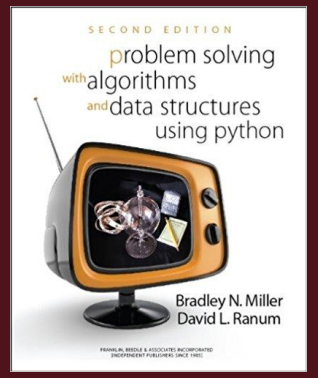
Problem Solving with Algorithms and Data Structure Using Python



Unit 1: Introduction

LECTURE 2: REVIEW OF PYTHON LANGUAGE

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- In this section, we will review the programming language Python and also provide some more detailed examples of the ideas from the previous section.
- If you are new to Python or find that you need more information about any of the topics presented, we recommend that you consult a resource such as the Python Language Reference or a Python Tutorial.
- Our goal here is to reacquaint you with the language and also reinforce some of the concepts that will be central to later chapters.

Review of Basic Python

- Python is a modern, easy-to-learn, object-oriented programming language. It has a powerful set of built-in data types and easy-to-use control constructs. Since Python is an interpreted language, it is most easily reviewed by simply looking at and describing interactive sessions. You should recall that the interpreter displays the familiar >>> prompt and then evaluates the Python construct that you provide. For example,

```
>>> print("Algorithms and Data Structures")
Algorithms and Data Structures
>>>
```

shows the prompt, the print function, the result, and the next prompt.

Data with Python

LECTURE 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.

Try These

```
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)
```

Boolean

- 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.

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

Boolean

- 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.

Table 1: Relational and Logical Operators

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	andand	Both operands True for result to be True
logical or	oror	One or the other operand is True for the result to be True
logical not	notnot	Negates the truth value, False becomes True, True becomes False

Identifiers

- 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:

Identifiers

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

Identifiers

- 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.

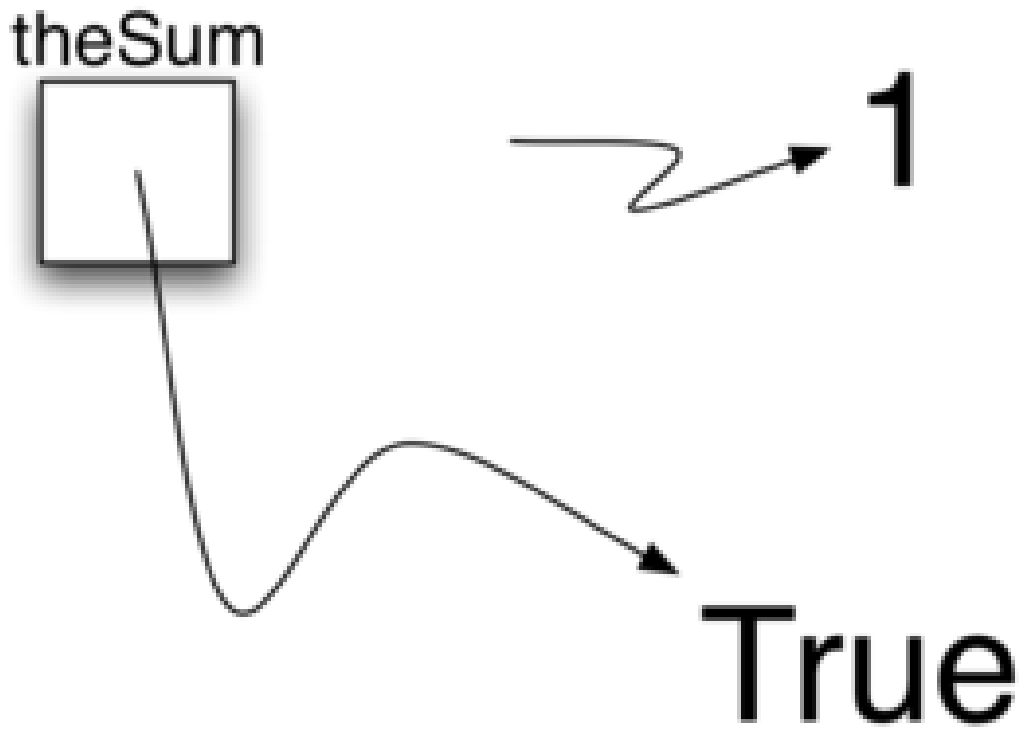


Figure 3: Variables Hold References to Data Objects

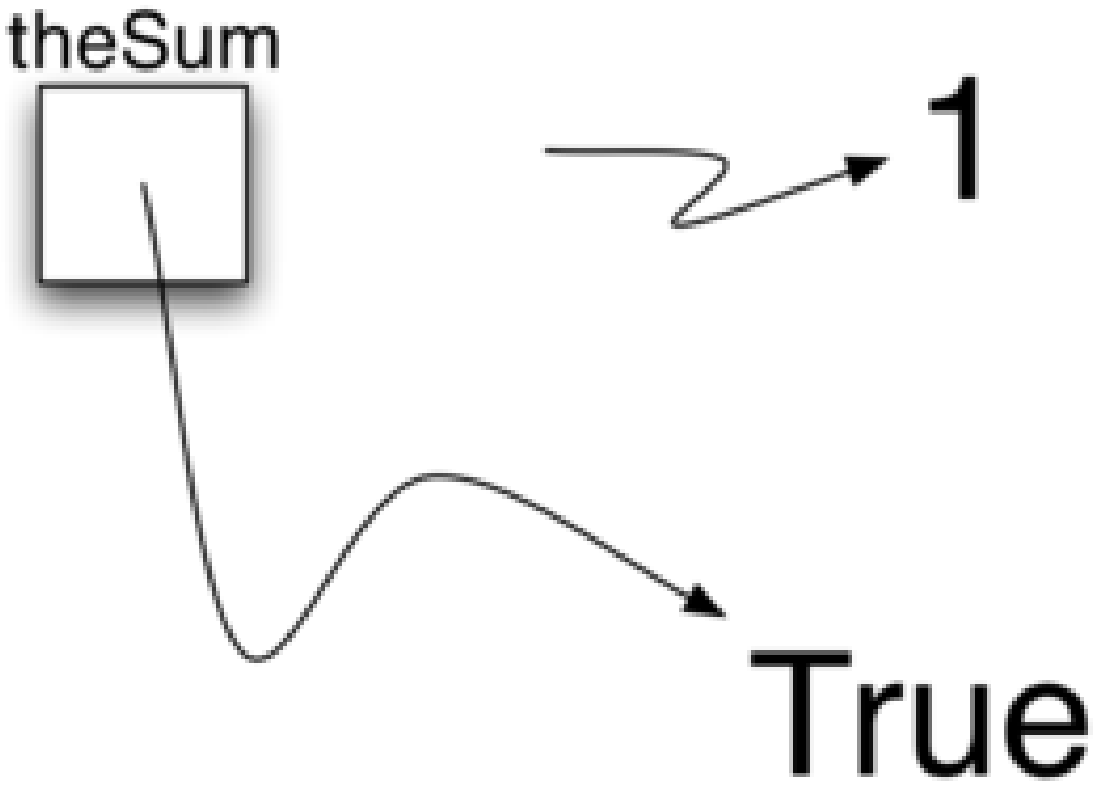


Figure 4: Assignment Changes the Reference

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.

Lists

LECTURE 1

Lists

- 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.

Lists

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

Lists

- 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.

Table 2: Operations on Any Sequence in Python

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

Lists

- 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]
```

Try These

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

Lists

- 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.

Table 3: Methods Provided by Lists in Python

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

Try These

```
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)
```

Try These

```
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)
```

Lists

- 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.

Lists

- 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.

Lists

```
>>> (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.

Lists

- 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.

```
>>> 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]
>>>
```

Lists

- 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

LECTURE 1

Strings

- **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).

Strings

- **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).

Strings

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

Strings

- 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.

Strings

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

Strings

- 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.

Table 4: Methods Provided by Strings in Python

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

Strings

- 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.

Strings

```
>>> 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

LECTURE 1

Tuples

- 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.

Tuples

```
>>> 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)
>>>
```

Tuples

- 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.

Tuples

```
>>> 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
>>>
```

Sets

LECTURE 1

Sets

- 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.

Sets

```
>>> {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: Operations on a Set in Python

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
-	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

Sets

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

Sets

- 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.

Table 6: Methods Provided by Sets in Python

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

Sets

```
>>> 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}<=yourSet
True
```

Sets

```
>>> {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()
>>>
```

Dictionary

LECTURE 1

Dictionary

- 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.

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

Dictionary

- 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. .

Try These

```
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)
```

Dictionary

- 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.

Dictionary

- 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.

Table 7: Operators Provided by Dictionaries in Python

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

```
>>> 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’
>>>
```

Table 8: Methods Provided by Dictionaries in Python

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

Input and Output

LECTURE 1

Input and Output

- We often have a need to interact with users, either to get data or to provide some sort of result. Most programs today use a dialog box as a way of asking the user to provide some type of input. While Python does have a way to create dialog boxes, there is a much simpler function that we can use.
- Python provides us with a function that allows us to ask a user to enter some data and returns a reference to the data in the form of a string. The function is called input.

Input and Output

- Python's input function takes a single parameter that is a string. This string is often called the prompt because it contains some helpful text prompting the user to enter something. For example, you might call input as follows:

```
aName = input('Please enter your name: ‘)
```

Input and Output

- Now whatever the user types after the prompt will be stored in the aName variable. Using the input function, we can easily write instructions that will prompt the user to enter data and then incorporate that data into further processing. For example, in the following two statements, the first asks the user for their name and the second prints the result of some simple processing based on the string that is provided.

```
aName = input("Please enter your name ")
print("Your name in all capitals is",aName.upper(),
      "and has length", len(aName))
```

Input and Output

- It is important to note that the value returned from the input function will be a string representing the exact characters that were entered after the prompt. If you want this string interpreted as another type, you must provide the type conversion explicitly. In the statements below, the string that is entered by the user is converted to a float so that it can be used in further arithmetic processing.

```
sradius = input("Please enter the radius of the circle ")
radius = float(sradius)
diameter = 2 * radius
```

String Formatting

- We have already seen that the print function provides a very simple way to output values from a Python program. print takes zero or more parameters and displays them using a single blank as the default separator. It is possible to change the separator character by setting the sep argument.
- In addition, each print ends with a newline character by default. This behavior can be changed by setting the end argument. These variations are shown in the following session:

String Formatting

```
>>> print("Hello")
Hello
>>> print("Hello","World")
Hello World
>>> print("Hello","World", sep="***")
Hello***World
>>> print("Hello","World", end="***")
Hello World***
>>>
```

String Formatting

- It is often useful to have more control over the look of your output. Fortunately, Python provides us with an alternative called formatted strings. A formatted string is a template in which words or spaces that will remain constant are combined with placeholders for variables that will be inserted into the string. For example, the statement

```
print(aName, "is", age, "years old.")
```

String Formatting

- contains the words is and years old, but the name and the age will change depending on the variable values at the time of execution. Using a formatted string, we write the previous statement as

```
print("%s is %d years old." % (aName, age))
```

String Formatting

- This simple example illustrates a new string expression. The % operator is a string operator called the format operator. The left side of the expression holds the template or format string, and the right side holds a collection of values that will be substituted into the format string. Note that the number of values in the collection on the right side corresponds with the number of % characters in the format string. Values are taken—in order, left to right—from the collection and inserted into the format string.

String Formatting

- Let’s look at both sides of this formatting expression in more detail. The format string may contain one or more conversion specifications. A conversion character tells the format operator what type of value is going to be inserted into that position in the string. In the example above, the %s specifies a string, while the %d specifies an integer. Other possible type specifications include i, u, f, e, g, c, or %. Table 9 summarizes all of the various type specifications.

Table 9: String Formatting Conversion Characters

Character	Output Format
d, i	Integer
u	Unsigned integer
f	Floating point as m.ddddd
e	Floating point as m.dddddde+/-xx
E	Floating point as m.ddddd E+/-xx
g	Use %e for exponents less than −4−4 or greater than +5+5, otherwise use %f
c	Single character
s	String, or any Python data object that can be converted to a string by using the str function.
%	Insert a literal % character

String Formatting

- In addition to the format character, you can also include a format modifier between the % and the format character. Format modifiers may be used to left-justify or right-justifiy the value with a specified field width. Modifiers can also be used to specify the field width along with a number of digits after the decimal point. Table 10 explains these format modifiers

Table 10: Additional formatting options

Modifier	Example	Description
number	%20d	Put the value in a field width of 20
-	%-20d	Put the value in a field 20 characters wide, left-justified
+	%+20d	Put the value in a field 20 characters wide, right-justified
0	%020d	Put the value in a field 20 characters wide, fill in with leading zeros.
.	%20.2f	Put the value in a field 20 characters wide with 2 characters to the right of the decimal point.
(name)	%(name)d	Get the value from the supplied dictionary using name as the key.

String Formatting

- The right side of the format operator is a collection of values that will be inserted into the format string. The collection will be either a tuple or a dictionary. If the collection is a tuple, the values are inserted in order of position. That is, the first element in the tuple corresponds to the first format character in the format string. If the collection is a dictionary, the values are inserted according to their keys. In this case all format characters must use the (name) modifier to specify the name of the key.

String Formatting

```
>>> price = 24 >>> item = "banana"
>>> print("The %s costs %d cents"% (item,price))
The banana costs 24 cents
>>> print("The %+10s costs %5.2f cents"% (item,price))
The     banana costs 24.00 cents
>>> print("The %+10s costs %10.2f cents"% (item,price))
The     banana costs       24.00 cents
>>> itemdict = {"item":"banana","cost":24}
>>> print("The %(item)s costs %(cost)7.1f cents"% itemdict)
The banana costs     24.0 cents
>>>
```

String Formatting

- In addition to format strings that use format characters and format modifiers, Python strings also include a format method that can be used in conjunction with a new Formatter class to implement complex string formatting. More about these features can be found in the Python library reference manual.

Control Structure

LECTURE 1

Control Structures

- As we noted earlier, algorithms require two important control structures: iteration and selection. Both of these are supported by Python in various forms. The programmer can choose the statement that is most useful for the given circumstance.
- For iteration, Python provides a standard while statement and a very powerful for statement. The while statement repeats a body of code as long as a condition is true.

Control Structures

```
>>> counter = 1
>>> while counter <= 5:
...         print("Hello, world")
...         counter = counter + 1

Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
```

Control Structures

- prints out the phrase “Hello, world” five times. The condition on the while statement is evaluated at the start of each repetition. If the condition is True, the body of the statement will execute. It is easy to see the structure of a Python while statement due to the mandatory indentation pattern that the language enforces.

Control Structures

- The while statement is a very general purpose iterative structure that we will use in a number of different algorithms. In many cases, a compound condition will control the iteration. A fragment such as

```
while counter <= 10 and not done:
    ...
```

- would cause the body of the statement to be executed only in the case where both parts of the condition are satisfied. The value of the variable counter would need to be less than or equal to 10 and the value of the variable done would need to be False (not False is True) so that True and True results in True.

Control Structures

- Even though this type of construct is very useful in a wide variety of situations, another iterative structure, the for statement, can be used in conjunction with many of the Python collections. The for statement can be used to iterate over the members of a collection, so long as the collection is a sequence. So, for example,

Control Structures

```
>>> for item in [1,3,6,2,5]:
...         print(item)
...
1
3
6
2
5
```

Control Structures

- assigns the variable item to be each successive value in the list [1,3,6,2,5]. The body of the iteration is then executed. This works for any collection that is a sequence (lists, tuples, and strings).
- A common use of the for statement is to implement definite iteration over a range of values. The statement

Control Structures

```
>>> for item in (5):
...         print(item**2)
...
0
1
4
9
16
>>>
```

Control Structures

- will perform the print function five times. The range function will return a range object representing the sequence 0,1,2,3,4 and each value will be assigned to the variable item. This value is then squared and printed.
- The other very useful version of this iteration structure is used to process each character of a string. The following code fragment iterates over a list of strings and for each string processes each character by appending it to a list. The result is a list of all the letters in all of the words.

Try These

```
wordlist = ['cat','dog','rabbit']
letterlist = [ ]
for aword in wordlist:
    for aletter in aword:
        letterlist.append(aletter)
print(letterlist)
```

Control Structures

- Selection statements allow programmers to ask questions and then, based on the result, perform different actions. Most programming languages provide two versions of this useful construct: the ifelse and the if. A simple example of a binary selection uses the ifelse statement.

Control Structures

```
if n<0:
    print("Sorry, value is negative")
else:
    print(math.sqrt(n))
```

Control Structures

- In this example, the object referred to by n is checked to see if it is less than zero. If it is, a message is printed stating that it is negative. If it is not, the statement performs the else clause and computes the square root.
- Selection constructs, as with any control construct, can be nested so that the result of one question helps decide whether to ask the next. For example, assume that score is a variable holding a reference to a score for a computer science test.

Control Structures

```
if score >= 90:
    print('A')
else: if score >=80:
    print('B')
        else: if score >= 70:
                print('C')
            else: if score >= 60:
                    print('D')
                else:
                    print('F')
```

Control Structures

- This fragment will classify a value called score by printing the letter grade earned. If the score is greater than or equal to 90, the statement will print A. If it is not (else), the next question is asked. If the score is greater than or equal to 80 then it must be between 80 and 89 since the answer to the first question was false. In this case print B is printed. You can see that the Python indentation pattern helps to make sense of the association between if and else without requiring any additional syntactic elements.

Control Structures

- An alternative syntax for this type of nested selection uses the elif keyword. The else and the next if are combined so as to eliminate the need for additional nesting levels. Note that the final else is still necessary to provide the default case if all other conditions fail.

Control Structures

```
if score >= 90:
    print('A’)
elif score >=80:
    print('B’)
elif score >= 70:
    print('C’)
elif score >= 60:
    print('D’)
else:
    print('F’)
```

Control Structures

- Python also has a single way selection construct, the if statement. With this statement, if the condition is true, an action is performed. In the case where the condition is false, processing simply continues on to the next statement after the if. For example, the following fragment will first check to see if the value of a variable n is negative. If it is, then it is modified by the absolute value function. Regardless, the next action is to compute the square root.

Control Structures

```
if n<0:
    n = abs(n)
print(math.sqrt(n))
```

Control Structures

- Returning to lists, there is an alternative method for creating a list that uses iteration and selection constructs known as a list comprehension. A list comprehension allows you to easily create a list based on some processing or selection criteria. For example, if we would like to create a list of the first 10 perfect squares, we could use a for statement:

Control Structures

```
>>> sqlist=[]
>>> for x in range(1,11):
        sqlist.append(x*x)
>>> sqlist
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>>
```

Control Structures

- Using a list comprehension, we can do this in one step as

```
>>> sqlist=[x*x for x in range(1,11)]
>>> sqlist
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>>
```

Control Structures

- The variable x takes on the values 1 through 10 as specified by the for construct. The value of x*x is then computed and added to the list that is being constructed. The general syntax for a list comprehension also allows a selection criteria to be added so that only certain items get added.

```
>>> sqlist=[x*x for x in range(1,11) if x%2 != 0]
>>> sqlist [1, 9, 25, 49, 81]
>>>
```

Control Structures

This list comprehension constructed a list that only contained the squares of the odd numbers in the range from 1 to 10. Any sequence that supports iteration can be used within a list comprehension to construct a new list.

```
>>>[ch.upper() for ch in 'comprehension' if ch not in 'aeiou’]
['C', 'M', 'P', 'R', 'H', 'N', 'S', 'N’]
>>>
```

Exception Handling

LECTURE 1

Exception Handling

- There are two types of errors that typically occur when writing programs. The first, known as a syntax error, simply means that the programmer has made a mistake in the structure of a statement or expression. For example, it is incorrect to write a for statement and forget the colon.

```
>>> for i in range(10)
SyntaxError: invalid syntax (<pyshell#61>, line 1)
```

- In this case, the Python interpreter has found that it cannot complete the processing of this instruction since it does not conform to the rules of the language. Syntax errors are usually more frequent when you are first learning a language.

Exception Handling

- The other type of error, known as a logic error, denotes a situation where the program executes but gives the wrong result. This can be due to an error in the underlying algorithm or an error in your translation of that algorithm. In some cases, logic errors lead to very bad situations such as trying to divide by zero or trying to access an item in a list where the index of the item is outside the bounds of the list. In this case, the logic error leads to a runtime error that causes the program to terminate. These types of runtime errors are typically called **exceptions**.

Exception Handling

- Most of the time, beginning programmers simply think of exceptions as fatal runtime errors that cause the end of execution. However, most programming languages provide a way to deal with these errors that will allow the programmer to have some type of intervention if they so choose. In addition, programmers can create their own exceptions if they detect a situation in the program execution that warrants it.

Exception Handling

- When an exception occurs, we say that it has been “raised.” You can “handle” the exception that has been raised by using a try statement. For example, consider the following session that asks the user for an integer and then calls the square root function from the math library. If the user enters a value that is greater than or equal to 0, the print will show the square root. However, if the user enters a negative value, the square root function will report a ValueError exception.

```
>>> anumber = int(input("Please enter an integer "))
Please enter an integer -23
>>> print(math.sqrt(anumber))
Traceback (most recent call last):
    File “<pyshell#102>”, line 1, in <module>
      print(math.sqrt(anumber))
ValueError: math domain error
>>>
```

Exception Handling

- We can handle this exception by calling the print function from within a try block. A corresponding except block “catches” the exception and prints a message back to the user in the event that an exception occurs. For example:

Exception Handling

```
>>> try:
        print(math.sqrt(anumber))
    except:
        print("Bad Value for square root")
        print("Using absolute value instead")
        print(math.sqrt(abs(anumber)))
Bad Value for square root
Using absolute value instead
4.79583152331
>>>
```

Exception Handling

- will catch the fact that an exception is raised by sqrt and will instead print the messages back to the user and use the absolute value to be sure that we are taking the square root of a non-negative number. This means that the program will not terminate but instead will continue on to the next statements.
- It is also possible for a programmer to cause a runtime exception by using the raise statement. For example, instead of calling the square root function with a negative number, we could have checked the value first and then raised our own exception. The code fragment below shows the result of creating a new RuntimeError exception. Note that the program would still terminate but now the exception that caused the termination is something explicitly created by the programmer.

Exception Handling

```
>>> if anumber < 0:
...         raise RuntimeError("You can't use a negative number")
...     else:
...         print(math.sqrt(anumber)) ...
Traceback (most recent call last):
      File “<stdin>”, line 2, in <module>
RuntimeError: You can't use a negative number
>>>
```

- There are many kinds of exceptions that can be raised in addition to the RuntimeError shown above. See the Python reference manual for a list of all the available exception types and for how to create your own.

Functions

LECTURE 1

Defining Functions

- The earlier example of procedural abstraction called upon a Python function called sqrt from the math module to compute the square root. In general, we can hide the details of any computation by defining a function. A function definition requires a name, a group of parameters, and a body. It may also explicitly return a value. For example, the simple function defined below returns the square of the value you pass into it.

Defining Functions

```
>>> def square(n):
...         return n**2
...
>>> square(3)
9
>>> square(square(3))
81
>>>
```

Defining Functions

- The syntax for this function definition includes the name, square, and a parenthesized list of formal parameters. For this function, n is the only formal parameter, which suggests that square needs only one piece of data to do its work. The details, hidden "inside the box," simply compute the result of n**2 and return it. We can invoke or call the square function by asking the Python environment to evaluate it, passing an actual parameter value, in this case, 3. Note that the call to square returns an integer that can in turn be passed to another invocation.

Defining Functions

- We could implement our own square root function by using a well-known technique called “Newton’s Method.” Newton’s Method for approximating square roots performs an iterative computation that converges on the correct value.
- The equation $newguess = \frac{1}{2}\left(oldguess + \frac{n}{oldguess}\right)$ takes a value n and repeatedly guesses the square root by making each `newguess` the `oldguess` in the subsequent iteration. The initial guess used here is n/2.

Defining Functions

- Listing 1 shows a function definition that accepts a value n and returns the square root of n after making 20 guesses. Again, the details of Newton's Method are hidden inside the function definition and the user does not have to know anything about the implementation to use the function for its intended purpose.
- Listing 1 also shows the use of the # character as a comment marker. Any characters that follow the # on a line are ignored.

Listing 1:

```
def squareroot(n):
    root = n/2 #initial guess will be 1/2 of n
    for k in range(20):
        root = (1/2)*(root + (n / root))
    return root


>>>squareroot(9)
3.0
>>>squareroot(4563)
67.54998149518621 6
>>>
```