LECTURE 3

Python Basics Part 2

FUNCTIONAL PROGRAMMING TOOLS

- Last time, we covered function concepts in depth. We also mentioned that Python allows for the use of a special kind of function, a *lambda* function.
- Lambda functions are small, anonymous functions based on the lambda abstractions that appear in many functional languages.
- As stated before, Python can support many different programming paradigms including functional programming.

Right now, we'll take a look at some of the handy functional tools provided by Python.

LAMBDA FUNCTIONS

- Lambda functions within Python.
  - Use the keyword *lambda* instead of *def.*
  - Can be used wherever function objects are used.
  - Restricted to one expression.
  - Typically used with functional programming tools.

```
>>> def f(x):
...     return x**2
...
>>> print (f(8))
64
>>> g = lambda x: x**2
>>> print (g(8))
64
```

FUNCTIONAL PROGRAMMING TOOLS

- **Filter**
- • filter(*function*, *sequence*) filters items from sequence for which function(*item*) is true.
- • Returns a string or tuple if sequence is one of those types, otherwise result is a list.

```
def even(x):
    if x % 2 == 0:
        return True
    else:
        return False

print(filter(even, range(0,30)))
```

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]

FUNCTIONAL PROGRAMMING TOOLS

- **Map**
- • map(*function, sequence*) applies function to each item in sequence and returns the results as a list.
- • Multiple arguments can be provided if the function supports it.

```
def square(x):
    return x**2

print(map(square, range(0,11)))
```

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

FUNCTIONAL PROGRAMMING TOOLS

- **Map**
- • map(*function, sequence*) applies function to each item in sequence and returns the results as a list.
- • Multiple arguments can be provided if the function supports it.

```
def expo(x, y):
    return x**y

print(map(expo, range(0,5), range(0,5)))
```

[1, 1, 4, 27, 256]

FUNCTIONAL PROGRAMMING TOOLS

- **Reduce**
- • reduce(*function, sequence*) returns a single value computed as the result of performing *function* on the first two items, then on the result with the next item, etc.
- • There's an optional third argument which is the starting value.

```
def fact(x, y):
    return x*y

print(reduce(fact, range(1,5)))
```

```
24
```

FUNCTIONAL PROGRAMMING TOOLS

- We can combine lambda abstractions with functional programming tools. This is especially useful when our function is small – we can avoid the overhead of creating a function definition for it by essentially defining it in-line.

```
>>> print(map(lambda x: x**2, range(0,11)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

MORE DATA STRUCTURES

- Lists
  - Slicing
  - Stacks and Queues
- Tuples
- Sets and Frozensets
- Dictionaries
- How to choose a data structure.
- Collections
  - Deques and OrderedDicts

WHEN TO USE LISTS

- When you need a collection of elements of varying type.
- When you need the ability to order your elements.
- When you need the ability to modify or add to the collection.
- When you don't require elements to be indexed by a custom value.
- When you need a stack or a queue.
- When your elements are not necessarily unique.

CREATING LISTS

- To create a list in Python, we can use bracket notation to either create an empty list or an initialized list.

```
mylist1 = [] # Creates an empty list
mylist2 = [expression1, expression2, ...]
mylist3 = [expression for variable in sequence]
```

- The first two are referred to as *list displays*, where the last example is a *list comprehension*.

CREATING LISTS

- We can also use the built-in list constructor to create a new list.

```
mylist1 = list()
mylist2 = list(sequence)
mylist3 = list(expression for variable in sequence)
```

- The sequence argument in the second example can be any kind of sequence object or iterable. If another list is passed in, this will create a copy of the argument list.

CREATING LISTS

- Note that you cannot create a new list through assignment.

```
# mylist1 and mylist2 point to the same list
mylist1 = mylist2 = []

# mylist3 and mylist4 point to the same list
mylist3 = []
mylist4 = mylist3

mylist5 = []; mylist6 = [] # different lists
```

ACCESSING LIST ELEMENTS

- If the index of the desired element is known, you can simply use bracket notation to index into the list.

```
>>> mylist = [34,67,45,29]
>>> mylist[2]
45
```

- If the index is not known, use the `index()` method to find the first index of an item. An exception will be raised if the item cannot be found.

```
>>> mylist = [34,67,45,29]
>>> mylist.index(67)
1
```

SLICING AND SLIDING

- The length of the list is accessible through `len(mylist)`.
- Slicing is an extended version of the indexing operator and can be used to grab sublists.

```
mylist[start:end]   # items start to end-1
mylist[start:]      # items start to end of the array
mylist[:end]        # items from beginning to end-1
mylist[:]           # a copy of the whole array
```

- You may also provide a step argument with any of the slicing constructions above.

```
mylist[start:end:step]  # start to end-1, by step
```

SLICING AND SLIDING

- The start or end arguments may be negative numbers, indicating a count from the end of the array rather than the beginning. This applies to the indexing operator.

```
mylist[-1]      # last item in the array
mylist[-2:]     # last two items in the array
mylist[:-2]     # everything except the last two items
```

- Some examples:

```
mylist = [34, 56, 29, 73, 19, 62]
mylist[-2]      # yields 19
mylist[-4::2]   # yields [29, 19]
```

INSERTING/REMOVING ELEMENTS

- To add an element to an existing list, use the append() method.

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.append(47)
>>> mylist
[34, 56, 29, 73, 19, 62, 47]
```

- Use the extend() method to add all of the items from another list.

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.extend([47,81])
>>> mylist
[34, 56, 29, 73, 19, 62, 47, 81]
```

INSERTING/REMOVING ELEMENTS

- Use the insert(*pos, item*) method to insert an item at the given position. You may also use negative indexing to indicate the position.

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.insert(2,47)
>>> mylist
[34, 56, 47, 29, 73, 19, 62]
```

- Use the remove() method to remove the first occurrence of a given item. An exception will be raised if there is no matching item in the list.

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.remove(29)
>>> mylist
[34, 56, 73, 19, 62]
```

LISTS AS STACKS

- You can use lists as a quick stack data structure.
- The append() and pop() methods implement a LIFO structure.
- The pop(*index*) method will remove and return the item at the specified index. If no index is specified, the last item is popped from the list.

```
>>> stack = [34, 56, 29, 73, 19, 62]
>>> stack.append(47)
>>> stack
[34, 56, 29, 73, 19, 62, 47]
>>> stack.pop()
47
>>> stack
[34, 56, 29, 73, 19, 62]
```

LISTS AS QUEUES

- Lists *can* be used as queues natively since insert() and pop() both support indexing. However, while appending and popping from a list are fast, inserting and popping from the beginning of the list are slow.
- Use the special *deque* object from the *collections* module.

```
>>> from collections import deque
>>> queue = deque([35, 19, 67])
>>> queue.append(42)
>>> queue.append(23)
>>> queue.popleft()
35
>>> queue.popleft()
19
>>> queue
deque([67, 42, 23])
```

OTHER OPERATIONS

- The `count(x)` method will give you the number of occurrences of item x within the list.

```
>>> mylist = ['a', 'b', 'c', 'd', 'a', 'f', 'c']
>>> mylist.count('a')
2
```

- The `sort()` and `reverse()` methods sort and reverse the list in place. The `sorted(mylist)` and `reversed(mylist)` built-in functions will return a sorted and reversed copy of the list, respectively.

```
>>> mylist = [5, 2, 3, 4, 1]
>>> mylist.sort()
>>> mylist
[1, 2, 3, 4, 5]
>>> mylist.reverse()
>>> mylist
[5, 4, 3, 2, 1]
```

CUSTOM SORTING

- Both the `sorted()` built-in function and the `sort()` method of lists accept some optional arguments.

```
sorted(iterable[, cmp[, key[, reverse]]])
```

- The *cmp* argument specifies a custom comparison function of two arguments which should return a negative, zero or positive number depending on whether the first argument is considered smaller than, equal to, or larger than the second argument. The default value is None.
- The *key* argument specifies a function of one argument that is used to extract a comparison key from each list element. The default value is None.
- The *reverse* argument is a Boolean value. If set to True, then the list elements are sorted as if each comparison were reversed.

CUSTOM SORTING

-

```
>>> mylist = ['b', 'A', 'D', 'c']
>>> mylist.sort(cmp = lambda x,y: cmp(x.lower(), y.lower()))
>>> mylist
['A', 'b', 'c', 'D']
```

Alternatively,

```
>>> mylist = ['b', 'A', 'D', 'c']
>>> mylist.sort(key = str.lower)
>>> mylist
['A', 'b', 'c', 'D']
```

str.lower() is a built-in string method.

WHEN TO USE SETS

- When the elements must be unique.
- When you need to be able to modify or add to the collection.
- When you need support for mathematical set operations.
- When you don't need to store nested lists, sets, or dictionaries as elements.

CREATING SETS

- Create an empty set with the set constructor.

```
myset = set()
myset2 = set([]) # both are empty sets
```

- Create an initialized set with the set constructor or the { } notation. Do not use empty curly braces to create an empty set – you'll get an empty dictionary instead.

```
myset = set(sequence)
myset2 = {expression for variable in sequence}
```

HASHABLE ITEMS

- The way a set detects non-unique elements is by indexing the data in memory, creating a hash for each element. This means that all elements in a set must be *hashable*.
- All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are also hashable by default.

MUTABLE OPERATIONS

- The following operations are not available for frozensets.
- The `add(x)` method will add element x to the set if it's not already there. The `remove(x)` and `discard(x)` methods will remove x from the set.
- The `pop()` method will remove and return an arbitrary element from the set. Raises an error if the set is empty.
- The clear() method removes all elements from the set.

```
>>> myset = set('abracadabra')
>>> myset
set(['a', 'b', 'r', 'c', 'd'])
>>> myset.add('y')
>>> myset
set(['a', 'b', 'r', 'c', 'd', 'y'])
>>> myset.remove('a')
>>> myset
set(['b', 'r', 'c', 'd', 'y'])
>>> myset.pop()
'b'
>>> myset
set(['r', 'c', 'd', 'y'])
```

MUTABLE OPERATIONS CONTINUED

set |= *other* | ...

Update the set, adding elements from all others.

set &= *other* & ...

Update the set, keeping only elements found in it and all others.

set -= *other* | ...

Update the set, removing elements found in others.

set ^= *other*

Update the set, keeping only elements found in either set, but not in both.

MUTABLE OPERATIONS CONTINUED

```
>>> s1 = set('abracadabra')
>>> s2 = set('alacazam')
>>> s1
set(['a', 'b', 'r', 'c', 'd'])
>>> s2
set(['a', 'l', 'c', 'z', 'm'])
>>> s1 |= s2
>>> s1
set(['a', 'b', 'r', 'c', 'd', 'l', 'z', 'm'])
>>> s1 = set('abracadabra')
>>> s1 &= s2
>>> s1
set(['a', 'c'])
```

SET OPERATIONS

- The following operations are available for both set and frozenset types.
- Comparison operators >=, <= test whether a set is a superset or subset, respectively, of some other set. The > and < operators check for proper supersets/subsets.

```
>>> s1 = set('abracadabra')
>>> s2 = set('bard')
>>> s1 >= s2
True
>>> s1 > s2
True
>>> s1 <= s2
False
```

SET OPERATIONS

- Union: `set | other | …`
  - Return a new set with elements from the set and all others.
- Intersection: `set & other & …`
  - Return a new set with elements common to the set and all others.
- Difference: `set - other - …`
  - Return a new set with elements in the set that are not in the others.
- Symmetric Difference: `set ^ other`
  - Return a new set with elements in either the set or other but not both.

SET OPERATIONS

```
>>> s1 = set('abracadabra')
>>> s1
set(['a', 'b', 'r', 'c', 'd'])
>>> s2 = set('alacazam')
>>> s2
set(['a', 'l', 'c', 'z', 'm'])
>>> s1 | s2
set(['a', 'b', 'r', 'c', 'd', 'l', 'z', 'm'])
>>> s1 & s2
set(['a', 'c'])
>>> s1 - s2
set(['b', 'r', 'd'])
>>> s1 ^ s2
set(['b', 'r', 'd', 'l', 'z', 'm'])
```

OTHER OPERATIONS

- `s.copy()` returns a shallow copy of the set *s*.
- `s.isdisjoint(other)` returns True if set *s* has no elements in common with set *other*.
- `s.issubset(other)` returns True if set *s* is a subset of set *other*.
- `len`, `in`, and `not in` are also supported.

WHEN TO USE TUPLES

- When storing elements that will not need to be changed.
- When performance is a concern.
- When you want to store your data in logical immutable pairs, triples, etc.

CONSTRUCTING TUPLES

- An empty tuple can be created with an empty set of parentheses.
- Pass a sequence type object into the tuple() constructor.
- Tuples can be initialized by listing comma-separated values. These do not need to be in parentheses but they can be.
- One quirk: to initialize a tuple with a single value, use a trailing comma.

```
>>> t1 = (1, 2, 3, 4)
>>> t2 = "a", "b", "c", "d"
>>> t3 = ()
>>> t4 = ("red", )
```

TUPLE OPERATIONS

- Tuples are very similar to lists and support a lot of the same operations.
- Accessing elements: use bracket notation (e.g. `t1[2]`) and slicing.
- Use `len(t1)` to obtain the length of a tuple.
- The universal immutable sequence type operations are all supported by tuples.
  - +, *
  - in, not in
  - min(t), max(t), t.index(x), t.count(x)

PACKING/UNPACKING

- Tuple packing is used to "pack" a collection of items into a tuple. We can unpack a tuple using Python's multiple assignment feature.

```
>>> s = "Susan", 19, "CS" # tuple packing
>>> name, age, major = s # tuple unpacking
>>> name
'Susan'
>>> age
19
>>> major
'CS'
```

WHEN TO USE DICTIONARIES

- When you need to create associations in the form of key:value pairs.
- When you need fast lookup for your data, based on a custom key.
- When you need to modify or add to your key:value pairs.

CONSTRUCTING A DICTIONARY

- Create an empty dictionary with empty curly braces or the dict() constructor.
- You can initialize a dictionary by specifying each key:value pair within the curly braces.
- Note that keys must be *hashable* objects.

```
>>> d1 = {}
>>> d2 = dict() # both empty
>>> d3 = {"Name": "Susan", "Age": 19, "Major": "CS"}
>>> d4 = dict(Name="Susan", Age=19, Major="CS")
>>> d5 = dict(zip(['Name', 'Age', 'Major'], ["Susan", 19, "CS"]))
>>> d6 = dict([('Age', 19), ('Name', "Susan"), ('Major', "CS")])
```

Note: zip takes two equal-length collections and merges their corresponding elements into tuples.

ACCESSING THE DICTIONARY

- To access a dictionary, simply index the dictionary by the key to obtain the value. An exception will be raised if the key is not in the dictionary.

```
>>> d1 = {'Age':19, 'Name':"Susan", 'Major':"CS"}
>>> d1['Age']
19
>>> d1['Name']
'Susan'
```

UPDATING A DICTIONARY

- Simply assign a key:value pair to modify it or add a new pair. The del keyword can be used to delete a single key:value pair or the whole dictionary. The clear() method will clear the contents of the dictionary.

```
>>> d1 = {'Age':19, 'Name':"Susan", 'Major':"CS"}
>>> d1['Age'] = 21
>>> d1['Year'] = "Junior"
>>> d1
{'Age': 21, 'Name': 'Susan', 'Major': 'CS', 'Year': 'Junior'}
>>> del d1['Major']
>>> d1
{'Age': 21, 'Name': 'Susan', 'Year': 'Junior'}
>>> d1.clear()
>>> d1
{}
```

BUILT-IN DICTIONARY METHODS

```
>>> d1 = {'Age':19, 'Name':"Susan", 'Major':"CS"}
>>> d1.has_key('Age') # True if key exists
True
>>> d1.has_key('Year') # False otherwise
False
>>> d1.keys() # Return a list of keys
['Age', 'Name', 'Major']
>>> d1.items() # Return a list of key:value pairs
[('Age', 19), ('Name', 'Susan'), ('Major', 'CS')]
>>> d1.values() # Returns a list of values
[19, 'Susan', 'CS']
```

Note: `in`, `not in`, `pop(`*key*`)`, and `popitem()` are also supported.

ORDERED DICTIONARY

Dictionaries do not remember the order in which keys were inserted. An ordered dictionary implementation is available in the collections module. The methods of a regular dictionary are all supported by the OrderedDict class.

An additional method supported by OrderedDict is the following:

```
OrderedDict.popitem(last=True)   # pops items in LIFO order
```

ORDERED DICTIONARY

```
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

>>> # dictionary sorted by value
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])

>>> # dictionary sorted by length of the key string
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2), ('banana', 3)])
```