Data Structures

- Sequence types
- Using sequences
- Set types
- Mapping types
- Additional techniques
- Worked examples

1. Sequence Types

- Overview
- Lists
- Splitting and joining
- Tuples
- Ranges

Overview

.

Basic sequence types

- List, tuple, and range

Text sequence types

- String

Binary sequence types

- bytes, bytesarray, and memoryview

Lists

There are several ways to create a list

- []
- [item, item, item …]
- list()
- list(iterable)

Example:

```
1   list1 = []
2   list2 = ["Italy", "France", "Spain"]
3   list3 = [3, 12, 19, 1, 2, 7]
4   list4 = list()
5   list5 = list(list3)
6   list6 = list("Hello")
7
8   print("list1 has %d items: %s" % (len(list1), list1))
9   print("list2 has %d items: %s" % (len(list2), list2))
10  print("list3 has %d items: %s" % (len(list3), list3))
11  print("list4 has %d items: %s" % (len(list4), list4))
12  print("list5 has %d items: %s" % (len(list5), list5))
13  print("list6 has %d items: %s" % (len(list6), list6))
```

Splitting and Joining

A common scenario where lists crop up in Python is when you call split() or join() on a string

- split() - splits a string into a list of substrings
- join() - joins a list into a concatenated string

Example:

```
str = "and we were singing, hymns and arias, land of my fathers, ar hyd yr nos"

words = str.split(", ")

lines = " ...\n".join(words)

print("%s" % lines)
```

Tuples

There are several ways to create a tuple

- ()
- a, or (a,)
- a,b,c or (a,b,c)
- tuple()
- tuple(iterable)

Example:

```
1   tuple1 = ()
2   tuple2 = "Norway",      # or:  tuple2 = ("Norway",)
3   tuple3 = 3, 19, 2       # or:  tuple3 = (3, 19, 2)
4   tuple4 = tuple()
5   tuple5 = tuple(tuple3)
6
7   print("tuple1 has %d items: %s" %   (len(tuple1), tupl
8   print("tuple2 has %d item(s): %s" % (len(tuple2), tupl
9   print("tuple3 has %d items: %s" %   (len(tuple3), tupl
10  print("tuple4 has %d items: %s" %   (len(tuple4), tupl
11  print("tuple5 has %d items: %s" %   (len(tuple5), tupl
```

Ranges

To create a range, use the range constructor

- range(stop)
- range(start, stop)
- range(start, stop, step)

Example:

```
1   def display_range(msg, r):
2       print("\n" + msg)
3       for i in r:
4           print(i)
5
6   range1 = range(5)
7   range2 = range(5,10)
8   range3 = range(5,10,2)
9
10  display_range("range1", range1)
11  display_range("range2", range2)
12  display_range("range3", range3)
```

2. Using Sequences

- Common sequence operations
- Slicing operations
- Unpacking operations
- Sequence modification operations
- Optional exercise

Common sequence operations

You can perform these operations on any sequence:

```
euro = ["GB", "ES", "NL", "F", "D", "I", "P"]
asia = ["SG", "JP"]

print("%s" % "P" in euro)              # True
print("%s" % "F" not in euro)          # False
print("%s" % (euro + asia))            # ['GB', 'ES', 'NL', 'F', 'D', 'I', 'P', 'SG', 'JP']
print("%s" % (asia * 2))               # ['SG', 'JP', 'SG', 'JP']
print("%s" % (2 * asia))               # ['SG', 'JP', 'SG', 'JP']
print("%d" % len(euro))                # 7
print("%s" % min(euro))                # D
print("%s" % max(euro))                # P
print("%d" % euro.index("NL"))         # 2
print("%d" % euro.index("NL", 1))      # 2
print("%d" % euro.index("NL", 1, 4)    # 2
print("%d" % euro.count("ES"))         # 1
```

Slicing operations

```
1   euro = ["GB", "ES", "NL", "F", "D", "I", "P"]
2   asia = ["SG", "JP"]
3
4   print("%s" % (euro[1]))                 # ES
5   print("%s" % (euro[1:5]))               # ['ES', 'NL', 'F', 'D']
6   print("%s" % (euro[1:5:2]))             # ['ES', 'F']
7   print("%s" % (euro[3:]))                # ['F', 'D', 'I', 'P']
8   print("%s" % (euro[:-3]))               # ['GB', 'ES', 'NL', 'F']
```

Unpacking operations

You can unpack (i.e. extract) elements in a sequence

The following example illustrates the techniques available

```
euro = ["GB", "ES", "NL", "F"]

# Manually getting items.
a, b, c, d = euro[0], euro[1], euro[2], euro[3]
print("%s %s %s %s" % (a, b, c, d))    # GB ES NL F

# Unpacking.
e, f, g, h = euro
print("%s %s %s %s" % (e, f, g, h))    # GB ES NL F

# Catch-all unpacking.
i, j, *k = euro
print("%s %s %s" % (i, j, k))          # GB ES ['NL', 'F']
```

Sequence modification operations

You can perform these operations on a mutable sequence such as a list:

```
euro = ["GB", "ES", "NL", "F"]

euro[0] = "CY"
euro[1:3] = ["US", "AU", "AT"]
euro.append("SW")
euro.extend(["YU", "ZR"])
euro.insert(1, "NI")
print("%s" % euro)      # ['CY', 'NI', 'US', 'AU', 'AT', 'F', 'SW', 'YU', 'ZR']

euro.pop()
euro.pop(1)
del euro[2:4]
print("%s" % euro)      # ['CY', 'US', 'F', 'SW', 'YU']

euro.remove("US")
euro.reverse()
print("%s" % euro)      # ['YU', 'SW', 'F', 'CY']

eurocopy = euro.copy()
euro.clear()
print("%s" % eurocopy)  # ['YU', 'SW', 'F', 'CY']
print("%s" % euro)      # []
```

Exercise

Write a Python program as follows:

- Ask the user to enter a series of numbers (-1 to quit)
- Determine which numbers are prime
- Display the prime numbers on the console

For the solution code: See Solutions\05-DataStructures\primes.py

Here are more detailed instructions for this exercise:

1. Write a function named get_numbers(). The function should loop around, asking the user to enter a number. For each number, add it to a list. Stop looping when the user enters -1. Return the list at the end of the function.

2. Write a function named find_primes(). The function takes one argument - a list of numbers. The function should loop through the numbers, to find the prime numbers. The function should return the prime numbers.

3. Write a function named display_numbers(). The function takes one argument - a list of numbers. The function displays the numbers on the screen.

Call these functions from your "main" code, to get numbers from the user, find which ones are prime,

3. Set Types

- Creating a set
- Creating a frozen set
- Common set operations
- Set modification operations

Creating a set

There are several ways to create a set

- {item, item, item, … }
- set()
- set(iterable)
- Via a comprehension, similar to lists

Example:

```
1   set1 = {"dog", "ant", "bat", "cat", "dog"}
2   set2 = set()
3   set3 = set(("dog", "ant", "bat", "cat", "dog"))
4   set4 = set("abracadabra")
5   set5 = {c.upper() for c in "abracadabra"}
6
7   print("set1 has %d items: %s" % (len(set1), set1))
8   print("set2 has %d items: %s" % (len(set2), set2))
9   print("set3 has %d items: %s" % (len(set3), set3))
10  print("set4 has %d items: %s" % (len(set4), set4))
11  print("set5 has %d items: %s" % (len(set5), set5))
```

Creating a frozen set

Creating a frozenset is similar to creating a set

- Use the frozenset constructor

Example:

```
1   set1 = frozenset({"dog", "ant", "bat", "cat", "dog"})
2   set2 = frozenset()
3   set3 = frozenset(("dog", "ant", "bat", "cat", "dog"))
4   set4 = frozenset("abracadabra")
5   set5 = frozenset({c.upper() for c in "abracadabra"})
6
7   print("set1 has %d items: %s" % (len(set1), set1))
8   print("set2 has %d items: %s" % (len(set2), set2))
9   print("set3 has %d items: %s" % (len(set3), set3))
10  print("set4 has %d items: %s" % (len(set4), set4))
11  print("set5 has %d items: %s" % (len(set5), set5))
```

Common set operations

You can perform these operations on any set:

```
s1 = {"GB", "US", "SG"}
s2 = {"GB", "US", "AU"}
s3 = {"F", "BE", "CA"}

print("%s" % ("GB" in s1))          # True
print("%s" % ("GB" not in s1))      # False
print("%s" % (s1.isdisjoint(s2)))   # False
print("%s" % (s1.isdisjoint(s3)))   # True
print("%s" % (s1.issubset(s2)))     # False
print("%s" % (s1 ⩽ s2))             # False
print("%s" % (s1 < s2))             # False
print("%s" % (s1.issuperset(s2)))   # False
print("%s" % (s1 ⩾ s2))             # False
print("%s" % (s1 > s2))             # False
print("%s" % (s1.union(s2, s3)))    # {'GB', 'US', 'BE', 'F', 'CA', 'AU', 'SG'}
print("%s" % (s1 | s2 | s3))        # {'GB', 'US', 'BE', 'F', 'CA', 'AU', 'SG'}
print("%s" % (s1.difference(s2, s3))) # {'SG'}
print("%s" % (s1 - s2 - s3))          # {'SG'}
print("%s" % (s1.symmetric_difference(s2))) # {'AU', 'SG'}
print("%s" % (s1 ^ s2))                     # {'AU', 'SG'}
```

Set modification operations

You can perform these operations on a mutable set:

```
s1.add("HK")
s1.remove("US")
s1.discard("D")
print("%s" % s1)       # {'SG', 'HK', 'GB'}

print("%s" % s1.pop()) # SG
print("%s" % s1)       # {'HK', 'GB'}

s1.update(s2,s3)
s1 |= s4 | s5
print("%s" % s1)       # {'D', 'AU', 'US', 'I', 'F', 'P', 'N', 'GB', 'CA', 'HK'}

s1.intersection_update(s2,s3)
s1 &= s4 & s5
print("%s" % s1)       # {'GB', 'US'}

s1.difference_update({"AA", "BB"},{"CC", "GB"})
s1 -= {"DD", "EE"} | {"FF", "GG"}
print("%s" % s1)       # {'US'}

s1.symmetric_difference_update(s2)


```

4. Mapping Types

- Creating a dictionary
- Iterating over a dictionary
- Accessing items in a dictionary

Creating a dictionary

There are several ways to create a dict

- {key:value, key:value, … }
- dict()
- dict(anotherDict)
- dict(keyword=value, keyword=value, …)
- dict(zip(keysIterable, valuesIterable))

Example:

```
1   dict1 = {"us":"+1", "nl":"+31", "no":"+47"}
2   dict2 = dict()
3   dict3 = dict({"us":"+1", "nl":"+31", "no":"+47"})
4   dict4 = dict(us="+1", nl="+31", no="+47")
5   dict5 = dict(zip(["us", "nl", "no"], ["+1", "+31", "+4
6
7   print("dict1 has %d items: %s" % (len(dict1), dict1))
8   print("dict2 has %d items: %s" % (len(dict2), dict2))
9   print("dict3 has %d items: %s" % (len(dict3), dict3))
10  print("dict4 has %d items: %s" % (len(dict4), dict4))
11  print("dict5 has %d items: %s" % (len(dict5), dict5))
```

Iterating over a dictionary

There are several ways to iterate over a dict

- Iterate over the items (i.e. key-value pairs)
- Iterate over the keys
- Iterate over the values

Example:

```
1   dialcodes = {"us": "+1", "nl": "+31", "no": "+47"}
2
3   print("Items:")
4   for k,v in dialcodes.items():
5       print(k, v)
6
7   print("\nKeys:")
8   for k in dialcodes.keys():
9       print(k)
10
11  print("\nValues:")
12  for v in dialcodes.values():
13      print(v)
```

Accessing items in a dictionary

There are various operations for accessing items in a dict

```
dialcodes = {"us": "+1", "nl": "+31", "no": "+47", "it": "+39"}

print("%s" % "us" in dialcodes)          # True
print("%s" % "us" not in dialcodes)      # False

dialcodes["uk"] = "+44"
print(dialcodes["uk"])                   # +44
print(dialcodes.get("fr"))               # None
print(dialcodes.get("fr", "xxx"))        # xxx

del dialcodes["no"]
print(dialcodes.pop("uk"))               # +44
print(dialcodes.pop("uk", "xxx"))        # xxx
print(dialcodes.setdefault("it", "???")) # ???

dialcodes.update({"ca":"+1", "it":"+39"})
print(dialcodes)  # {'ca': '+1', 'us': '+1', 'nl': '+31'}
```

5. Additional Techniques

- Generators
- List comprehensions
- Set comprehensions
- Dictionary comprehensions
- Filtering, sorting, and mapping
- Working with JSON data

Generators

A generator is a special kind of function that returns a collection, one item at a time

- Use the yield keyword to yield the next value on each call

Example - consider the following two functions

- The 1st version returns a collection "all at once"
- The 2nd version yields a collection one element at a time

```
def getNums():
    nums = []
    while True:
        num = int(input("Number? "))
        if num == -1 :
            break
        nums.append(num)
    return nums

# Client code.
nums = getNums()
for n in nums:
    print("  %d" % n)
```

```
def getNumsB():
     while True:
        num = int(input("Number? "))
        if num == -1 :
            break
        yield num



# Client code.
nums = getNums()
for n in nums:
    print("  %d" % n)
```

List comprehensions

You can create a list from another sequence

- Apply an operation on all the items in an existing sequence
- This is known as a "list comprehension"

Example:

```
squares = [x**2 for x in range(6)]

ftemps = [32, 68, 212]
ctemps = [(f-32)*5/9 for f in ftemps]

print("squares: %s" % squares)
print("ftemps:  %s" % ftemps)
print("ctemps:  %s" % ctemps)
```

Set comprehensions

You can also create a "set comprehension"

- i.e. a set created from another sequence

Example:

```
ftemps = range(0, 50, 5)
ctemps = { int((f-32)*5/9) for f in ftemps }

print("ctemps:  %s" % ctemps)
```

Dictionary comprehensions

You can also create a "dictionary comprehension"

- i.e. a collection of key/value pairs created from another sequence

Example:

```
1   mydict = { i : i*i for i in range(1, 6) }
2
3   print("mydict:  %s" % mydict)
```

Filtering, sorting, and mapping (1/2)

Python defines functions that allow you to filter, sort, and map (i.e. transform) the elements in a collection

Example

```
names = ["Zak", "Tim", "Ben", "Joe", "Kim", "Bud", "Te

bnames = list(filter(startsWithB, names))
print(bnames)

sortedBnames = sorted(bnames)
print(sortedBnames)

mappedSortedBnames = list(map(topAndTail, sortedBnames
print(mappedSortedBnames)
```

```
def startsWithB(element):
  if len(element) and element[0] == 'B':
    return True
  else:
    return False
```

```
def topAndTail(element):
    return "***" + element + "***"
```

Filtering, sorting, and mapping (2/2)

The sorted() function takes two optional arguments, which allow you to take control over the sorting

- key - function that indicates what aspect to sort items on
- reverse - boolean (default is false, i.e. ascending order)

Example

```
1   names = ["Andy", "Jayne", "Em", "Tom"]
2
3   sortedNamesAlphabetically = sorted(names)
4   print(sortedNamesAlphabetically)
5
6   sortedNamesByLength = sorted(names, key=personNameLength)
7   print(sortedNamesByLength)
8
9   sortedNamesByLengthDescending = sorted(names, key=personNameLength, reverse=True)
10  print(sortedNamesByLengthDescending)
```

Working with JSON data (1/3)

JSON is a popular string data format

- Typically used for passing data to/from REST services
- Very easy to read/write JSON data in JavaScript (and in Python ☺)

Here are some example JSON strings:

```
personJson = '{ "name": "Andy", "age": 21, "height": 1.67, "isWelsh": true }'
coordsJson = '[ { "x": 100, "y": 150 }, { "x": 200, "y": 250 } ]'
```

To read/write JSON data in Python, use the standard Python module named json

- json.loads() loads JSON data into a Python dictionary/list
- json.dumps() dumps a Python dictionary/list into a JSON string

Working with JSON data (2/3)

These examples show how to load JSON data into Python data structures

```
import json

personJson = '{"name": "Andy", "age": 21, "height": 1.67, "isWelsh": true }'

person = json.loads(personJson)

print("%s is %d years old" % (person["name"], person["age"]))
print("Height is %.2f, Welshness is %s" % (person["height"], person["isWelsh"]))
```

```
import json

coordsJson = '[ { "x": 100, "y": 150 }, { "x": 200, "y": 250 } ]'

coords = json.loads(coordsJson)

print("Point 0 is %d, %d" % (coords[0]["x"], coords[0]["y"]))
print("Point 1 is %d, %d" % (coords[1]["x"], coords[1]["y"]))
```

Also see readJsonFromFile.py and sampledata.json

Working with JSON data (3/3)

These examples show how to dump Python data into a JSON string

```
import json

person = {"name": "Andy", "age": 21, "height": 1.67, "isWelsh": True }

personJson = json.dumps(person, indent=4)

print(personJson)
```

```
import json

coords = [ { "x": 100, "y": 150 }, { "x": 200, "y": 250 } ]

coordsJson = json.dumps(coords, indent=4)

print(coordsJson)
```