Structure Types, Mutability and Higher-Order Functions

Chapter 5

Types so far

- `int`
- `float`
- `str`
- `bool`
  - We have hinted at this with True and False

Type definitions

- Two axes for types
  - Scalar or non-scalar
  - Mutable or immuable

Scalar v Non-Scalar

- Scalar
  - Can be subdivided into meaningful parts
  - A string can be broken into substrings – down to individual characters
- Non-Scalar
  - Can not be subdivided
    - Python doesn’t support partial truth (Boolean)
    - Numeric values can have math operations performed but can’t themselves be broken up

Mutability

Immutability

- So far the types we have used are **immutable**
  - `int`
  - `float`
  - `bool`
  - `str`
- Object that are **immutable** can not be changed
  - `i = 1`
  - `1 = 2`
  - `i++`
  - `x = 'abcdef'`
  - `x = x[2:4]`
  - `x[1] = 'd'`

Mutability

- A **mutable** object can be changed
- It is not necessary to create a new object and assign a value
- Remember a variable is “just a name”
- More when we get to **lists**

Tuples

- Immutable
  - Once created it can not be change
- Sequence of objects
  - Unlike a string it can contain a variety of types

- Creation
  - Comma separated list of objects enclosed by parentheses

```
t_empty = ()
t2 = (1, 'two', True)
```

More Tuple-ing

- Singleton tuple (containing one object)

```
t_singleton = (1, )
t = (1)  # this is just an integer assignment
¯\_(ツ)_/¯
```

Tuple Operations

- We can use many of the same operations as strings

```
t_main = (1, 'two', True)
print (t_main)
   (1, 'two', True)
print(len(t_main))
   3
print(t_main + t_main)
   (1, 'two', True, 1, 'two', True)
print(3 * t_main)
   (1, 'two', True, 1, 'two', True, 1, 'two', True)
print(t_main + 13)
   TypeError: can only concatenate tuple (not "int") to tuple
```

Tuple Slicing

```
t = ('abc', 123, 12.5, 'Hello Kitty')
print(t[1:3])
   (123, 12.5)
print(t[:2])
   ('abc', 123)
print(t[2:])
   (12.5, 'Hello Kitty')
print(t[1:2])
   (12.5, )
print(t[2])
   12.5
t[1] = 'abc'
   TypeError: 'tuple' object does not support item assignment
```

Tuple Nesting

```
t1 = (1, 'two', 3)
t2 = (t1, 3.25)
print(t2)
    ((1, 'two', 3), 3.25)
print((t1 + t2))
   (1, 'two', 3, (1, 'two', 3), 3.25)
print((t1 + t2)[3])
   (1, 'two', 3)
print((t1 + t2)[2:5])
   (3, (1, 'two', 3), 3.25)
```

Foreign Tuples

```
def intersect(t1, t2):
    """ Assumes t1 and t2 are tuples
        returns a tuple of all elements in both
tuples"""
    result = ()
    for e in t1:
        if e in t2:
            result += (e,)
    return result
```

- **For** and **in** == ‘foreign’ # get it?

Multiple Assignment

```
x, y = y, x
a, b, c = ‘abc’
```

- We can do the same thing with a tuple of known size

```
a, b, c = (1, ‘two’, 3)
```

- Most commonly used with functions

```
def tuple_me(a, b, c):
    return a, b, c

print(tuple_me(1, 515, 'abc'))
```

Ranges

- We already saw ranges with the `for` keyword
- Immutable objects
- Can do all the same operations except concatenation and repetition
- Must use the keyword range

```
r = range(0, 10, 2)
print(r)
     range(0, 10, 2)
print(r[2])
     4
print(r[2:4])
     range(4, 8, 2)
```

Lists

- Our first **mutable** type **YAY!**
- An ordered sequence of objects
  - Can have different types
  - Created using square brackets []
  - Can perform many of the same operations as on tuples
  - Singleton lists still need that trailing comma
    - `l_singleton = [‘One is the loneliest number’, ]`

Sample List Operations

```
l_main = [1, 'two', True]
print (l_main)
     [1, 'two', True]
print(len(l_main))
     3
print(l_main + l_main)
     [1, 'two', True, 1, 'two', True]
print(3 * l_main)
     [1, 'two', True, 1, 'two', True, 1, 'two',
True]
```

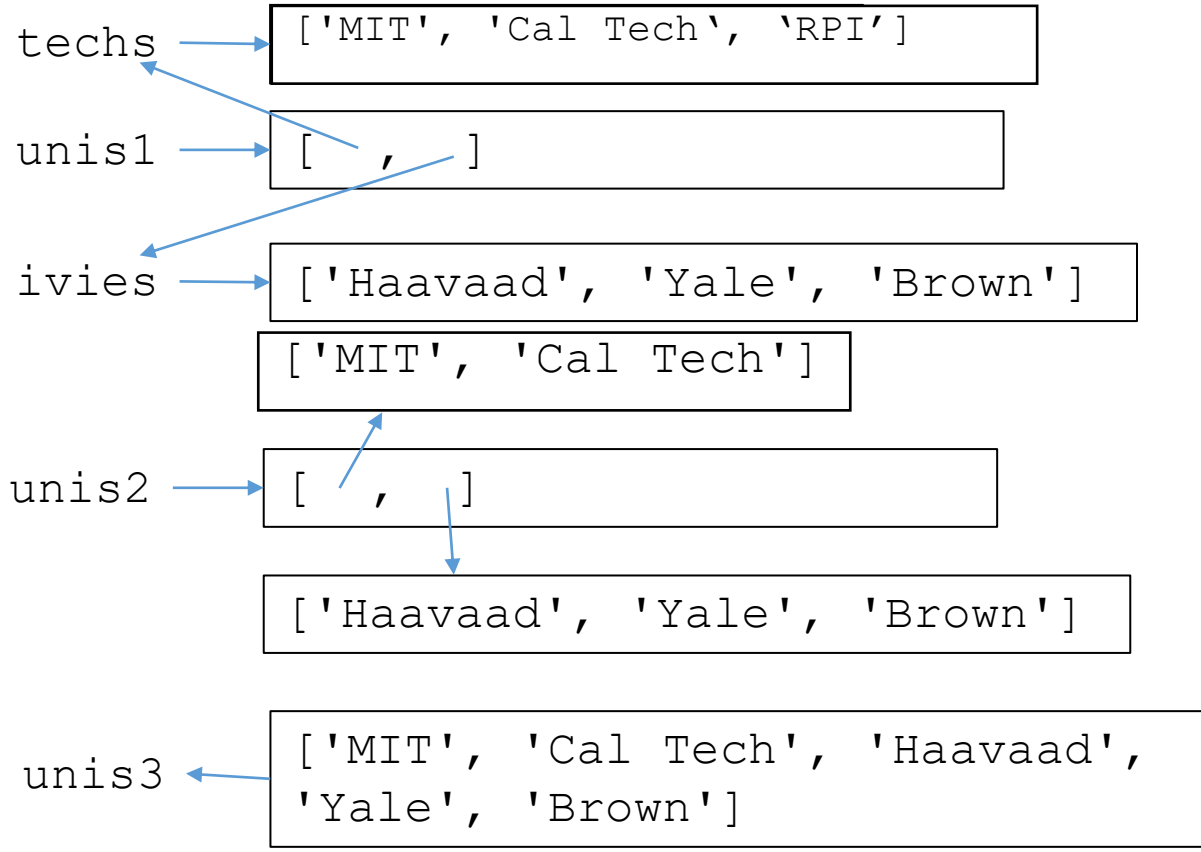
Mutating a list

```
l1[1] = "one"
print(l1)
```

```
[1, 'one', True]
```

Assignment v Mutation

```
techs = ['MIT', 'Cal Tech']
ivies = ['Haavaad', 'Yale', 'Brown']
unis1 = [techs, ivies]
unis2 = [['MIT', 'Cal Tech'],
['Haavaad', 'Yale', 'Brown']]
unis3 = techs + ivies
print(unis1 == unis2)
print(id(unis1) == id(unis2))
techs.append('RPI')
print(techs)
print(unis1)
print(unis2)
print(unis3)
```



List Aliasing

- Two variables point to the same object
  - techs ⇔ unis1[0]
  - ivies ⇔ unis1[1]
- Any change through one “path” affects both values

Append, Extend and Concatenate

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = list1 + list2    # New object
list1.extend(list2)  # Adds elements from list2 to list1
print(list1)
      [1, 2, 3, 4, 5, 6]
list1.append(list2)  # Adds list list2 as an element to list1
print(l1)
      [1, 2, 3, 4, 5, 6, [4, 5, 6]]
print(l3)
      [1, 2, 3, 4, 5, 6]
```

Some More List Functions

- **L.count(e)** returns the number of times that e occurs in L.
- **L.insert(i, e)** inserts the object e into L at index i.
- **L.remove(e)** deletes the first occurrence of e from L.
- **L.index(e)** returns the index of the first occurrence of e in L. It raises an exception (see Chapter 7) if e is not in L.
- **L.pop(i)** removes and returns the item at index i in L. If i is omitted, it defaults to -1, to remove and return the last element of L.
- **L.sort()** sorts the elements of L in ascending order.
- **L.reverse()** reverses the order of the elements in L.

For your consideration

```
def removeDups(L1, L2):
    """Assumes that L1 and L2 are lists.
       Removes any element from L1 that also occurs in L2"""
    for e1 in L1:
        if e1 in L2:
            L1.remove(e1)

L1 = [1,2,3,4]
L2 = [1,2,5,6]
removeDups(L1, L2)
print('L1 =', L1)
```

```
L1 = [2, 3, 4]
```

How Then Do I Clone a List?

```
# This is an alias not a clone
newList = l1


# Full slice is a clone!
newList = l1[:]
# Another way to clone
newList = list(l1)
```

Deep Copy

- What about complex lists (lists of lists …)

```
import copy

L1 = [1, 2, 3]
L2 = [4, 5, 6]
deep_list = [l1, l2]
shallow_copy = deep_list[:]
deep_copy = copy.deepcopy(deep_list)
l2.append('dorf')
print('Shallow Copy: ', shallow_copy)
         [[1, 2, 3], [4, 5, 6, 'dorf']]
print('Deep Copy:    ', deep_copy)
        [[1, 2, 3], [4, 5, 6]]
```

List Comprehension

- Applies an operation to all elements in a sequence
  - returns a new list

```
L2 = [x**2 for x in range(1,7)]
[1, 4, 9, 16, 25, 36]
```

- What if types are mixed?

```
mixed = [1, 2, 'a', 3, 4.0]
L2 = [x**2 for x in mixed]
TypeError: unsupported operand type(s) for ** or
pow(): 'str' and 'int'
```

- List Comprehension can include conditionals

```
L2 = [x**2 for x in mixed if type(x) == int]
```

Functions as Objects (5.4)

- Functions are **first-class objects** in Python
  - Functions can be used like objects of other types
  - We have seen a function assignment while talking about scope
  - When a function is passed as an argument to another function we use the term **higher-order programming**

Function as an argument example

```
def applyToEach(L, f):
    """Assumes L is a list, f a function
       Mutates L by replacing each element, e, of L by f(e)"""
    for i in range(len(L)):
        L[i] = f(L[i])

L = [1, -2, 3.33]
print('L =', L)
print('Apply abs to each element of L.')
applyToEach(L, abs)
print('L =', L)
print('Apply int to each element of', L)
applyToEach(L, int)
print('L =', L)
```

Python **`map`** function

- More powerful than applyToEach
- First argument is a function
- Next *n* arguments map to the arguments of the first function

```
L1 = [1, 28, 36]
L2 = [2, 57, 9]
print map(min, L1, L2)
```

Lambda expressions

- Anonymous (unnamed) functions
- Frequently used as arguments to higher order function

```
L = []
for i in map(lambda x, y: x**y, [1, 2, 3, 4],
[3, 2, 1, 0]):
    L.append(i)
print(L)
```

Strings, Tuples, Ranges and Lists (5.5)

Operations on sequences

- **seq[i]** returns the ith element in the sequence.
- **len(seq)** returns the length of the sequence.
- **seq1 + seq2** returns the concatenation of the two sequences.
- **n * seq** returns a sequence that repeats seq n times (not available for ranges).
- **seq[start:end]** returns a slice of the sequence.
- **e in seq** is True if e is contained in the sequence and False otherwise.
- **e not in seq** is True if e is not in the sequence and False otherwise.
- **for e in seq** iterates over the elements of the sequence.

Strings, Tuples, Ranges and Lists

Type Comparison

Type	Type of Elements	Examples of Literals	Mutable
`str`	Characters	‘’, ‘a’, ‘abc’	No
`tuple`	Any type	(), (3,), (3, 4, ‘abc’)	No
`range`	Integers	range(10), range(0,10,2)	No
`list`	Any Type	[], [3,], [3, 4, ‘abc’]	Yes

Strings, Tuples, Ranges and Lists
Some methods on Strings

- **S.count(s1)** counts how many times the string s1 occurs in s.
- **S.find(s1)** returns the index of the first occurrence of the substring s1 in s, and -1 if s1 is not in s.
- **S.rfind(s1)** same as find, but starts from the end of s (the “r” in rfind stands for reverse).
- **S.index(s1)** same as find, but raises an exception (see Chapter 7) if s1 is not in s.
- **S.rindex(s1)** same as index, but starts from the end of s.
- **S.lower()** converts all uppercase letters in s to lowercase.
- **S.replace(old, new)** replaces all occurrences of the string old in s with the string new.
- **S.rstrip()** removes trailing white space from s.
- **S.split(d)** Splits s using d as a delimiter. Returns a list of substrings of s. For example, the value of 'David Guttag plays basketball'.split(' ') is ['David', 'Guttag', 'plays', 'basketball']. If d is omitted, the substrings are separated by arbitrary strings of whitespace characters (space, tab, newline, return, and formfeed).

Sets {}

- Mutable **unordered** sequence
  - Can’t be indexed or sliced
  - Each element is unique
  - Common mathematics set operations
    - in
    - union |
    - Intersection &
    - difference –
    - issubset <=
    - issuperset >=
  - Designated by curly braces

Sets

- Elements must be “hashable”
  - Python scalar types
  - String
  - User objects with `__hash__` and `__eq__` implemented

Dictionaries

- The Python type **`dict`** creates a sequence of key/value pairs
  - Created using curly braces {}
  - Key:Value
  - Dictionaries are **mutable**
  - Ordering is transparent to the user
    - No index, slice or range operations
  - Elements are retrieved by the key

```
monthNumbers = {'Jan':1, 'Feb':2, 'Mar':3,
                'Apr':4, 'May':5,
                1:'Jan', 2:'Feb', 3:'Mar',
                4:'Apr', 5:'May'}
print('The third month is ' + monthNumbers[3])
dist = monthNumbers['Apr'] - monthNumbers['Jan']
print('Apr and Jan are', dist, 'months apart')
```

How are dictionaries stored

- Not indexed?
- The key must be **hashable**
  - Converted to an integer value through some sort of function
- Iterating through a dictionary returns the keys
- Assess a value by using its key

- Strings and tuples are often used for keys
- Values can be complex objects

Some common operations on Dictionaries

- **len(d)** returns the number of items in d.
- **d.keys()** returns a list containing the keys in d.
- **d.values()** returns a *view* of the values in d.
- **d.items()** returns a *view* of (key, value) pairs in d
- **k in d** returns True if key k is in d.
- **d[k]** returns the item in d with key k.
- **d.get(k, v)** returns D[k] if k is in D, and v otherwise.
- **d[k] = v** associates the value v with the key k in d. If there is already a value associated with k, that value is replaced.
- **del d[k]** removes the key k from d.
- **for k in d** iterates over the keys in d.

Dictionary Comprehension

- Like list comprehension
- *Except*
  - {}
  - Iterates key, value pairs

- `{key: value for id1, id2 in iterable` *`if test`*`}`