Python Programming
Iterables

Iterables

- Types of iterables available in Python.
- Lists.
- Tuples.
- Ranges.
- Dictionaries.
- Sets.
- The *args and **kwargs parameters.

Introduction

- Iterables are objects that can return their members one at a time. The iterables that will be covered are:
  - lists
  - tuples
  - ranges
  - dictionaries
  - sets.

Definitions

1. Sequences are iterables that can return members based on their position within the iterable. Examples of sequences are strings, lists, tuples, and ranges

2. Lists are mutable (changeable) sequences similar to arrays in other programming languages.

3. Tuples are immutable sequences.

4. Ranges are immutable sequences of numbers often used in for loops.

5. Dictionaries are mappings that use arbitrary keys to map to values. Dictionaries are like associative arrays in other programming languages.

6. Sets are mutable unordered collections of distinct immutable objects. So, while the set itself can be modified, it cannot be populated with objects that can be modified.

Sequences

- String is a sequence:

```
>>> 'Hello, world!'[1]
'e'
```

- Other sequences are:
  - lists
  - tuples
  - ranges

Lists

- Like arrays:

```
colors = ["red", "blue", "green", "orange"]
```

- methods:
  - mylist.append(x) – Appends x to mylist.
  - mylist.remove(x) – Removes first element with value of x from mylist. Errors if no such element is found.
  - mylist.insert(i, x) – Inserts x at position i.
  - mylist.count(x) – Returns the number of times that x appears in mylist.
  - mylist.index(x) – Returns the index position of the first element in mylist whose value is x or a ValueError if no such element exists.
  - mylist.sort() – Sorts mylist.
  - mylist.reverse() – Reverses the order of mylist.
  - mylist.pop(n) – Removes and returns the element at position n in mylist. If n is not passed in, the last element in the list is popped (removed and returned).
  - mylist.clear() – Removes all elements from mylist.
  - mylist.copy() – Returns a copy of mylist.
  - mylist.extend(anotherlist) – Appends anotherlist onto mylist.

Lists

```
>>> colors = ["red", "blue", "green",
"orange"]
>>> colors
['red', 'blue', 'green', 'orange']
>>> colors.append("purple") # Append
purple to colors
>>> colors
['red', 'blue', 'green', 'orange',
'purple']
>>> colors.remove("green") # Remove
green from colors
>>> colors
['red', 'blue', 'orange', 'purple']
>>> colors.insert(2, "yellow") # Insert
yellow in position 2
>>> colors
['red', 'blue', 'yellow', 'orange',
'purple']
>>> colors.index("orange") # Get
position of orange
3
```

```
>>> colors.sort() # Sort colors in
place
>>> colors
['blue', 'orange', 'purple', 'red',
'yellow']
>>> colors.reverse() # Reverse order
of colors
>>> colors
['yellow', 'red', 'purple', 'orange',
'blue']
>>> colors.pop() # Remove and return
last element
'blue'
>>> colors.pop(1) # Remove and return
element at position 1
'red'
>>> colors # Notice blue and red have
been removed
['yellow', 'purple', 'orange']
```

Lists

```
>>> colors_copy = colors.copy() #
Create a copy of colors
>>> colors_copy
['yellow', 'purple', 'orange']
>>> colors.extend(colors_copy) #
Append colors_copy to colors
>>> colors
['yellow', 'purple', 'orange',
'yellow', 'purple', 'orange']
>>> colors_copy.clear() # Delete all
elements from colors_copy
>>> colors_copy # Notice colors_copy
is now empty
[]
```

```
>>> del colors_copy # Delete
colors_copy
>>> colors_copy # It's gone
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'colors_copy' is not
defined
```

Copying a list

```
>>> colors = ["red", "blue", "green", "orange"]
>>> colors_copy = colors.copy()
>>> colors_copy
['red', 'blue', 'green', 'orange']
>>> colors_copy.sort()
>>> colors_copy
['blue', 'green', 'orange', 'red']
>>> colors
['red', 'blue', 'green', 'orange'] # colors remains unsorted
```

- colors remains the same

Copying a list

```
>>> colors_copy2 = colors
>>> colors_copy2
['red', 'blue', 'green', 'orange']
>>> colors_copy2.sort()
>>> colors_copy2
['blue', 'green', 'orange', 'red']
>>> colors
['blue', 'green', 'orange', 'red'] # Wait! colors is sorted too!
>>> id(colors)
2147162345152
>>> id(colors_copy)
2147163702400 # different id than colors
>>> id(colors_copy2)
2147162345152 # the same id as colors
```

- colors_copy2 is a pointer to colors

Deleting list elements

- del
- Demo 26: iterables/Demos/del_list.p

Sequences and Random

- random.choice(seq)

```
>>> import random
>>> colors = ["red", "blue", "green", "orange"]
>>> random.choice(colors)
'orange'
>>> random.choice(colors)
'green'
```

- random.shuffle(seq)

```
>>> import random
>>> colors = ["red", "blue", "green", "orange"]
>>> random.shuffle(colors)
>>> colors
['green', 'red', 'blue', 'orange']
```

Exercise 16

- Remove and Return Random Element
  - 10-20 min

Tuples

- tuples are like lists, but immutable

```
MAGENTA = (255, 0, 255)
MAGENTA = 255, 0, 255 # Avoid this
```

- Demo 27: iterables/Demos/tuples.py
- The takeaway here is: Always use parentheses when creating tuples.

When to use tuples

- tuples are meant for holding heterogeneous collections ofdata. In tuples, the position of the element is meaningful.

- use cases for tuples include:

1. X-Y Coordinates (e.g, (55, -23))

2. Latitude-Longitude Coordinates (e.g., (43.0298, -76.0044))

3. Geometric Shapes (notice that these are tuples of tuples):

```
line = ((-40, 10), (-80, 170))
triangle = ((140, 200), (180, 270), (335, 180))
rectangle = ((40, 100), (80, 170), (235, 80), (195, 10))
```

Tuples

- empty tuple

```
t_empty = ()
```

- single element

```
t_single = ("a",)

>>> t1 = ("a",)
>>> type(t1)
<class 'tuple'>
>>> t2 = ("a")
>>> type(t2)
<class 'str'>
```

Ranges

- A range is an immutable sequence of numbers often used in for loops (which will be covered later)
- Ranges are created using range(), which can take one, two, or three arguments (note that stop number is not included):

```
range(stop)
range(start, stop)
range(start, stop, step)

range(10) # range starting at 0 and ending at 9
range(5, 11) # range starting at 5 and ending at 10
range(0, 13, 3) # range starting at 0, ending at 12, in steps of 3
range(4, -4, -1) # range starting at 4, ending at -3, in steps of -1
```

Converting sequences to lists

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 11))
[5, 6, 7, 8, 9, 10]
>>> list(range(0, 13, 3))
[0, 3, 6, 9, 12]
>>> list(range(-4, 4))
[-4, -3, -2, -1, 0, 1, 2, 3]
```

- You can convert any type of sequence to a list with the list() function:

```
>>> coords = (55, -23)
>>> list(coords)
[55, -23]
>>> list("Hello, world!")
['H', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!']
```

Indexing

- If we consider a sequence from left to right, the first element (the left-most) is at position 0. If we consider a sequence from right to left, the first element (the right-most) is at position -1.

```
>>> fruit = ['apple', 'orange', 'banana', 'pear', 'lemon', 'watermelon']
>>> fruit[0]
'apple'
>>> fruit[-1]
'watermelon'
>>> fruit[4]
'lemon'
>>> fruit[-3]
'pear'
```

Exercise 17

- Simple Rock, Paper, Scissors Game
  - 15-20 min

Slicing

- Slicing is the process of getting a slice or segment of a sequence as a new sequence.

```
sub_sequence = orig_sequence[first_pos:last_pos]
```

- If first_pos is left out, then it is assumed to be 0. F

```
>>> ["a", "b", "c", "d", "e"][:3]
['a', 'b', 'c']

>>> ["a", "b", "c", "d", "e"][3:]
['d', 'e']
```

Slicing

- below are examples of slicing a list, but any sequence type can be sliced

```
>>> fruit = ["apple", "orange", "banana", "pear", "lemon", "watermelon"]
>>> fruit[0:5]
['apple', 'orange', 'banana', 'pear', 'lemon']
>>> fruit[1:4]
['orange', 'banana', 'pear']
>>> fruit[4:]
['lemon', 'watermelon']
>>> fruit[-3:]
['pear', 'lemon', 'watermelon']
>>> fruit[:3]
['apple', 'orange', 'banana']
>>> fruit[-4:-1]
['banana', 'pear', 'lemon']
>>> fruit[:]
['apple', 'orange', 'banana', 'pear', 'lemon', 'watermelon']
```

Exercise 18

- Slicing Sequences
  - 10-20 min

min(), max()

- min(iter) and max(iter)

```
>>> colors = ["red", "blue", "green", "orange", "purple"]
>>> min(colors)
'blue'
>>> max(colors)
'red'
>>> ages = [27, 4, 15, 99, 33, 25]
>>> min(ages)
4
>>> max(ages)
99

>>> min("GuiDo")
'D'
```

sum()

- sum(iter[,start])

```
>>> nums = range(1, 6)
>>> sum(nums) # 1 + 2 + 3 + 4 + 5
15
>>> sum(nums, 10)
25
```

Converting sequences to string

- str.join(seq)

```
>>> colors = ["red", "blue", "green", "orange"]
>>> ','.join(colors)
'red,blue,green,orange'
>>> ', '.join(colors) # space after comma
'red, blue, green, orange'
>>> ':'.join(colors)
'red:blue:green:orange'
>>> ' '.join(colors)
'red blue green orange'
```

Splitting strings

- split()

```
>>> sentence = 'We are no longer the Knights Who Say "Ni!"'
>>> list_of_words = sentence.split()
>>> list_of_words
['We', 'are', 'no', 'longer', 'the', 'Knights', 'Who', 'Say', '"Ni!"']
```

```
>>> fruit = "apple, banana, pear, melon"
>>> fruit.split(",")
['apple', ' banana', ' pear', ' melon'
```

- get rid of the extra space

```
>>> fruit = "apple, banana, pear, melon"
>>> fruit.split(", ")
['apple', 'banana', 'pear', 'melon']
```

Splitting strings

- split() takes a second optional parameter, maxsplit, to indicate the maximum number of times to split the string. For example:

```
>>> fruit = "apple, banana, pear, melon"
>>> fruit.split(", ", 2)
['apple', 'banana', 'pear, melon']
```

- splitlines()

```
>>> fruit = """apple
     banana
     pear
     melon"""
>>> fruit.splitlines()
['apple', 'banana', 'pear', 'melon']
```

- Demo 28: iterables/data/states.txt
- Demo 29: iterables/Demos/states.py

Unpacking sequences

- assign multiple values at once

```
>>> first_name, last_name, company = "Guido", "Rossum", "Python"
>>> first_name
'Guido'
>>> last_name
'Rossum'
>>> company
'Python'
```

- Reverse (unpack) a sequence

```
>>> about_me = ("Guido", "Rossum", "Python")
>>> first_name, last_name, company = about_me
>>> first_name
'Guido'
>>> last_name
'Rossum'
>>> company
'Python'
```

Dictionaries

- Dictionaries are created with curly braces and comma-delimited key-value pairs

```
>>> dict = {
'key1': 'value 1',
'key2': 'value 2',
'key3': 'value 3'
}
>>> dict['key2'] = 'new value 2' # assign new value to existing key
>>> dict['key4'] = 'value 4' # assign value to new key
>>> print(dict['key1']) # print value of key
value 1
```

- Demo 30: iterables/Demos/dict.py

Dictionary methods

- assume this dictionary:

```
grades = {
"Math": 93,
"Art": 74,
"Music": 86
}
```

- mydict.get(key[, default])

```
>>> grades.get('English') 97
>>> grades.get('French') # returns None
>>> grades.get('French', 0)
0
```

Dictionary methods

- mydict.pop(key[, default])

```
>>> grades.pop('English')
97
>>> grades # Notice 'English' has been removed
{'Math': 93, 'Art': 74, 'Music': 86}
>>> grades.pop('English')
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
KeyError: 'English'
>>> grades.pop('English', 'Not found')
'Not found'
```

- mydict.popitem()

```
>>> grades.popitem()
('Music', 86)
>>> grades # Notice 'Music' has been removed
{'Math': 93, 'Art': 74}
```

Dictionary methods

- mydict.copy()

```
>>> grades_copy = grades.copy()
>>> grades['English'] = 94 # Add 'English' back to grades
>>> grades['Music'] = 100 # Add 'Music' back to grades
>>> grades # Notice they're back
{'Math': 93, 'Art': 74, 'English': 94, 'Music': 100}
>>> grades_copy # But grades_copy still has the old data
{'Math': 93, 'Art': 74}
```

- mydict.clear()

```
>>> grades.clear()
>>> grades # Dictionary is now empty
{}
```

Dictionary methods

- update()
- assume the following dictionary:

```
grades = {
"English": 97,
"Math": 93,
"Art": 74,
"Music": 86
}

grades.update({"Math": 97, "Gym": 93}) # argument is dict with keys
grades.update(Math=97, Gym=93) # individual arguments
grades.update([('Math', 97), ('Gym', 93)]) # argument is list of tuples

grades['Math'] = 97
```

Dictionary methods

- setdefault()
  - If key does not exist in the dictionary, key is added with a value of default.
  - If key exists in the dictionary, the value for key is left unchanged.
- Demo 31: iterables/Demos/setdefault.py
- When to use:
- Imagine you are creating a dictionary of grades from data in a file or a database. You cannot be sure what data that source contains, but you need grades for four specific subjects. You can populate your dictionary using the external data, and then use setdefault() to make sure you have data for all keys:

```
grades = get_data() # Imaginary function that returns dictionary
grades.setdefault('English') = 0
grades.setdefault('Math') = 0
grades.setdefault('Art') = 0
grades.setdefault('Music') = 0
```

- The grade for any one of the keys in the setdefault() calls will only be 0 if the dictionary returned by get_data() doesn’t include that key.

Dictionary view objects

- mydict.keys()
- mydict.values()
- mydict.items()

```
>>> grades = {
"English": 97,
"Math": 93,
"Art": 75
}

>>> grades.keys()
dict_keys(['English', 'Math', 'Art'])
>>> grades.values()
dict_values([97, 93, 75])
>>> grades.items()
dict_items([('English', 97), ('Math',
93), ('Art', 75)]
```

- Demo 32: iterables/Demos/dict_views.py
- To get a list from a dictionary view, use the list() method:

```
>>> grades = {
"English": 97,
"Math": 93,
"Art": 75
}

>>> list(grades.keys())
['English', 'Math', 'Art']
>>> list(grades.values())
[97, 93, 75]
>>> list(grades.items())
[('English', 97), ('Math', 93),
('Art', 75)]
```

Deleting dictionary keys

- use del
- Demo 33: iterables/Demos/del_dict.py

len() function

```
>>> len("hello")

5

>>> len( ["a", "b", "c"] )

3

>>> len( (255, 0, 255) )

3

>>> len({"Math": 97, "Music": 86, "Global Studies": 85})

3
```

Exercise 19

- Creating a Dictionary from User Input
  - 15-25 min

Sets

- Sets are mutable unordered collections of distinct immutable objects.

```
>>> classes = {"English", "Math", "Global Studies", "Art", "Music"}
>>> type(classes)
<class 'set'>
```

- one great use for sets is to remove duplicates from a list

```
>>> veggies = ["tomato", "spinach", "pepper", "pea", "tomato", "pea"]
>>> v_set = set(veggies) # converts to set and remove duplicates
>>> v_set
{'pepper', 'tomato', 'spinach', 'pea'}
>>> veggies = list(v_set) # converts back to list
>>> veggies
['pepper', 'tomato', 'spinach', 'pea']

#or

>>> veggies = ["tomato", "spinach", "pepper", "pea", "tomato", "pea"]
>>> veggies = list(set(veggies)) # remove duplicates
```

Sets

- or use a function:

```
def remove_dups(the_list):
    return list(set(the_list))

veggies = ["tomato", "spinach", "pepper", "pea", "tomato", "pea"]
veggies = remove_dups(veggies)
```

- veggies will now contain:

```
['pepper', 'tomato', 'spinach', 'pea'
```

*args and **kwargs

- When defining a function, you can include two special parameters to accept an arbitrary number of arguments:

- *args – A parameter that begins with a single asterisk will accept an arbitrary number of non-keyworded arguments and store them in a tuple. Often the variable is named *args, but you can call it whatever you want (e.g., *people or *colors). Note that any parameters that come after *args in the function signature are keyword-only parameters.

- **kwargs – A parameter that begins with two asterisks will accept an arbitrary number of keyworded arguments and store them in a dictionary. Often the variable is named **kwargs, but, as with *args, you can call it whatever you want (e.g., **people or **colors. When included, **kwargs must be the last parameter in the function signature

*args and **kwargs

- The parameters in a function definition must appear in the following order:

1. Non-keyword-only parameters that are required (i.e., have no defaults).
2. Non-keyword-only parameters that have defaults.
3. *args
4. Keyword-only parameters (with or without defaults).
5. **kwargs

Using *args

- use *args to allow the function to accept an arbitrary number of parameters
- Demo 34: iterables/Demos/add_nums.py

Using **kwargs

- function with **kwargs

```
def greet_me(**kwargs):
    print(kwargs)

greet_me(a=1,b=2,c=3)
```

- output:

```
{'a': 1, 'b': 2, 'c': 3}
```