1.8. Getting Started with Data

We stated above that Python supports the object-oriented programming paradigm. This means that Python considers data to be the focal point of the problem-solving process. In Python, as well as in any other object-oriented programming language, we define a **class** to be a description of what the data look like (the state) and what the data can do (the behavior). Classes are analogous to abstract data types because a user of a class only sees the state and behavior of a data item. Data items are called **objects** in the object-oriented paradigm. An object is an instance of a class.

1.8.1. Built-in Atomic Data Types¶

We will begin our review by considering the atomic data types. Python has two main built-in numeric classes that implement the integer and floating point data types. These Python classes are called `int` and `float` . The standard arithmetic operations, +, -, *, /, and ** (exponentiation), can be used with parentheses forcing the order of operations away from normal operator precedence. Other very useful operations are the remainder (modulo) operator, %, and integer division, //. Note that when two integers are divided, the result is a floating point. The integer division operator returns the integer portion of the quotient by truncating any fractional part.

Run | Load History | Show CodeLens

```
print(2+3*4)
print((2+3)*4)
print(2**10)
print(6/3)
print(7/3)
print(7//3)
print(7%3)
print(3/6)
print(3//6)
print(3%6)
print(2**100)

```

ActiveCode: 1 Basic Arithmetic Operators (intro_1)

The boolean data type, implemented as the Python `bool` class, will be quite useful for representing truth values. The possible state values for a boolean object are `True` and `False` with the standard boolean operators, `and` , `or` , and `not` .

rofBasicPython.html)

```
>>> True
True
>>> False
False
>>> False or True
True
>>> not (False or True)
False
>>> True and True
True
```

Boolean data objects are also used as results for comparison operators such as equality (==) and greater than (>). In addition, relational operators and logical operators can be combined together to form complex logical questions. Table 1 shows the relational and logical operators with examples shown in the session that follows.

Operation Name	Operator	Explanation
less than	$<$	Less than operator
greater than	$>$	Greater than operator
less than or equal	$<=$	Less than or equal to operator
greater than or equal	$>=$	Greater than or equal to operator
equal	$==$	Equality operator
not equal	$!=$	Not equal operator
logical and	and	Both operands True for result to be True
logical or	or	One or the other operand is True for the result to be True
logical not	not	Negates the truth value, False becomes True, True becomes False

Run | Load History | Show CodeLens

```
1 print(5==10)
2 print(10 > 5)
3 print((5 >= 1) and (5 <= 10))
4
```

rofBasicPython.html)

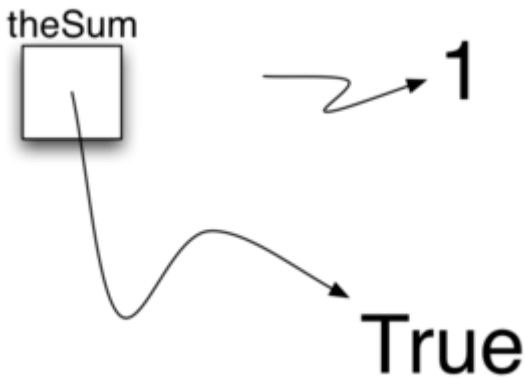
ActiveCode: 2 Basic Relational and Logical Operators (intro_2)

Identifiers are used in programming languages as names. In Python, identifiers start with a letter or an underscore (_), are case sensitive, and can be of any length. Remember that it is always a good idea to use names that convey meaning so that your program code is easier to read and understand.

A Python variable is created when a name is used for the first time on the left-hand side of an assignment statement. Assignment statements provide a way to associate a name with a value. The variable will hold a reference to a piece of data and not the data itself. Consider the following session:

```
>>> theSum = 0
>>> theSum
0
>>> theSum = theSum + 1
>>> theSum
1
>>> theSum = True
>>> theSum
True
```

The assignment statement `theSum = 0` creates a variable called `theSum` and lets it hold the reference to the data object `0` (see Figure 3). In general, the right-hand side of the assignment statement is evaluated and a reference to the resulting data object is “assigned” to the name on the left-hand side. At this point in our example, the type of the variable is integer as that is the type of the data currently being referred to by `theSum` . If the type of the data changes (see Figure 4), as shown above with the boolean value `True` , so does the type of the variable (`theSum` is now of the type boolean). The assignment statement changes the reference being held by the variable. This is a dynamic characteristic of Python. The same variable can refer to many different types of data.



rofBasicPython.html)

1.8.2. Built-in Collection Data Types

In addition to the numeric and boolean classes, Python has a number of very powerful built-in collection classes. Lists, strings, and tuples are ordered collections that are very similar in general structure but have specific differences that must be understood for them to be used properly. Sets and dictionaries are unordered collections.

A **list** is an ordered collection of zero or more references to Python data objects. Lists are written as comma-delimited values enclosed in square brackets. The empty list is simply `[ ]`. Lists are heterogeneous, meaning that the data objects need not all be from the same class and the collection can be assigned to a variable as below. The following fragment shows a variety of Python data objects in a list.

```
>>> [1,3,True,6.5]
[1, 3, True, 6.5]
>>> myList = [1,3,True,6.5]
>>> myList
[1, 3, True, 6.5]
```

Note that when Python evaluates a list, the list itself is returned. However, in order to remember the list for later processing, its reference needs to be assigned to a variable.

Since lists are considered to be sequentially ordered, they support a number of operations that can be applied to any Python sequence. Table 2 reviews these operations and the following session gives examples of their use.

Operation Name	Operator	Explanation
indexing	[ ]	Access an element of a sequence
concatenation	+	Combine sequences together
repetition	*	Concatenate a repeated number of times
membership	in	Ask whether an item is in a sequence
length	len	Ask the number of items in the sequence
slicing	[:]	Extract a part of a sequence

Note that the indices for lists (sequences) start counting with 0. The slice operation, myList[1:3], returns a list of items starting with the item indexed by 1 up to but not including the item indexed by 3.

Sometimes, you will want to initialize a list. This can quickly be accomplished by using repetition. For example,

```
>>> myList = [0] * 6
>>> myList
[0, 0, 0, 0, 0, 0]
```


One very important aside relating to the repetition operator is that the result is a repetition of references to the data objects in the sequence. This can best be seen by considering the following session:

Run | Load History | Show CodeLens

```
myList = [1,2,3,4]
A = [myList]*3
print(A)
myList[2]=45
print(A)

```

ActiveCode: 3 Repetition of References (intro_3)

The variable `A` holds a collection of three references to the original list called `myList`. Note that a change to one element of `myList` shows up in all three occurrences in `A`.

Lists support a number of methods that will be used to build data structures. Table 3 provides a summary. Examples of their use follow.

Method Name	Use	Explanation
append	alist.append(item)	Adds a new item to the end of a list
insert	alist.insert(i,item)	Inserts an item at the ith position in a list
pop	alist.pop()	Removes and returns the last item in a list
pop	alist.pop(i)	Removes and returns the ith item in a list
sort	alist.sort()	Modifies a list to be sorted
reverse	alist.reverse()	Modifies a list to be in reverse order
del	del alist[i]	Deletes the item in the ith position
index	alist.index(item)	Returns the index of the first occurrence of `item`
count	alist.count(item)	Returns the number of occurrences of `item`
remove	alist.remove(item)	Removes the first occurrence of `item`

Run | Load History | Show CodeLens

ofBasicPython.html)

```
myList = [1024, 3, True, 6.5]
myList.append(False)
print(myList)
myList.insert(2,4.5)

```

```
print(myList)
print(myList.pop())
print(myList)
print(myList.pop(1))
print(myList)
myList.pop(2)
print(myList)
myList.sort()
print(myList)
myList.reverse()
print(myList)
print(myList.count(6.5))
print(myList.index(4.5))
myList.remove(6.5)
print(myList)
del myList[0]
print(myList)

```

ActiveCode: 4 Examples of List Methods (intro_5)

You can see that some of the methods, such as `pop`, return a value and also modify the list. Others, such as `reverse`, simply modify the list with no return value. `pop` will default to the end of the list but can also remove and return a specific item. The index range starting from 0 is again used for these methods. You should also notice the familiar "dot" notation for asking an object to invoke a method. `myList.append(False)` can be read as "ask the object `myList` to perform its `append` method and send it the value `False`." Even simple data objects such as integers can invoke methods in this way.

```
>>> (54).__add__(21)
75
>>>
```

In this fragment we are asking the integer object `54` to execute its `add` method (called `__add__` in Python) and passing it `21` as the value to add. The result is the sum, `75`. Of course, we usually write this as `54+21`. We will say much more about these methods later in this section.

One common Python function that is often discussed in conjunction with lists is the `range` function. `range` produces a range object that represents a sequence of values. By using the `list` function, it is possible to see the value of the range object as a list. This is illustrated below.

rofBasicPython.html)

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

The range object represents a sequence of integers. By default, it will start with 0. If you provide more parameters, it will start and end at particular points and can even skip items. In our first example, `range(10)`, the sequence starts with 0 and goes up to but does not include 10. In our second example, `range(5,10)` starts at 5 and goes up to but not including 10. `range(5,10,2)` performs similarly but skips by twos (again, 10 is not included).

Strings are sequential collections of zero or more letters, numbers and other symbols. We call these letters, numbers and other symbols *characters*. Literal string values are differentiated from identifiers by using quotation marks (either single or double).

```
>>> "David"
'David'
>>> myName = "David"
>>> myName[3]
'i'
>>> myName*2
'DavidDavid'
>>> len(myName)
5
>>>
```

Since strings are sequences, all of the sequence operations described above work as you would expect. In addition, strings have a number of methods, some of which are shown in Table 4. For example,

```
>>> myName
'David'
>>> myName.upper()
'DAVID'
>>> myName.center(10)
'  David   '
>>> myName.find('v')
2
>>> myName.split('v')
['Da', 'id']
```

Of these, `split` will be very useful for processing data. `split` will take a string and return a list of strings using the split character as a division point. In the example, `v` is the division point. If no division is specified, the split method looks for whitespace characters such as tab, newline and space.

Method Name	Use	Explanation
`center`	`astring.center(w)`	Returns a string centered in a field of size `w`
`count`	`astring.count(item)`	Returns the number of occurrences of `item` in the string
`ljust`	`astring.ljust(w)`	Returns a string left-justified in a field of size `w`
`lower`	`astring.lower()`	Returns a string in all lowercase
`rjust`	`astring.rjust(w)`	Returns a string right-justified in a field of size `w`
`find`	`astring.find(item)`	Returns the index of the first occurrence of `item`
`split`	`astring.split(schar)`	Splits a string into substrings at `schar`

A major difference between lists and strings is that lists can be modified while strings cannot. This is referred to as **mutability**. Lists are mutable; strings are immutable. For example, you can change an item in a list by using indexing and assignment. With a string that change is not allowed.

```
>>> myList
[1, 3, True, 6.5]
>>> myList[0]=2**10
>>> myList
[1024, 3, True, 6.5]
>>>
>>> myName
'David'
>>> myName[0]='X'

Traceback (most recent call last):
  File "<pyshell#84>", line 1, in -toplevel-
    myName[0]='X'
TypeError: object doesn't support item assignment
>>>
```

Tuples are very similar to lists in that they are heterogeneous sequences of data. The difference is that a tuple is immutable, like a string. A tuple cannot be changed. Tuples are written as comma-delimited values enclosed in parentheses. As sequences, they can use any operation described above. For example,

rofBasicPython.html)

```
>>> myTuple = (2,True,4.96)
>>> myTuple
(2, True, 4.96)
>>> len(myTuple)
3
>>> myTuple[0]
2
>>> myTuple * 3
(2, True, 4.96, 2, True, 4.96, 2, True, 4.96)
>>> myTuple[0:2]
(2, True)
>>>
```

However, if you try to change an item in a tuple, you will get an error. Note that the error message provides location and reason for the problem.

```
>>> myTuple[1]=False

Traceback (most recent call last):
  File "<pyshell#137>", line 1, in -toplevel-
    myTuple[1]=False
TypeError: object doesn't support item assignment
>>>
```

A set is an unordered collection of zero or more immutable Python data objects. Sets do not allow duplicates and are written as comma-delimited values enclosed in curly braces. The empty set is represented by `set()`. Sets are heterogeneous, and the collection can be assigned to a variable as below.

```
>>> {3,6,"cat",4.5,False}
{False, 4.5, 3, 6, 'cat'}
>>> mySet = {3,6,"cat",4.5,False}
>>> mySet
{False, 4.5, 3, 6, 'cat'}
>>>
```

Even though sets are not considered to be sequential, they do support a few of the familiar operations presented earlier. Table 5 reviews these operations and the following session gives examples of their use.

Operation Name	Operator	Explanation
membership	in	Set membership
length	len	Returns the cardinality of the set
|	`aset | otherset`	Returns a new set with all elements from both sets
&	`aset & otherset`	Returns a new set with only those elements common to both sets

Operation Name	Operator	Explanation
-	`aset - otherset`	Returns a new set with all items from the first set not in second
<=	`aset <= otherset`	Asks whether all elements of the first set are in the second

```
>>> mySet
{False, 4.5, 3, 6, 'cat'}
>>> len(mySet)
5
>>> False in mySet
True
>>> "dog" in mySet
False
>>>
```

Sets support a number of methods that should be familiar to those who have worked with them in a mathematics setting. Table 6 provides a summary. Examples of their use follow. Note that `union`, `intersection`, `issubset`, and `difference` all have operators that can be used as well.

Method Name	Use	Explanation
union	`aset.union(otherset)`	Returns a new set with all elements from both sets
intersection	`aset.intersection(otherset)`	Returns a new set with only those elements common to both sets
difference	`aset.difference(otherset)`	Returns a new set with all items from first set not in second
issubset	`aset.issubset(otherset)`	Asks whether all elements of one set are in the other
add	`aset.add(item)`	Adds item to the set
remove	`aset.remove(item)`	Removes item from the set
pop	`aset.pop()`	Removes an arbitrary element from the set
clear	`aset.clear()`	Removes all elements from the set

rofBasicPython.html)

```
>>> mySet
{False, 4.5, 3, 6, 'cat'}
>>> yourSet = {99,3,100}
>>> mySet.union(yourSet)
{False, 4.5, 3, 100, 6, 'cat', 99}
>>> mySet | yourSet
{False, 4.5, 3, 100, 6, 'cat', 99}
>>> mySet.intersection(yourSet)
{3}
>>> mySet & yourSet
{3}
>>> mySet.difference(yourSet)
{False, 4.5, 6, 'cat'}
>>> mySet - yourSet
{False, 4.5, 6, 'cat'}
>>> {3,100}.issubset(yourSet)
True
>>> {3,100}<=yourSet
True
>>> mySet.add("house")
>>> mySet
{False, 4.5, 3, 6, 'house', 'cat'}
>>> mySet.remove(4.5)
>>> mySet
{False, 3, 6, 'house', 'cat'}
>>> mySet.pop()
False
>>> mySet
{3, 6, 'house', 'cat'}
>>> mySet.clear()
>>> mySet
set()
>>>
```

Our final Python collection is an unordered structure called a **dictionary**. Dictionaries are collections of associated pairs of items where each pair consists of a key and a value. This key-value pair is typically written as key:value. Dictionaries are written as comma-delimited key:value pairs enclosed in curly braces. For example,

```
>>> capitals = {'Iowa':'DesMoines','Wisconsin':'Madison'}
>>> capitals
{'Wisconsin': 'Madison', 'Iowa': 'DesMoines'}
>>>
```

We can manipulate a dictionary by accessing a value via its key or by adding another key-value pair. The syntax for access looks much like a sequence access except that instead of using the index of the item we use the key value. To add a new value is similar.

rofBasicPython.html)

Run | Load History | Show CodeLens

```
capitals = {'Iowa':'DesMoines','Wisconsin':'Madison'}
print(capitals['Iowa'])
capitals['Utah']='SaltLakeCity'
print(capitals)
capitals['California']='Sacramento'
print(len(capitals))
for k in capitals:
   print(capitals[k]," is the capital of ", k)

```

ActiveCode: 5 Using a Dictionary (intro_7)

It is important to note that the dictionary is maintained in no particular order with respect to the keys. The first pair added (`'Utah': 'SaltLakeCity'`) was placed first in the dictionary and the second pair added (`'California': 'Sacramento'`) was placed last. The placement of a key is dependent on the idea of "hashing," which will be explained in more detail in Chapter 4. We also show the length function performing the same role as with previous collections.

Dictionaries have both methods and operators. Table 7 and Table 8 describe them, and the session shows them in action. The `keys` , `values` , and `items` methods all return objects that contain the values of interest. You can use the `list` function to convert them to lists. You will also see that there are two variations on the `get` method. If the key is not present in the dictionary, `get` will return `None` . However, a second, optional parameter can specify a return value instead.

Operator	Use	Explanation
`[]`	`myDict[k]`	Returns the value associated with `k` , otherwise its an error
`in`	`key in adict`	Returns `True` if key is in the dictionary, `False` otherwise
`del`	del `adict[key]`	Removes the entry from the dictionary

rofBasicPython.html)

```
>>> phoneext={'david':1410,'brad':1137}
>>> phoneext
{'brad': 1137, 'david': 1410}
>>> phoneext.keys()
dict_keys(['brad', 'david'])
>>> list(phoneext.keys())
['brad', 'david']
>>> phoneext.values()
dict_values([1137, 1410])
>>> list(phoneext.values())
[1137, 1410]
>>> phoneext.items()
dict_items([('brad', 1137), ('david', 1410)])
>>> list(phoneext.items())
[('brad', 1137), ('david', 1410)]
>>> phoneext.get("kent")
>>> phoneext.get("kent","NO ENTRY")
'NO ENTRY'
>>>
```

Method Name	Use	Explanation
`keys`	`adict.keys()`	Returns the keys of the dictionary in a dict_keys object
`values`	`adict.values()`	Returns the values of the dictionary in a dict_values object
`items`	`adict.items()`	Returns the key-value pairs in a dict_items object
`get`	`adict.get(k)`	Returns the value associated with `k`, `None` otherwise
`get`	`adict.get(k,alt)`	Returns the value associated with `k`, `alt` otherwise

Note

This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

Run | Load History | Show CodeLens

```
1
2
3
```

rofBasicPython.html)

ActiveCode: 6 (scratch_01_01)