2`
  - I am a function that takes an argument named `num` and returns the result of doubling it.

- **Lambda n**: `(n%2) == 0`
  - I am a function that takes an argument named `n` and returns a boolean that indicates whether it’s even.

**Why lambda?**

In the 1930s and 40s, Alonzo Church developed a model of computation called the **lambda calculus**.

It is a programming language with only three kinds of expressions:
- variables, e.g. `x`
- functions expressed in lambda notation, e.g. the identity function \(\lambda x . x\)
- function application, e.g. \((\lambda x . x) y\)

Remarkably, this simple language can express any computable program – even though it has no built-in numbers, arithmetic, booleans, conditionals, lists, loops, or recursion! (Take CS235 & CS251)
Exercises: **lambda** and Sorting

For each of the following expressions, predict the order of the sorted results:

```python
sorted(people, key=lambda person: len(lastName(person)))
```

```python
sorted(people+people2,
      key=lambda p: (len(firstName(p)+lastName(p)), age(p)))
```

```python
sorted(people+people2,
      key=lambda x: (lastName(x)[-1], firstName(x)[1]))
```

Below, replace the `??` with a `lambda` to sort `people2` in ascending order first by age, then by length of last name, then by first name.

```python
sorted(people2,
      key=lambda p: (age(p),
                     len(lastName(p)),
                     firstName(p)))
```

---

**The list accumulation pattern**

In the list accumulation pattern, the elements of an input list are combined into a result using a binary operator applied left-to-right.

```
sum( [8, 3, 6, 7])
```

---

**reduce** captures the list accumulation pattern

Python provides a `reduce` function that captures this pattern. When invoked as `reduce(combiner, elements)`, it returns the result of combining the elements of the list from left to right using the binary function `combiner`.

```
reduce(comb, [e1, e2, ..., en])
```

---

**reduce** examples

In [44]: `reduce(lambda sumSoFar,n: sumSoFar+n, [2,7,1,3])`

Out[44]: 13 # 2 + 7 + 1 + 3

In [45]: `reduce(lambda prodSoFar,n: prodSoFar*n, [2,7,1,3])`

Out[45]: 42 # 2 * 7 * 1 * 3

In [46]: `reduce(lambda pairsSoFar,n: (pairsSoFar,n), [2,7,1,3])`

Out[46]: (((2, 7), 1), 3)

In [47]: `reduce(lambda sentenceSoFar,word: sentenceSoFar + ' ' + word, ['I', 'have', 'a', 'dream'])`

Out[47]: 'I have a dream'
Your turn: joinStrings

```
def joinStrings(sep, strings):
    return reduce(lambda stringSoFar,s: stringSoFar + sep + s, strings)
```

Functional Programming 19-29

reduce corner cases

```
In [48]: reduce(lambda a,b:a*b, [7])
Out[48]: 7

In [49]: reduce(lambda a,b:a*b, [])
------> 1 reduce(lambda a,b:a*b, [])

TypeError: reduce() of empty sequence with no initial value

In [50]: reduce(lambda a,b:a*b, [], 1)
Out[50]: 1

In [51]: reduce(lambda a,b:a+b, [7], 10)
Out[51]: 17
```

Optional initial value for accumulating the accumulating result.

Functional Programming 19-30

reduce with an initial result

Sometimes it's necessary to provide reduce with an initial result for the accumulation process.

It's always necessary if we want to accumulate over an empty list!

```
def myReduce(combine, elts, initialResult=None):
    '''Returns result of combining elements in elts
    from left to right, starting with initialResult
    (if it is supplied), or elts[0] (if it is not supplied)'''
    if initialResult != None:
        result = initialResult
        values = elts
    elif len(elts) == 0:
        raise TypeError('myReduce on empty sequence with no initialResult')
    else:
        result = elts[0]
        values = elts[1:]
    for val in values:
        result = combine(result, val)
    return result
```

Functional Programming 19-31

Our own reduce function

```
def myReduce(combine, elts, initialResult=None):
    '''Returns result of combining elements in elts
    from left to right, starting with initialResult
    (if it is supplied), or elts[0] (if it is not supplied)'''
    if initialResult != None:
        result = initialResult
        values = elts
    elif len(elts) == 0:
        raise TypeError('myReduce on empty sequence with no initialResult')
    else:
        result = elts[0]
        values = elts[1:]
    for val in values:
        result = combine(result, val)
    return result
```
**Your turn: reduce examples**

In [52]: `reduce(lambda listSoFar, list: listSoFar + list, [[[8, 2, 5], [17]], [], [6, 3]])`

Out[52]: `[8, 2, 5, 17, 6, 3]`

In [53]: `reduce(lambda lenSoFar, s: lenSoFar + len(s), ['I', 'have', 'a', 'dream'], 0)`

Out[53]: `11`  # sum of string lengths

# Simpler approach:
# sum(map(len, ['I', 'have', 'a', 'dream']))

In [54]: `reduce(lambda strSoFar, n: strSoFar + str(n), [7, 215, 36], '')`

Out[54]: `'721536'`

In [55]: `reduce(lambda boolSoFar, n: boolSoFar and n>0, [8, 3, 7, 6, 4], True)`

Out[55]: `True`  # Are all elements positive?

**A new way to define factorial**

Recall that the factorial of a nonnegative integer n is the product of the integers 1 up to n.

Here’s a definition that expresses this almost directly:

```python
def factorial(n):
    return reduce(lambda prodSoFar, num: prodSoFar * num, range(1, n+1), 1)
```

**Higher order list operations: map, filter**

- Map pattern:
  - List comprehension: `[str(x) for x in numList]`
  - Map function: `map(str, numList)`

- Filter pattern:
  - `def isEven(x):
    return x%2==0`
  - List comprehension: `[x for x in numList if isEven(x)]`
  - Filter function: `filter(isEven, numList)`

- Why use `map` and `filter` instead of list comprehensions?
  - Although `map` and `filter` can always be expressed with list comprehensions, most other higher-order functions cannot
  - Good practice with higher order functions

**Review: the list mapping pattern**

In the list mapping pattern:

- the output list is the same length as the input list;
- each element of the output list is the result of applying some function to the corresponding element of the input list.

```python
mapDouble([8, 3, 6, 7, 2, 4])
```

```plaintext
8 8 8 8 8 8
```

```python
mapPluralize(['donut', 'muffin', 'bagel'])
```

```plaintext
[donuts, muffins, bagels]
```
**map** captures the list mapping pattern

Python provides a **map** function that captures this pattern. When invoked as **map**(*function, elements*), it returns a new output list that's the same length as *elements* in which every element is the result of applying *function* to the corresponding element of *elements*.

```
map(f, [e1, e2, ..., en])
   |   |   |
   f  f  f
   ↓  ↓  ↓
[f(e1), f(e2), ..., f(en)]
```

**Nested Functions Revisited**

- **mapScale**(factor, listOfNumbers) takes a scaling factor and a list of numbers and returns a list that results from scaling each number by the factor
  - Definition using list comprehension:
    ```python
def mapScale(factor, listOfNumbers):
    return [n * factor for n in listOfNumbers]
```
  - Definition using nested inner function:
    ```python
def mapScale(factor, listOfNumbers):
    def scaleByFactor(n):
        return factor * n
    return map(scaleByFactor, listOfNumbers)
```
  - Functions defined within functions have access to all local variables of the parent function

```
def scaleByFactor(n):
    return factor * n
def mapScale(factor, nums):
    return map(scaleByFactor, nums)
```

**Eliminating inner definitions with lambda**

```
def mapScale(factor, nums):
    return map(lambda n: factor * n, nums)
```

In [62]: mapScale(3, [8, 3, 6, 7, 2, 4])
Out[62]: [24, 9, 18, 21, 6, 12]

In [63]: mapScale(10, [8, 3, 6, 7, 2, 4])
Out[63]: [80, 30, 60, 70, 20, 40]

# version without an inner definition
def mapScale(factor, nums):
    return map(lambda n: factor * n, nums)
Your turn: avoiding inner definitions w/ \texttt{lambda}

\begin{verbatim}
def mapPreConcat(prefix, strings):
    return map(lambda s: prefix + s, listOfStrings)
\end{verbatim}

In [64]: mapPreConcat('com', ['puter', 'pile', 'mute'])
Out[64]: ['computer', 'compile', 'commute']

\begin{verbatim}
def mapSuffixes(string):
    return map(lambda i: string[i:], range(len(string)))
\end{verbatim}

In [65]: mapSuffixes('python')
Out[65]: ['python', 'ython', 'thon', 'hon', 'on', 'n']

Limitations of \texttt{lambda}

\texttt{lambda} bodies must be a single expression.

They can’t be statement (such as \texttt{print} statements, \texttt{if} statements) or sequences of statements.

For these, \texttt{map} still needs helper functions!

\begin{verbatim}
def printDouble(num):
    print num, '->', 2*num
    return num
\end{verbatim}

In [19]: map(printDouble, [8, 3, 5])
Out[19]: [8, 3, 5]

\begin{verbatim}
def myAbs(num):
    if num < 0:
        return -num
    else:
        return num
\end{verbatim}

In [20]: map(myAbs, [-7, 2, -6])
Out[20]: [7, 2, 6]

Using \texttt{map} on non-list sequences

The \texttt{map} function can be used on any sequence, but always returns a list.

In [17]: map(lambda s: s.upper(), 'foo')
Out[17]: ['F', 'O', 'O']

In [18]: map(lambda s: s.upper(), ('ant', 'bat', 'cat'))
Out[18]: ['ANT', 'BAT', 'CAT']

For mapping over each letter of a string, we can get a string from the resulting list of strings by using the \texttt{join} method.

In [19]: ''.join(map(lambda s: s.upper(), 'foo'))
Out[19]: 'FOO'

Mapping over multiple lists

The \texttt{map} function allows any number of list arguments as long as the supplied function takes a number of arguments that’s the same as the number of lists.

\begin{verbatim}
def printDouble(num):
    print num, '->', 2*num
    return num
\end{verbatim}

In [13]: map(lambda a,b: a+b, [8, 3, 5], [10, 20, 30])
Out[13]: [18, 23, 35]

In [14]: map(lambda a,b: a*b, [8, 3, 5], [10, 20, 30])
Out[14]: [80, 60, 150]

In [15]: map(lambda a,b: (a,b), [8, 3, 5], [10, 20, 30])
Out[15]: [(8, 10), (3, 20), (5, 30)]

In [16]: map(lambda a,b,c: a*b+c, [8, 3], [10, 20], [7, 2])
Out[16]: [87, 62]
Mapping over multiple Lists

When mapping over multiple lists, all the lists must have the same length; if not, an exception will be raised.

```
In [17]: map(lambda a,b: a+b, [8,3,5,6], [10,20,30])
```

```
TypeError Traceback (most recent call last)
<ipython-input-93-e73284726> in <module>
    1    2
----> 3 map(lambda a,b: a+b, [8,3,5,6], [10,20,30])

TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

Our own map function

How does map work? To illustrate, we can define our own version of Python's map function as follows:

```
def myMap(f, elts):
    result = []
    for e in elts:
        result.append(f(e))
    return result
```

(This version does not support multiple list arguments.)

```
In[21]: myMap(double, [8,3,6,7,2,4])
Out[21]: [16,6,12,14,4,8]
```

Review: the list filtering pattern

In the list filtering pattern, the output list includes only those input elements for which a certain predicate is True.

```
filterEvens( [8, 3, 6, 7, 2, 4])
           |   |   |   |   |
isEven  isEven  isEven  isEven  isEven  isEven
↓     ↓     ↓     ↓     ↓     ↓
  True  False  True  False  True  True
[8, 6, 2, 4]
```

filter captures the list filtering pattern

Python provides a filter function that captures this pattern. When invoked as filter(pred, elts), it returns a new output list contains only those elements in elts for which the predicate pred returns True (in the same relative order).

```
filter(pred, [e1, e2, ..., en])
     |     |     |
pred  pred  pred
↓     ↓     ↓
bool1 bool2 booln
  keep e1  if bool1 is True;  keep en
  if bool1 is True;  if bool2 is True;
  omit e1  omit e2  if bool en
  if bool1 is False is False
```

```
Your turn: filter examples using \textbf{lambda}

In [29]: filter(lambda n: n>0, [8, -3, 6, 7, -2, -4, 5])
Out[29]: [8, 6, 7, 5]

In [30]: filter(lambda s: s[0]=='a', ['a', 'be', 'at', 'bat', 'ant', 'cat', 'dog', 'aardvark'])
Out[30]: ['a', 'at', 'any', 'aardvark']

Functional Programming 19-49

Using \textbf{filter} on non-list sequences

The output of \texttt{filter} is the same type of sequence as the input.

In [40]: filter(lambda c: c not in 'aeiouy', 'facetiously')
Out[40]: 'fctsl'

In [41]: filter(lambda n: (n%2)==0, [1,2,3,4,5])
Out[41]: (2, 4) # A tuple, not a list

Functional Programming 19-51

Your turn: \textbf{filterSameLength}

\begin{verbatim}
def filterSameLength(s, words):
    return filter(lambda w: len(s) == len(word), words)
\end{verbatim}

In [25]: filterSameLength('ant', ['the', 'gray', 'cat', 'is', 'big'])
Out[25]: ['the', 'cat', 'big']

In [26]: filterSameLength('purr', ['the', 'gray', 'cat', 'is', 'big'])
Out[26]: ['gray']

In [27]: filterSameLength('q', ['the', 'gray', 'cat', 'is', 'big'])
Out[27]: []

Functional Programming 19-50

Our own \textbf{filter} function

How would we define our own \texttt{filter} function?

\begin{verbatim}
def myFilter(pred, list):
    '''Returns a new list that contains every element of \texttt{elts} satisfying
    the predicate \texttt{pred}, in the same relative order.
    ...'''
    resultList = []
    for elt in list:
        if pred(elt):
            resultList.append(elt)
    return resultList
\end{verbatim}

In [40]: myFilter(lambda c: c not in 'aeiouy', 'facetiously')
Out[40]: 'fctsl'

In [41]: myFilter(lambda n: (n%2)==0, [1,2,3,4,5])
Out[41]: (2, 4) # A tuple, not a list

Functional Programming 19-52