**PYTHON**

**Week 1: Review of Essentials in CS Content**

**input** -- get data from the keyboard, a file, or some other device.

**output** -- display data on the screen or send data to a file or other device.

**math** -- perform basic mathematical operations such as addition and multiplication.

**conditional execution** -- check for certain conditions and execute the appropriate code.

**repetition** -- perform some action repeatedly, usually with some variation

If you have any incorrect answers, go to the blue navigation bar and click on **Content**. To the left, click on the module called **Review General Content**. Then on the right, locate the file called **Computer Programming** to review the information.

Arrange in order the steps for creating a computer program.

The correct answer is **requirements, design, coding, and testing.**

**syntax error** -- This error refers to the structure of a program and the rules about that structure not being correct.

**runtime error** -- This error does not appear until after the program has started running.

**semantic error** -- This type of error will run successfully in the sense that the computer will not generate any error messages, but it will not produce the desired results.

In Python, standard **output** looks like this:

print("Hello, World!")

This is an example of a print statement, which doesn’t actually print anything on paper. It displays a value on the screen. In this case, the result is the words "Hello World!" The quotation marks in the program indicate the beginning and end of the text to be displayed; they don’t appear in the result.

If you would like to include the value of a variable in your output, you can use the , symbol. In the example below the value for age is inserted. So the output is "My age is 21". If the variables contains a string (ie letters), you can use the + symbol.

age = 21;  
print("My age is ", age);

The following example creates a variable called sMessage to store a String -> Hello World! to be outputted to the console.

sMessage = "Hello World!"  
print ("Message: " + sMessage)

The next example shows how to output data on separate lines of consoles:

print("Orange")  
print("Blue")  
print("Green")  
print("Purple")

In Python, the **input** command is called input(). This built-in function allows you to enter some text to create a prompt:

>>> sample = input("Enter sample text: ")

In Python all input from the user is a series of characters. If you would like certain numerals to be interpreted as numerical values rather than as characters, put the eval() command around your input() command.

This example creates a variable called input to store user input from the command prompt.

input = input()

print(input)

This example shows how to capture input of different datatypes typed by the user.

myname = input("Name: ")

myage = eval(input("Age: "))

mysalary = eval(input("Salary: "))

print ("My name is " + myname)

print ("I am " + myage)

print ("I earn " + mysalary)

**Arithmetic**

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called *operands*.

|  |  |  |
| --- | --- | --- |
| **Arithmetic operators** | | **Examples** |
| **Addition** | + | 20+32 |
| **Subtraction** | - | hour-1 |
| **Multiplication** | \* | hour\*60+minute |
| **Division** | / | minute/60 |
| **Exponentiation** | \*\* | 5\*\*2 (5+9)\*(15-7) |

In Python, the result of the division above is a float. The new operator // performs floor division.

7//2 = 3

If either of the operands is a floating-point number, Python performs floating-point division, and the result is a float:

minute/60.0 = 0.98333333333333328

One of the most common forms of multiple assignment is an update, where the new value of the variable depends on the old.

x = x + 1

This means “get the current value of x, add 1, and then update x with the new value.” If you try to update a variable that doesn’t exist, you get an error, because Python evaluates the right side before it assigns a value to x:

x = x + 1

This line will give you a*NameError: name 'x' is not defined*

Before you can update a variable, you have to initialize it, usually with a simple assignment:

x = 0  
x = x + 1

Updating a variable by adding 1 is called *incrementing*; subtracting 1 is called *decrementing*.

**Arrays**

Python also has arrays, but they are called lists. It is generally the same concept, just with a different name.

Like a string, a listis a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called elements or sometimes items. There are several ways to create a new list; the simplest is to enclose the elements in square brackets, [ and ], as shown in the following examples:

[10, 20, 30, 40]  
['crunchy frog', 'ram bladder', 'lark vomit']

The first example is a list of four integers. The second is a list of three strings. The elements of a list don’t have to be the same type. The following list contains a string, a float, an integer, and another list:

['spam', 2.0, 5, [10, 20]]

A list within another list is nested. A list that contains no elements is called an empty list; you can create one with empty brackets, []. As you might expect, you can assign list values to variables:

>>> cheeses = ['Cheddar', 'Edam', 'Gouda']  
>>> numbers = [17, 123]  
>>> empty = []  
>>> print (cheeses, numbers, empty)  
      ['Cheddar', 'Edam', 'Gouda'] [17, 123] []

The most common way to traverse the elements of a list is with a “for” loop. The syntax is the same as for strings:

for cheese in cheeses:  
       print (cheese)

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions range and len:

for i in range(len(numbers)):  
       numbers[i] = numbers[i] \* 2

This loop traverses the list and updates each element. len (short for "length") returns the number of elements in the list. range returns a list of indices from 0 to n - 1, where n is the length of the list. Each time through the loop, i gets the index of the next element. The assignment statement in the body uses i to read the old value of the element and to assign the new value.

Python provides methods that operate on lists. For example, append adds a new element to the end of a list:

>>> t = ['a', 'b', 'c']  
>>> t.append('d')  
>>> print (t)  
['a', 'b', 'c', 'd']

extend takes a list as an argument and appends all of the elements:

>>> t1 = ['a', 'b', 'c']  
>>> t2 = ['d', 'e']  
>>> t1.extend(t2)  
>>> print (t1)  
['a', 'b', 'c', 'd', 'e']

The following example combines the two lists and leaves t2 unmodified. sort arranges the elements of the list from low to high:

>>> t = ['d', 'c', 'e', 'b', 'a']  
>>> t.sort()  
>>> print (t)  
['a', 'b', 'c', 'd', 'e']

List methods are void as statements; they modify the list and return None. If you accidentally write t = t.sort(), you will be disappointed with the result.

There are several ways to delete elements from a list. If you know the index of the element you want, you can use pop:

>>> t = ['a', 'b', 'c']  
>>> x = t.pop(1)  
>>> print (t)  
['a', 'c']  
>>> print (x)  
b

pop modifies the list and returns the element that was removed. If you don’t provide an index, it deletes and returns the last element. If you don’t need the removed value, you can use the del operator:

>>> t = ['a', 'b', 'c']  
>>> del t[1]  
>>> print (t)  
['a', 'c']

If you know the element you want to remove (but not the index), you can use remove:

>>> t = ['a', 'b', 'c']  
>>> t.remove('b')  
>>> print (t)  
['a', 'c']

The return value from remove is None. To remove more than one element, you can use del with a slice index:

>>> t = ['a', 'b', 'c', 'd', 'e', 'f']  
>>> del t[1:5]  
>>> print (t)  
['a', 'f']

Example 1

name = ['P','Y','T’,'H', 'O', 'N']  
pname = ""  
for letter in name:  
       pname += letter  
print(pname)  
print("Finished...")

Example 2

teams = []  
howmany = int(input("How many teams?  : "))  
for i in range(howmany):  
       teams.append(input("Name of team #" + str(i+1) + " : "))

for team in teams:  
       print(team)  
print("Finished…")

**Week 1: Python Debugger**

**View video here:** <https://www.youtube.com/watch?v=Fegwf_7otLA&feature=youtu.be>

Often, it’s a problem just to find the part of the program that contains an error. Most programming environments come with a debugger, which is a program that can help you find bugs. Typically, your program can be run under the control of the debugger. The debugger allows you to set “breakpoints” in your program. A breakpoint is a point in the program where the debugger will pause the program so you can look at the values of the program’s variables. The idea is to track down exactly when things start to go wrong during the program’s execution. The debugger will also let you execute your program one line at a time, so that you can watch what happens in detail once you know the general area in the program where the bug is lurking.

Here are the commands that are used in the Python Debugger. Some of the terms, like *functions*, you have not covered. But they will be covered later in the course. Use this table as a reference.

|  |  |
| --- | --- |
| **Command** | **Description** |
| Go | Execute the program until it hits a breakpoint or terminates |
| Step | Executes a line of code. If the line is a function, it begins executing each line of the body of the function. |
| Over | Executes a line of code. If the line is a function, it executes the entire function is one step. |
| Out | Executes all of the lines within a function, until the function ends. |
| Quit | Stops the execution. |

Here is a screen capture video showing how to use the Python Debugger. Transcript:

Python Debugger Clean As programs get larger and more complex, it is harder to find defects. A debugger is a diagnostic tool to help understand the internal workings of a piece of software. Normally programs execute lines in fractions of a second. A debugger lets you execute a program one line at a time and see what is happening internally. Here, you'll see a program on the right, it's a simple program. It asks what age the user is. It accepts the age and puts it in a variable called age. It computes the variable double age as age times two and then it prints out if you are whatever age you are now, this is what you'll be, you're halfway to this double age. Then it adds one to eight and prints out what you'll be next year. You can see over here in the left, if you put in 25 it says if you are 25 now, you are halfway to 50, and next year you'll be 26. Let's say we want to, instead of having the program run very quickly -- We want it to run one line at a time so we can see what's happening. To do that, but first we need to turn on the debugger and I would recommend checking all of these check boxes if they are not already checked. Now that the debugger is turned on when I run the program -- It executes it one line at a time. So you notice over here there's nothing displayed and when I want to execute a line, notice here on the right the first line is highlighted. I can click over and the highlight drops to the second line, which means it executed line one and sure enough, line one has been executed. And it says here we're not looking at line two. Let's execute line two. Now before line two can finish, we have to come here and enter something. Let's say the user is five. Now we are on line three. We haven't executed it yet and at the bottom you'll see that there's a variable that's been created called age and the current value is five. So it'll keep track of all your variables and what values they are. Now we'll click over again and you notice age is still five, but now double age is 10. Click over again, it printed the line five on the screen and it says here line five, now, I'm sorry. It printed line four and it says it's ready for now line five. When I click over it'll execute line five. Notice age changed to six and if I click over again, it'll execute line six and the program is finished. But let's say I have a large program and I don't want to sit and click through each line. There's one area of the code, I kind of want to jump to that area of the codes and see what's going on. I can do that by setting what's called a break point. If you pick a line and you right click and say set break point, it'll be highlighted as yellow and you can set multiple break points in a program. And you also can clear a break point. Now when I run the program -- And we'll begin and if I click the go button, it will run every line very quickly, as quickly as it does if you just executed it without the debugger, but when it gets to the break point it will stop. So, for example, 25, so now we've got to line four and we have stopped. And you notice the variable age is 25 and double age is 50 because it's already executed the first three lines of code. And now if I click go again, it'll just run to the end of the program. It'll continue to run until it hits a break point or I can click over and step through. So now we're at this line and if I click over, age will change to 26. And there are other buttons up here and we'll describe in the text what they do.

**Week 2: Conditions**

**Conditional Statements and Boolean Expressions**

For the purpose of this week's lesson, we will focus our attention on conditions and conditional statements. Each programming language's syntax for conditions is a little different, but the underlying concepts are the same. The conditional statements provide the logic that each program uses to evaluate what functionality and subroutines are executed, creating an application program's flow.

All conditions evaluate *Boolean expressions*, which are expressions that can be evaluated to *True* or *False*. The most common expressions are mathematical expressions such as a > 5. Depending upon the value of variable a, this expression could be true or false. Most expressions have two arguments and an operator. In the example above, a and 5 are arguments and > is the operator. Arguments can be variables, such as a, and literals, such as 5. Most programming languages have common math operators, such as:

* equals =
* not equals ≠
* greater than >
* less than <
* greater than or equal to ≥
* less than or equal to ≤

In addition to these operators, there are conjunctions of *and* and *or*. So, you can string multiple expressions into a single expression, such as a > 5 and b < 2. Parentheses ( ) can be used to control order of operations.

((taxesDue > 5000) and ((dependents < 2) or (donations > 1000)))

There is also a not operator, which negates the expression; for example, not(a > 5). If a is greater than 5, it will evaluate to False, otherwise True.

Finally, functions, which we will talk about later in the course, can return True or False and can be used in expressions.

**If Statements**

An *if*statement tells the computer to take one of two alternative courses of action, depending on whether the value of a given Boolean-valued expression is *true* or *false*. It is an example of a "branching" or "decision" statement. An *if* statement has the form:

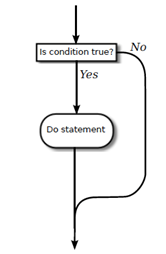
if ( <boolean-expression> )  
<statement1>  
else  
<statement2>

When the computer executes an*if*statement, it evaluates the Boolean expression. If the value is *true*, the computer executes the first statement and skips the statement that follows the "else". If the value of the expression is *false*, then the computer skips the first statement and executes the second one. Note that in any case, one and only one of the two statements inside the *if*statement is executed. The two statements represent alternative courses of action; the computer decides between these courses of action based on the value of the Boolean expression.

In many cases, you want the computer to choose between doing something and not doing it. You can do this with an *if* statement that omits the else part:

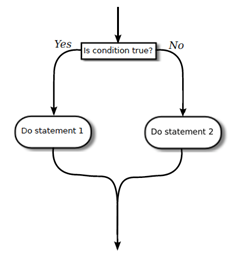
if ( <boolean-expression> )  
<statement1>

To execute this statement, the computer evaluates the expression. If the value is *true*, the computer executes the <statement> that is contained inside the *if*statement; if the value is *false*, the computer skips over that <statement>. In either case, the computer then continues with whatever follows the *if* statement in the program.

**If Statement Flow of Control**  


In these diagrams, the arrows represent the flow of time as the statement is executed. Control enters the diagram at the top and leaves at the bottom. Similarly, a flow control diagram for an *if..else* statement makes it clear that exactly one of the two nested statements is executed:

**If..Else Flow of Control**



Of course, either or both of the <statements> in an *if* statement can be a block.

**References**

This week's content has been adapted from the following resources, which are available under the licenses described below.

Downey, A. (2012). *Think Python: How to Think Like a Computer Scientist*, v. 2.2.20. Needham, Massachusetts: Green Tea Press. Retrieved from [www.thinkpython.com](http://www.thinkpython.com/)  
*Think Python: How to Think Like a Computer Scientist* is used under the terms of the [Creative Commons Attribution-NonCommercial 3.0 Unported](http://creativecommons.org/licenses/by-nc/3.0/) license.

Eck, D. J. (2014, August). *Introduction to Programming Using Java*, v. 7.0. Geneva, New York: Hobart and William Smith Colleges. Retrieved from <http://math.hws.edu/javanotes>  
*Introduction to Programming Using Java* is used under the terms of the [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported](http://creativecommons.org/licenses/by-nc-sa/3.0/) license.

Haschke, P. (2013, January 31). *An Introduction to R*. Rochester, New York: University of Rochester, The Star Lab.  
*An Introduction to R* is used under the terms of the [Creative Commons Attribution 3.0 Unported](http://creativecommons.org/licenses/by/3.0/) license.

**Python**

A Boolean expression is an expression that is either true or false. The following examples use the operator == , which compares two operands and produces True if they are equal and False otherwise:

>>> 5 == 5  
True  
>>> 5 == 6  
False

True and False are special values that belong to the type bool; they are not strings:

>>> type(True)  
<type 'bool'>  
>>> type(False)  
<type 'bool'>

The ==  operator is one of the relational operators; the others are:

x != y # x is not equal to y  
x > y # x is greater than y  
x < y # x is less than y  
x >= y # x is greater than or equal to y  
x <= y # x is less than or equal to y

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols. A common error is to use a single equal sign (=) instead of a double equal sign (==). Remember that = is an assignment operator and == is a relational operator.



In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the "if" statement:

if x > 0:  
print 'x is positive'

The Boolean expression after "if" is called the condition. If it is true, then the indented statement gets executed. If not, nothing happens.

"If" statements have the same structure as function definitions: a header followed by an indented body. Such statements are called compound statements.

There is no limit on the number of statements that can appear in the body, but there has to be at least one. Occasionally, it is useful to have a body with no statements (usually as a placeholder for code you haven’t written yet). In that case, you can use the pass statement, which does nothing.

A second form of the "if" statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

if x%2 == 0:  
print 'x is even'  
else:  
print 'x is odd'

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional:

if x < y:  
print 'x is less than y'  
elif x > y:  
print 'x is greater than y'  
else:  
print 'x and y are equal'

"elif" is an abbreviation of "else if." Again, exactly one branch will be executed. There is no limit on the number of "elif" statements. If there is an else clause, it has to be at the end, but there doesn’t have to be one. Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

There are three logical operators: and , or , and not . The semantics (meaning) of these operators is similar to their meaning in English. For example:

* x > 0 and x < 10 is true only if x is greater than 0 and less than 10.
* n%2 == 0 or n%3 == 0  is true if either of the conditions is true; that is, if the number is divisible by 2 or 3.
* the not operator negates a Boolean expression, so not (x > y) is true if x > y is false; that is, if x is less than or equal to y.

Strictly speaking, the operands of the logical operators should be Boolean expressions, but Python is not very strict. Any nonzero number is interpreted as "true."

>>> 17 and True  
True

This flexibility can be useful, but there are some subtleties to it that might be confusing. You might want to avoid it (unless you know what you are doing).

**Week 3: While Loops**

## While Loops

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

A while loop is used to repeat a given statement over and over. Of course, it’s not likely that you would want to keep repeating it forever. That would be an infinite loop, which is generally a bad thing. (There is an old story about computer pioneer Grace Murray Hopper, who read instructions on a bottle of shampoo telling her to “lather, rinse, repeat.” As the story goes, she claims that she tried to follow the directions, but she ran out of shampoo. In case you don’t get it, this is a joke about the way that computers mindlessly follow instructions.)

To be more specific, a while loop will repeat a statement over and over, but only so long as a specified condition remains true. A while loop has the form:

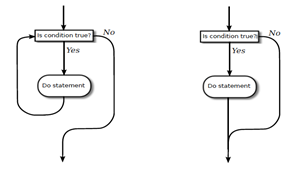
while (<boolean-expression>)  
<statement>

The semantics of the while statement go like this: When the computer comes to a while statement, it evaluates the <boolean-expression>, which yields either true or false as its value. If the value is false, the computer skips over the rest of the while loop and proceeds to the next command in the program.

If the value of the expression is true, the computer executes the <statement> or block of statements inside the loop. Then it returns to the beginning of the while loop and repeats the process. That is, it reevaluates the <boolean-expression>, ends the loop if the value is false, and continues it if the value is true. This will continue over and over until the value of the expression is false when the computer evaluates it; if that never happens, then there will be an infinite loop.

Sometimes, novice programmers confuse while statements with simple if statements (with no else part), although their meanings are quite different. The <statement> in an if is executed at most once, while the <statement> in a while can be executed any number of times. It can be helpful to look at diagrams of the flow of control for while and simple if statements:

**While Loop Flow of Control        If Statement Flow of Control**



## ****References****

This week’s content has been adapted from the following resources, which are available under the licenses described below.

Downey, A. (2012). Think Python: How to Think Like a Computer Scientist, v. 2.2.20. Needham, Massachusetts: Green Tea Press. Retrieved from [www.thinkpython.com](http://www.thinkpython.com/)  
Think Python: How to Think Like a Computer Scientist is used under the terms of the [Creative Commons Attribution-NonCommercial 3.0 Unported](http://creativecommons.org/licenses/by-nc/3.0/) license.

Eck, D. J. (2014, August). Introduction to Programming Using Java, v. 7.0. Geneva, New York: Hobart and William Smith Colleges. Retrieved from <http://math.hws.edu/javanotes>  
Introduction to Programming Using Java is used under the terms of the [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported](http://creativecommons.org/licenses/by-nc-sa/3.0/) license.

Haschke, P. (2013, January 31). An Introduction to R. Rochester, New York: University of Rochester, The Star Lab.  
An Introduction to R is used under the terms of the [Creative Commons Attribution 3.0 Unported](http://creativecommons.org/licenses/by/3.0/) license.

**Python**

Because iteration is so common, Python provides several language features to make it easier.

One is the ***while***statement. Here is a version of *countdown* that uses a *while* statement:

def countdown(n):

while n > 0:

print n

n = nprint

'Blastoff!'

You can almost read the *while* statement as if it were English. It means, “While n is greater than

0, display the value of n and then reduce the value of n by 1. When you get to 0, display the

word Blastoff!”

The body of the loop should change the value of one or more variables so that eventually the

condition becomes false and the loop terminates. Otherwise, the loop will repeat forever, which

is called an *infinite loop*.

In the case of countdown, we can prove that the loop terminates because we know that the

value of n is finite, and we can see that the value of n gets smaller each time through the loop,

so eventually we have to get to 0. In other cases, it is not so easy to tell:

def sequence(n):

while n != 1:

print n,

if n%2 == 0: # n is even

n = n/2

else: # n is odd

n = n\*3+1

The condition for this loop is n != 1 , so the loop will continue until n is 1, which makes the

condition false.

Each time through the loop, the program outputs the value of n and then checks whether it is

even or odd. If it is even, n is divided by 2. If it is odd, the value of n is replaced with n\*3+1. For

example, if the argument passed to sequence is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2,

1.

Since n sometimes increases and sometimes decreases, there is no obvious proof that n will ever

reach 1, or that the program terminates. For some particular values of n, we can prove

termination. For example, if the starting value is a power of two, then the value of n will be even

each time through the loop until it reaches 1. The previous example ends with such a sequence,

starting with 16.

**Example 1**

iVal = 1

iMax = 12

while iVal < iMax:

iVal = iVal + 3

print("Answer " + str(iVal))

**Example 2**

1 = 1

i2 = 0

while i1 < 8:

while i2 < 6:

print("Even Numbers " + str(i2))

i2 += 2

print("Odd Numbers " + str(i1))

i1 += 2

print("Finished…")

**Example 3**

i = 5

print("Loop: ")

while i > 3:

i -= 1

print("Loop finshed…" + str(i))

## 

**Week 4: For Loops**

We turn in this section to another type of loop, the for statement. Any for loop is equivalent to some while loop, so the language doesn’t get any additional power by having for statements. But for a certain type of problem, a for loop can be easier to construct and easier to read than the corresponding while loop. It’s quite possible that in real programs, for loops actually outnumber while loops.

The for statement makes a common type of while loop easier to write. Many while loops have the general form:

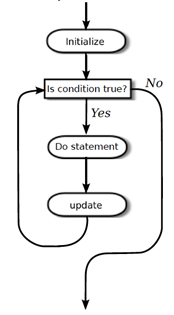
<initialization>  
while (<continuation-condition> )  
<statements>  
<update>

It begins by initializing some variable to make the continuation-condition true.  Then, statements are performed. Somewhere in the body of the loop, an update is made to the variables in the continuation-condition, eventually making it false.

In the for loop, the initialization, continuation condition, and updating have all been combined in the first line of the body of the for loop. This keeps everything involved in the "control" of the loop in one place, which helps make the loop easier to read and understand. The for loop is executed in exactly the same way as the original code: the initialization part is executed once, before the loop begins. The continuation condition is executed before each execution of the loop, and the loop ends when this condition is false. The update part is executed at the end of each execution of the loop, just before jumping back to check the condition.

In general, blocks of code that need to execute 0 or more times should use a while loop.  Blocks of code that need to execute a fix number of times should use a for loop.

**For Loop Flow of Control**

****

**References**

This week’s content has been adapted from the following resources, which are available under the licenses described below.

Downey, A. (2012). *Think Python: How to Think Like a Computer Scientist*, v. 2.2.20. Needham, Massachusetts: Green Tea Press. Retrieved from [www.thinkpython.com](http://www.thinkpython.com/)  
*Think Python: How to Think Like a Computer Scientist* is used under the terms of the [Creative Commons Attribution-NonCommercial 3.0 Unported](http://creativecommons.org/licenses/by-nc/3.0/) license.

Eck, D. J. (2014, August). *Introduction to Programming Using Java*, v. 7.0. Geneva, New York: Hobart and William Smith Colleges. Retrieved from <http://math.hws.edu/javanotes>  
*Introduction to Programming Using Java* is used under the terms of the [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported](http://creativecommons.org/licenses/by-nc-sa/3.0/) license.

Haschke, P. (2013, January 31). *An Introduction to R*. Rochester, New York: University of Rochester, The Star Lab.  
*An Introduction to R* is used under the terms of the [Creative Commons Attribution 3.0 Unported](http://creativecommons.org/licenses/by/3.0/) license.

## Python

In Python, the format of the *for* loop is as follows:

for <variable> in <sequence>:  
<statements>

The for loop normally has a variable that is set to different values for each loop of the <statements>.  A <sequence> defines what values are in the variable in the loop.

There are many different sequences in Python. Here are just a few examples:

A sequence could be an array.

cheeses = [' Cheddar' , ' Edam' , ' Gouda' ]  
for cheese in cheeses:  
print (cheese)

A sequence could be a range. Range is a special function in Python. Range takes 1 to 3 parameters (start, stop, and step).

* A range with 1 parameter like range(3) returns a list of numbers from 0 up to the parameter (0, 1, 2).
* A range with 2 parameters like range(1, 5) returns a list of numbers from the first parameter up to the second parameter (1, 2, 3, 4).
* A range with 3 parameters like range(1, 8, 2) returns a list of numbers from the first parameter up to the second parameter, stepping by the third parameter (1, 3, 5, 7).
* So, a sequence can be a range like this:

for x in range(1, 8, 2):  
print(x)

A sequence can also be records read from a file, but reading files are not covered in this course. So, we won't go into too much detail. When reading a file, you can create a sequence of each line of the file and read them one line at a time.

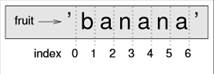
A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a traversal. One way to write a traversal is with a "while" loop:

index = 0  
while index < len(fruit):  
letter = fruit[index]  
print letter  
index = index + 1

This loop traverses the string and displays each letter on a line by itself. The loop condition is index < len(fruit), so when index is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index len(fruit)-1, which is the last character in the string.

Exercise: Write a function that takes a string as an argument and displays the letters backward, one per line.

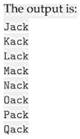
Another way to write a traversal is with a "for" loop:



for char in fruit:  
print char

Each time through the loop, the next character in the string is assigned to the variable char. The loop continues until no characters are left.

The following example shows how to use concatenation (string addition) and a "for" loop to generate an abecedarian series (that is, in alphabetical order). In Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

prefixes = 'JKLMNOPQ'  
suffix = 'ack'  
for letter in prefixes:  
print letter + suffix

Of course, that's not quite right because "Ouack" and "Quack" are misspelled.  The following program counts the number of times the letter "a" appears in a string (banana):

word = 'banana'  
count = 0

for letter in word:  
if letter == 'a':  
count = count + 1  
print count

This program demonstrates another pattern of computation called a counter. The variable count is initialized to 0 and then incremented each time an "a" is found. When the loop exits, count contains the result—the total number of a's.

**Example 1**

iMax = 12  
iTotal = 0

for iVal in range(0,iMax):  
iVal = iVal \* 2  
iTotal = iTotal + iVal  
print("Answer" + str(iTotal))

**Example 2**

for i1 in range(1,6):  
if i1%2==1:  
print("Odd: " + str(i1))          
for i2 in range(0,12):  
if i2%2==0:  
print("Even: " + str(i2))  
print("\*^\*^\*^")  
print("Finished…")

**Example 3**

iTotal = 0  
iMax = 5  
for i1 in reversed(range(0,18)):  
print("Loop: :" + str(i1))  
iTotal += i1  
print("Value: " + str(iTotal))

**View this video:** <https://www.youtube.com/watch?v=4GwQiv5OFPA&feature=youtu.be>

**Week 5: Functions**

## Functions

In this module, we will cover the topic of functions. We use the term function, which is probably the most common term. But there are other terms for the same concept.  These terms are procedure, method, and subroutine.

As you can probably imagine, programs can get large and complex. We have made small programs so far, only a few lines of code. But they can't do very much. In order to create a program to perform useful tasks, we need thousands of lines or even millions of lines of code. Much of the work in creating a program is managing the size and complexity. Functions can be used for both.

One way to break up a complex program into manageable pieces is to use functions.

**• Consists of the instructions for carrying out a certain task, grouped together and given a name
• The name can be used as a stand-in for the whole set of instructions
• As a computer executes a program, whenever it encounters a function name, it executes all the instructions associated with that function
• Functions can be used over and over at different places in the program
• A function can even be used inside another function**

This allows you to write simple functions and then use them to help write more complex functions, which can then be used in turn in other functions. In this way, complex programs can be built step-by-step, where each step in the construction is reasonably simple. Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

A function consists of instructions for performing some task, chunked together and given a name. “Chunking” allows you to deal with a potentially complicated task as a single concept. Instead of worrying about the many steps that the computer might have to go through to perform that task, you just need to remember the name of the function. Whenever you want a program to perform the task, you just call the function. Functions are a major tool for dealing with complexity.

A function is sometimes said to be a “black box” because you can’t see what’s “inside” it (or, to be more precise, you usually don’t want to see inside it, because then you would have to deal with all the complexity that the function is meant to hide). Of course, a black box that has no way of interacting with the rest of the world would be pretty useless. A black box needs some kind of interface with the rest of the world, which allows some interaction between what’s inside the box and what’s outside.

A physical black box might have buttons on the outside that you can push, dials that you can set, and slots that can be used for passing information back and forth. Since we are trying to hide complexity, not create it, we have the first rule of black boxes:

****

****Are there any examples of black boxes in the real world? Yes; in fact, you are surrounded by them. Your television, your car, your mobile phone, your refrigerator. . . .

You can turn your television on and off, change channels, and set the volume by using elements of the television’s interface—dials, remote control, don’t forget to plug in the power—without understanding anything about how the thing actually works.

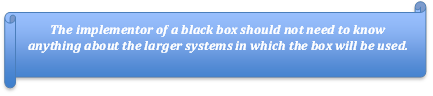
The same goes for a mobile phone, although the interface in that case is more complicated.

Now, a black box does have an inside—the code in a function that actually performs the task, or in our example, all the electronics inside your television set. The inside of a black box is called its implementation. The second rule of black boxes is:

****

In fact, it should be possible to change the implementation, as long as the behavior of the box, as seen from the outside, remains unchanged. For example, when the insides of TV sets went from using vacuum tubes to using transistors, the users of the sets didn’t need to know about it—or even know what using transistors means. Similarly, it should be possible to rewrite the inside of a function, to use more efficient code, for example, without affecting the programs that use that function.

Of course, to have a black box, someone must have designed and built the implementation in the first place. The black box idea works to the advantage of the implementor as well as the user of the black box. After all, the black box might be used in an unlimited number of different situations. The implementor of the black box doesn’t need to know about any of that. The implementor just needs to make sure that the box performs its assigned task and interfaces correctly with the rest of the world. This is the third rule of black boxes:

****

In a way, a black box divides the world into two parts: the inside (implementation) and the outside. The interface is at the boundary, connecting those two parts.

There are four parts to a function:

1. Every function has a name. You use the name when you define the function and when you call it.
2. Every function has a block of code, sometimes called the body of the function. These are the instructions executed when the function is called.
3. Sometimes, functions need additional information. Parameters (sometimes called arguments) are used to pass data into a function.
4. Sometimes, functions need to return data. All programming languages allow functions to return data to the code that calls the function.

****

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

1. Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
2. Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
3. Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
4. Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

# ****References****

This week’s content has been adapted from the following resources, which are available under the licenses described below.

Downey, A. (2012). Think Python: How to Think Like a Computer Scientist, v. 2.2.20. Needham, Massachusetts: Green Tea Press. Retrieved from [www.thinkpython.com](http://www.thinkpython.com/)  
Think Python: How to Think Like a Computer Scientist is used under the terms of the [Creative Commons Attribution-NonCommercial 3.0 Unported](http://creativecommons.org/licenses/by-nc/3.0/) license.

Eck, D. J. (2014, August). Introduction to Programming Using Java, v. 7.0. Geneva, New York: Hobart and William Smith Colleges. Retrieved from <http://math.hws.edu/javanotes>  
Introduction to Programming Using Java is used under the terms of the [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported](http://creativecommons.org/licenses/by-nc-sa/3.0/) license.

Haschke, P. (2013, January 31). An Introduction to R. Rochester, New York: University of Rochester, The Star Lab.  
An Introduction to R is used under the terms of the [Creative Commons Attribution 3.0 Unported](http://creativecommons.org/licenses/by/3.0/) license.

## Python

### Defining a Function

In Python, a function is a named sequence of statements that perform a computation. When you define a function, you specify the name and the sequence of s. Later, you can “call” the function by name.

A function must be defined somewhere. The definition has to include the name of the function, enough information to make it possible to call the function, and the code that will be executed each time the function is called. A function definition in Python takes the form:

def <function-name> (<parameter-list>):  
<statements>

It will take us a while to get through what all this means in detail.

**Statements Parameter**  
The <statements> that are indented after the : in a function definition make up the body of the function. These statements are the inside, or implementation part, of the “black box,” as discussed in the previous section. They are the instructions that the computer executes when the function is called.

If the function has a function whose job is to compute some value, then the value can be returned to the area of code that calls the function. Normally, the return is determined in the last line of the body of the function. The command return is used to indicate what is returned. This also terminates the function, which is why it normally is the last line. For example, if a function determines interest and has the principal and interest rate, the return command could look like this:

return (principal \* interestRate)

**Parameter-List**  
Next, we come to the <parameter-list> of the function. Parameters are part of the interface of a function. They represent information that is passed into the function from outside, to be used by the function’s internal computations. For a concrete example, imagine a program named Television that includes a function named changeChannel(). The immediate question: What channel should it change to? A parameter can be used to answer this question. The declaration of the changeChannel() function might look like

def changeChannel(channelNum):  
...

This declaration specifies that changeChannel() has a parameter named channelNum. However, channelNum does not yet have any particular value. A value for channelNum is provided when the function is called; for example:

changeChannel(17)

Here are a few examples of function definitions, leaving out the statements that define what the functions do:

def playGame():  
...

def getNextN(N):  
...

def lessThan(x, y):  
...

### Calling a Function

When you define a function, all you are doing is telling the computer that the function exists and what it does. The function doesn’t actually get executed until it is called. For example, the playGame() function given as an example above could be called using the following function call statement:

playGame()

This statement could occur anywhere in the same program. More generally, a function call statement for a static function takes the form

<function-name>(<parameters >)

**Function Call Statement Parameters List • The parameter list can be empty, but the parentheses must be there even if there is nothing between them.
• A parameter list can consist of one or more parameter declarations, which are separated by commas.
• The number and types of parameters when calling a function must match the number and type of parameters in the function definition.
**

Here are some examples showing how to call a defined function.

**Calling a Defined Function**

| **Defined** | **Called** |
| --- | --- |
| def printCalendar():     … | printCalendar() |
| def changeChannel(channel):     … | changeChannel(5) |
| def sendWelcomeEmail(name, address):     … | sendWelcomeEmail("Joe", "joe@cool.com") |
| def addNum(x, y):     … | sum = addNum(4, 5) |

**Example 1**

def MyFunction():  
printSomething()  
print ("")  
def printSomething():  
print ("Something will be printed")

**Example 2**

def ReturnFunction():  
s = computeMath(25, 30, " Return Function")  
print (s)  
def computeMath(num5, num7, num9):  
printLine = str(num5+num7) + num9  
return printLine

**Example 3**

def testFunction():  
figures = ["2","0","1","6"]  
printThis(figures)  
print("Finished…")  
def printThis(numbers){  
for x in numbers:  
print(x)

View video example: Old Man had a farm

**Week 6: Packages and Libraries**

## Packages and Libraries

As computers and their user interfaces have become easier to use, they have also become more complex for programmers to deal with. You can write programs for a simple console-style user interface using just a few subroutines that write output to the console and read the user’s typed replies. A modern graphical user interface (GUI), with windows, buttons, scroll bars, menus, text-input boxes, and so on, might make things easier for the user, but it forces the programmer to cope with a hugely expanded array of possibilities. The programmer sees this increased complexity in the form of great numbers of subroutines that are provided for managing the user interface, as well as for other purposes.



Source: Pexels. (n.d.). Used under the terms of the Creative Commons Zero license. Retrieved from https://www.pexels.com/photo/apple-iphone-desk-laptop-287/

Someone who wanted to program for Macintosh computers—and to produce programs that look and behave the way users expect them to—had to deal with the Macintosh Toolbox, a collection of well over 1,000 different subroutines. There are routines for opening and closing windows, for drawing geometric figures and text to windows, for adding buttons to windows, and for responding to mouse clicks on the window. There are other routines for creating menus and for reacting to user selections from menus.

Aside from the user interface, there are routines for opening files and reading data from them, for communicating over a network, for sending output to a printer, for handling communication between programs, and in general for doing all the standard things that a computer has to do. Microsoft Windows provides its own set of subroutines for programmers to use, and they are quite a bit different from the subroutines used on the Mac. Linux has several different GUI toolboxes for the programmer to choose from.

The analogy of a "toolbox" is a good one to keep in mind. Every programming project involves a mixture of innovation and reuse of existing tools. A programmer is given a set of tools to work with, starting with the set of basic tools that are built into the language: things like

* variables
* assignment statements
* if statements, and loops

To these, the programmer can add existing toolboxes full of routines that have already been written for performing certain tasks.

These tools, if they are well-designed, can be used as true black boxes: they can be called to perform their assigned tasks without worrying about the particular steps they go through to accomplish those tasks. The innovative part of programming is to take all these tools and apply them to some particular project or problem like

* word processing
* keeping track of bank accounts
* processing image data from a space probe
* web browsing
* computer games

This is called applications programming.

A software toolbox is a kind of black box, and it presents a certain interface to the programmer. This interface is a specification of what routines are in the toolbox, what parameters they use, and what tasks they perform. This information constitutes the API, or application programming interface, associated with the toolbox. The Macintosh API is a specification of all the routines available in the Macintosh Toolbox.

A company that makes some hardware device—say a card for connecting a computer to a network—might publish an API for that device consisting of a list of routines that programmers can call in order to communicate with and control the device. Scientists who write a set of routines for doing a complex computation—such as solving “differential equations,”—would provide an API to allow others to use those routines without understanding the details of the computations they perform.

These toolkits are often called packages or libraries. Generally, there are classifications of functionality that can be leveraged in your program:

1. There is standard functionality that comes with each programming language.  This functionality is installed when you install the software environment and is available in any program.
2. Another type of code that can be leveraged is code that needs to be imported, but comes with your software environment. It is already included with the software environment, but you need to indicate that you want to use it in your program.
3. The final category is code written by someone who is completely separate from the software vendor. Many people write code and put it on the Internet for people to download and use. In this category, you need to get the code and include it in the software environment. Then, you can import it and use it.

# ****References****

This week’s content has been adapted from the following resources, which are available under the licenses described below.

Downey, A. (2012). Think Python: How to Think Like a Computer Scientist, v. 2.2.20. Needham, Massachusetts: Green Tea Press. Retrieved from [www.thinkpython.com](http://www.thinkpython.com/)Think Python: How to Think Like a Computer Scientist is used under the terms of the [Creative Commons Attribution-NonCommercial 3.0 Unported](http://creativecommons.org/licenses/by-nc/3.0/) license.

Eck, D. J. (2014, August). Introduction to Programming Using Java, v. 7.0. Geneva, New York: Hobart and William Smith Colleges. Retrieved from <http://math.hws.edu/javanotes>Introduction to Programming Using Java is used under the terms of the [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported](http://creativecommons.org/licenses/by-nc-sa/3.0/) license.

Haschke, P. (2013, January 31). An Introduction to R. Rochester, New York: University of Rochester, The Star Lab.  
An Introduction to R is used under the terms of the [Creative Commons Attribution 3.0 Unported](http://creativecommons.org/licenses/by/3.0/) license.

## Python

### Standard Functionality

The Python programming language is supplemented by a large, standard API. Some of the functionality you have already seen, like the range() function. This function is a good example of a standard function. When you used it in previous modules, you did not need to import it. There are hundreds of such functions.

You can find documentation on all standard Python functionality by using this link:

<https://docs.python.org/3/library/index.html>

But some of this needs to be imported, which is covered in the next section.

### Standard Functionality that Needs to Be Imported

Additional functionality is included in Python, but you need to indicate in your program that you want to use it. For example, trigonometric functions need to be imported. Arithmetic is built into Python, and you do not need to import it. But some programs need trigonometric functions. You could code them yourselves, but that would be a lot of work.

Luckily, trigonometric functions have been built for you. You can see a document for them by going to the Python Standard Library page listed below. Select Mathematical Functions.

<https://docs.python.org/3/library/index.html>

To use the trigonometric functions, you need to import them using the import command. In Python, there are two ways to do this.

#### Import Library (#1)

You could put the import command with the module name at the top of your program.

import math

Once you do that, you can access any of the math methods by doing:

math.cos(1)

#### Import Class (#2)

A second way of using this functionality is to specify the function in the import command.

from math import cos

Now, you can use the cos function just like any other function.

cos(1)

### External Functionality

Even through Python has thousands of functions that contain functionality, it may not contain the functionality that you need. In this case, you can write the new functionality from scratch or find someone that has already written it.

These functions are normally stored on the Internet, and you would need to search and find them. Some require an installation, while others just give you code (in the form of .py files) to include in your project. Each should include instructions.

Here is an example. On professor and author John M. Zelle's website (<http://mcsp.wartburg.edu/zelle/python/>) there is a graphics.py file that allows you to create Windows and graphics applications. If you were to download the graphics.py file and put it in the same folder as your Python code, you could use it in a program.

At the top of your program, you would put:

from graphics import \*

Then in your program, you could use it. On his website, Zelle cites an example in which you could create a program that would draw a circle of radius 10 centered in a 100 x 100 window:

win = GraphWin("My Circle", 100, 100)  
c = Circle(Point(50,50), 10)  
c.draw(win)

Using the from and import commands also allows you to create a program using multiple files. You might need some functionality that Python doesn't provide. Let's say that you need the Pythagorean theorem. You might remember from geometry that the Pythagorean theorem states that three lengths (a, b, and c) form a right triangle if and only if the square of the side opposite the right angle (called the hypotenuse) is equal to the sum of the squares of the other sides. That theorem is often expressed as:

****

And let's also assume that you want to use the functionality in many different Python programs. If we put the code in its own file and use the from and import commands, we gain a number of benefits.

1. We only need to write the code once.
2. If we need to change it, we only need to change it in one place.
3. It makes our programs smaller, thus easier to read.

Our function might look like this and we would save it in a file named Pythagorean.py:

def rightTriangle(a, b, c):  
if (a\*\*2 + b\*\*2 == c\*\*2):  
return True  
else:  
return False

If we put this into a file and run it, the function will get loaded into the Python environment. We can test it by typing in:

rightTriangle(1, 2, 3)

It will return False. Then, we could type in:

rightTriangle(12, 35, 37)

And it will return True.

Now to use it in other programs, you can use the from Pythagorean import rightTriangle command. Inside this script, you can use the new function that you created. This is a good way to leverage code that you write or locate from others.

# ****Reference****

Zelle, J. M. (n.d.) Simple Graphics Library: graphics.py. In Python Programming: An Introduction to Computer Science (website). Retrieved from <http://mcsp.wartburg.edu/zelle/python/>   
The open-source software is licensed under the free [GNU General Public License](http://www.gnu.org/licenses/gpl.html).

**Week 7: Packages and Libraries Documentation**

In addition to packages and libraries, most programming languages have a repository of documentation on the language. This documentation is searchable and updated by the community. In many ways, these repositories are more useful than a book, because they are

* searchable
* contain more information than can be placed in a single book
* constantly updated

You should become familiar with repositories for your language. It will be helpful in the future.

In addition to these repositories, there are independent community sites that can provide additional information. In many cases, they have discussion forums where you can post questions that may be answered by more experienced developers.

Programming languages are simple in some respects and complex in others. All programming languages have a handful of basic commands and concepts like the ones that we have studied in this course. The packages and libraries that come with programming languages all use these basic commands. But packages and libraries provide tremendous functionality and can save us time and money while programming.

Most programming languages have thousands of packages and libraries. This can create a problem for programmers. Programmers do not have the mental capacity or time to learn them all. So, we don't! Good programmers learn to quickly search and find packages and libraries when they need functionality. Naturally, some programmers have certain packages and libraries memorized through the repetition of using them often.

Documentation of Packages and Libraries

The official document for Python is called Python Docs. https://docs.python.org/3/

Here is also a very good Python forum:

https://python-forum.io/