



Programming in Python I

Notes

Table of Contents

Python Features Overview.....	11
Key Features.....	11
Python's Programming Features	15
Python.org	16
About.....	17
Downloads	18
Documentation	19
Community Section	20
Other Web Resources	21
Programming Language Compatibility.....	22
Python and Other Languages	23
Languages	24
Variable.....	25
Scenario 1.....	25
Scenario 2.....	25
Scenario 3.....	25
Variable Naming.....	27
Program 1.....	29
Program 2.....	29
Assignment Operator.....	31
Python's Math Operators.....	31
Symbol.....	31
Operation.....	31
Comments	37
Line Continuation Character.....	38
Syntax	40
The Print Function	41
The Bin Statement.....	42
In Anaconda	44
Overview.....	46
Boolean Expressions	48

Relational Operator	48
Meaning	48
Logical Operators	54
Operator	54
Meaning	54
Boolean Variables	58
Introduction	59
Indexing of a String	60
Len Function	62
Concatenation	63
String Slicing	63
Python Code	64
Result	64
"In" and "Not In" Operators	65
String Testing Methods	67
String Testing Method	67
Result	67
String Modification Methods	69
String Modification Method	69
Result	69
String Search and Replace Methods	70
String Search and Replace Methods	70
Results	70
The Input Function	72
Common Type Conversions	73
Meaning	73
The Print Function	74
Statement	74
Output	74
Python Code	75
Result	75
Python Code	75

Result.....	75
Python Code.....	76
Result.....	76
Python Code.....	76
Result.....	76
Python Code.....	77
Result.....	77
Python Code.....	78
Result.....	78
Python Code.....	79
Result.....	79
Python's Escape Characters.....	79
Escape Character	79
Effect.....	79
String Concatenation in the Print Function	83
Python Code.....	83
Result.....	83
String Formatting.....	88
Formatting Floating Point Values	89
Python Code.....	89
Result.....	89
Formatting Integer Values.....	91
Python Code.....	91
Result.....	91
Formatting String Values.....	92
Python Code.....	92
Result.....	92
Combination of Format Specifiers	92
Looping Structure	94
Background	95
The Primary Loop Statements	97

Overview.....	100
Python Example.....	102
Overview – The While Loop	104
For Loops.....	108
General Form	109
Index.....	119
Value.....	119
Overview.....	120
Overview.....	126
Local Variables	126
Global Variables	131
Overview.....	133
Definitions.....	133
Global Function Example	134
Local Function Example.....	135
Lambda Function Example	136
Methods Example.....	137
Overview.....	138
Overview.....	140
Overview.....	145
Regular Function	145
Lambda Function	145
Overview.....	147
Overview.....	153
Overview.....	155
Overview.....	159
Creating a Module	159
File Handling Overview	163
Opening a File: General Format.....	166
Writing to a File: General Format	168
Reading from a File – Entire Content: General Format.....	169
Reading from a File – Line by Line: General Format.....	169

Detecting End of File Using While.....	171
Detecting End of File Using For Loop	172
Record Structure	172
Modifying an Existing File	174
Overview.....	176
Sequence Operators	176
Indexing	177
• start defaults to 0	178
• end defaults to the length of the list	178
• increment defaults to 1	178
Concatenation	181
Checking for Membership	181
Repetitions.....	183
Lists Overview	184
Creating Lists	185
Functions for Lists.....	190
Out of Bounds	190
For Loop	192
Alternative for Loop.....	193
List Slicing.....	193
+ Operator.....	194
* Operator.....	195
in/not in Operator.....	196
List Methods.....	196
Copying a List	200
Functions and Lists	200
Multidimensional Lists.....	202
Overview.....	203
Why Use a Tuple?.....	203
Creating Tuples.....	204
Tuple Operations.....	206
Operation.....	206
Meaning.....	206
Tuple Indexing and Slicing.....	209
Index	210
Value.....	210
No Enclosing Delimiters	211
Updating Tuples	212

Delete Tuple Elements	212
Higher Order Functions Overview	214
Overview.....	217
Dictionary: Creation.....	218
Dictionary: Adding and Modifying Entries.....	219
Dictionary: Deleting Entries.....	220
Dictionary: Retrieving Values	220
Dictionary: Looping Through all Entries Example	222
Dictionary: Determining the Size of the Dictionary.....	222
Dictionary: Checking to see if the Dictionary Contains a Certain Entry	223
Dictionary: Methods.....	224
Overview.....	225
Types of Exception Handling	226
Programmer-defined	226
System-defined	226
Assertions.....	226
Exception Handling – Usage	227
Common Errors	228
Outside Components	228
Real Life Example:.....	229
Example:.....	230
Example	232
Exception Handling – Control Flow Mechanism	234
The Overuse of Exception Handling to Control the Flow of a Program.....	236
Pros to the use of Exception Handling Control the Flow.....	237
Overview.....	239
Overview.....	243
Theoretical Approach to Efficiency	245
Linear Search.....	245
• If the search value is not on the list, then n searches have to be performed.....	245
• If the search value is not on the list, then $n/2$ searches on average have to be performed.	245
What is C?	248
Overview.....	249
Test Design Techniques.....	250
Overview.....	252
Test Design Techniques.....	255
Overview.....	256

• Although there are more clever ways to implement database lookups, I always programmed it in the simplest manner.....	257
• If you are developing your own algorithms, then the keep-it-simple idea may not hold but try to follow this principle as much as possible.....	257
• Another pair of eyes also means that you set aside the program and take a break from the examination.....	258
• Work on some other project or get a good night's sleep! Sometimes then the problem is obvious.	258
Overview.....	260
Defining Classes – General Form	261
Accessing Instance Variables.....	264
Methods.....	266
Overview.....	269
Types	272
Polynomial	273
Trinomial.....	274
Overview.....	278
Abstraction.....	278
Encapsulation	280
Data Types and Objects	280
Data Attributes.....	281
Overview.....	282
Overview.....	286
Sample Mean	286
Programming Considerations	288
Median	288
Programming Considerations	290
Programming Considerations	291
Programming Considerations	292
Standard Deviation.....	293
Variance (S^2) = average squared deviation of values from mean	293
Standard deviation (S) = square root of the variance	293
Think it through!.....	293
Functions.....	294
Measures of Spread.....	294
Don't recreate the wheel!.....	294
Built-in Functions.....	295

Overview.....	297
Programming Considerations	299
Probability Simulations.....	300
Programming Considerations	301
Programming Considerations	301
Programming Considerations	303
Critical Values.....	304
Programming Considerations	305
Classical Approach (By Hand)	306
Programming Considerations	307
Linear Correlation Coefficient	308
Correlation Coefficient.....	309
Correlation	310
Formulas	310
Use.....	310
Linear Regression Equation	311
Simple Linear Regression	311
Formulas	311
Meanings	311
Confidence Intervals	314
Estimation	315
NumPy and SciPy	316
Mean Confidence Interval in Python	317
Overview of Hypothesis Testing	318
<ul style="list-style-type: none">• Null hypothesis – Statement regarding the value(s) of unknown parameter(s)(will always contain an equality), Represented by H_0.....• Alternative hypothesis – Statement contradictory to the null hypothesis (will always contain an inequality), Represented by H_1.....• Quantity based on sample data and null hypothesis used to test between null and alternative hypotheses	318
Hypothesis Testing	319
Formulas	319
Terms	320
<ul style="list-style-type: none">• α is the probability of Type I error• β is the probability of Type II error	321

Standard Normal Distribution	321
---	------------

Notes: Overview of Python Features

Python Features Overview

Python was designed for simplicity but a programming language needs to have certain basic programming features. Some of the key features are listed in the next section. For a full discussion of these features, visit [python.org](https://wiki.python.org/moin/BEGINNERSGUIDE/Overview):

<https://wiki.python.org/moin/BEGINNERSGUIDE/Overview>

Key Features

1. **Uses a syntax that makes the program readable by humans.** There is a reason why the Python site lists this feature first: Many languages are very difficult to interpret straightaway.

Consider this piece of code (not any specific language):

```
a = 1000  
b = 0.05  
c = a * b  
print("sales tax was ", c)
```

Compared to this code:

```
sales = 1000  
tax_rate = 0.05
```

```
sales_tax = sales * tax_rate  
print("sales tax was ", sales_tax)
```

This comparison shows how it is possible to make a program easier to read simply by ensuring an effective use of variable names. Syntax can also make a program more readable. Consider this APL program:

```
x [□x 6?40]
```



This program generates 6 random numbers from 1 to 40 that are non-repeating and sorted from smallest to largest. Even if the variable name was not x, then the syntax makes this code very difficult to read.

2. **Syntax is simpler than other popular high-level languages.** Besides making programming easier for beginners, it allows for rapid development. “Simpler” in this case means increased productivity and a reduction in errors.
3. **Has a large standard library.** This simplifies and speeds up coding, reducing development time. If you have a particular function that you would like to perform in Python, there probably is a library that would simplify the coding; that is, someone else has already done the testing and research for the code and made it available as a packaged library.
4. **As an example.** I was debugging a socket Perl program for a website on a Windows platform that sent emails. The program had to be fixed before I could go home. The company was potentially losing sales leads. After six hours, I asked if the program had to be written in Perl using sockets.

Based on their reaction, these individuals did not even know what Perl or sockets were used for. I found sample ASP code for sending an email (not a library) and had the website working in twenty minutes. From a business standpoint, they would have liked the idea to have come up six hours ago due to potentially losing customers. Libraries are very similar to this example. A company would prefer you to use a pre-constructed library versus writing the code from scratch. Libraries may be a bit tricky to set up in your environment but, once they are, it can drastically reduce the development time of your project!

5. Python's interpretative nature allows easy testing and debugging.

Compiled languages are often difficult to test and debug, mainly because code must be compiled before it can be tested. Syntax errors are fairly easy to identify and correct. Logic errors, on the other hand, can be challenging as well as testing. Python's interpretative nature allows a line-by-line testing and debugging.

6. Easy integration of modules implemented in a compiled language. It can be a challenge to bring in new languages into an organization. Consider if all of your systems are written in C including the payroll component. Consider if the desire is to add a new feature into Payroll but have it programming in Python. If there is no way to interface programming languages, the new feature will have to be done in C. Without an appropriate interface, there would be two options: 1) Rewrite the entire system in Python (costly and not practical) or 2) Write the new feature in C (which is why many companies continue to program in the same archaic language).

- 7. Can be embedded into an application that allows a programmable interface.** This feature again allows for greater compatibility between different systems.
- 8. Is designed to run on many platforms.** Many programs are discovering that applications need to be run on different operating systems/hardware. For example, the developers of YouTube would like their videos to run on every operating system/hardware.
- 9. Is available under an open source license and has free versions.** Open source has many nice features that are not included in the scope of this course. For further reading on the benefits of open source:

<http://www.cioinsight.com/it-strategy/application-development/slideshows/nine-advantages-of-open-source-software.html>

Python's Programming Features

The following is a list of Python's Programming features:

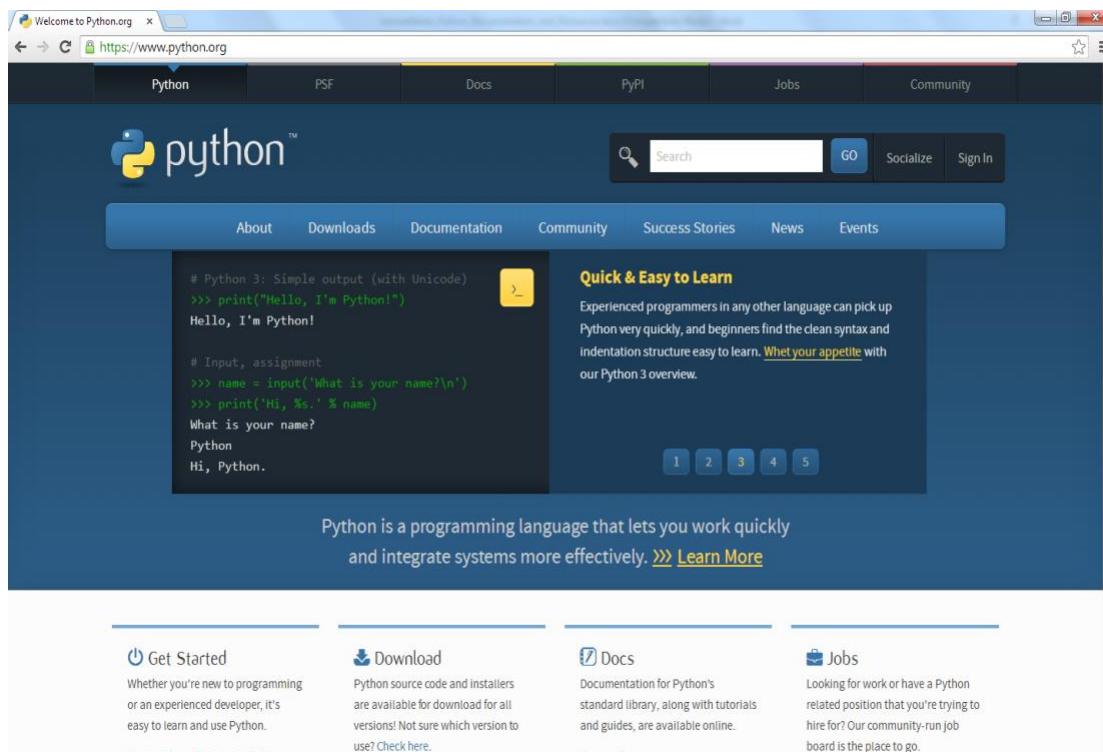
1. Many basic data types, including numbers (floating point, complex, and unlimited-length long integers), strings, lists, and dictionaries.
2. Object-oriented programming, including classes and multiple inheritances.
3. Modules and packages to allow re-use of code.
4. Exception handling, including raising and catching exceptions.
5. Data types are strongly and dynamically typed.
6. Generators and list comprehensions.
7. Excellent memory management that is automatically handled.

The above items can be found in other programming languages. The website is pointing out these features to emphasize that, while Python has many features that simplify coding, it also has the same programming features as more complex languages.

Notes: Python Documentation and Resources

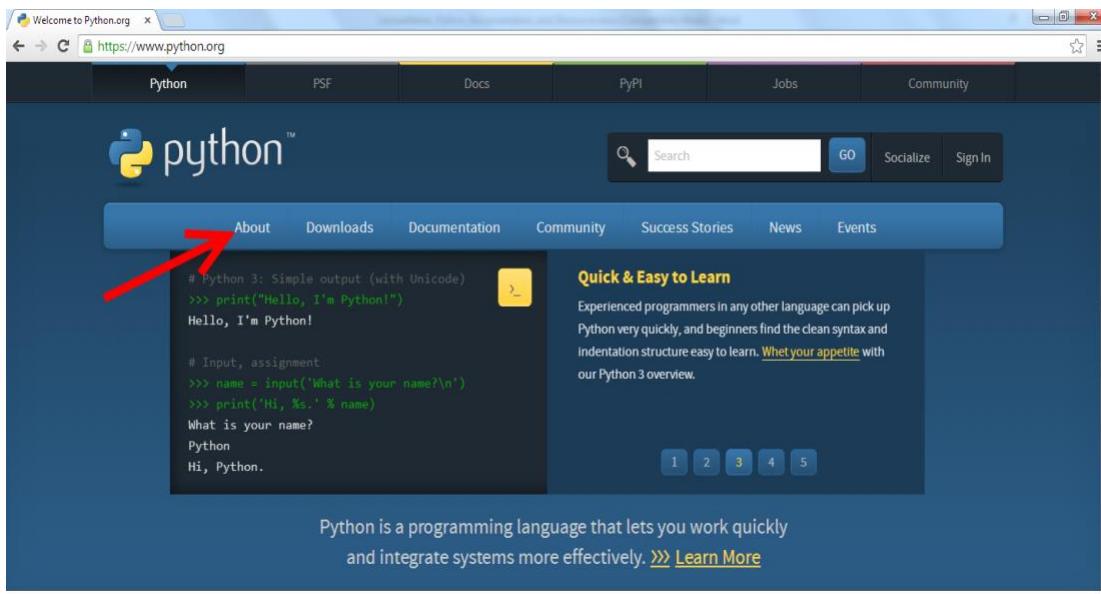
Python.org

Python documentation and resources can be found on the main Python website:
www.python.org



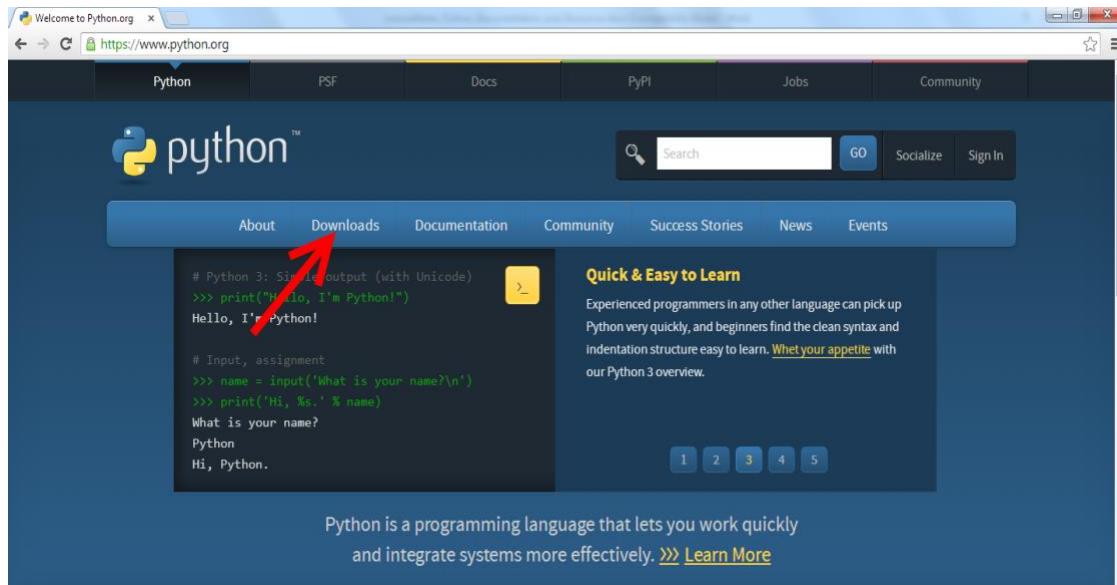
Keep in mind that this website might change on a regular basis but the main resources that are presented here should be present somewhere on the site.

About



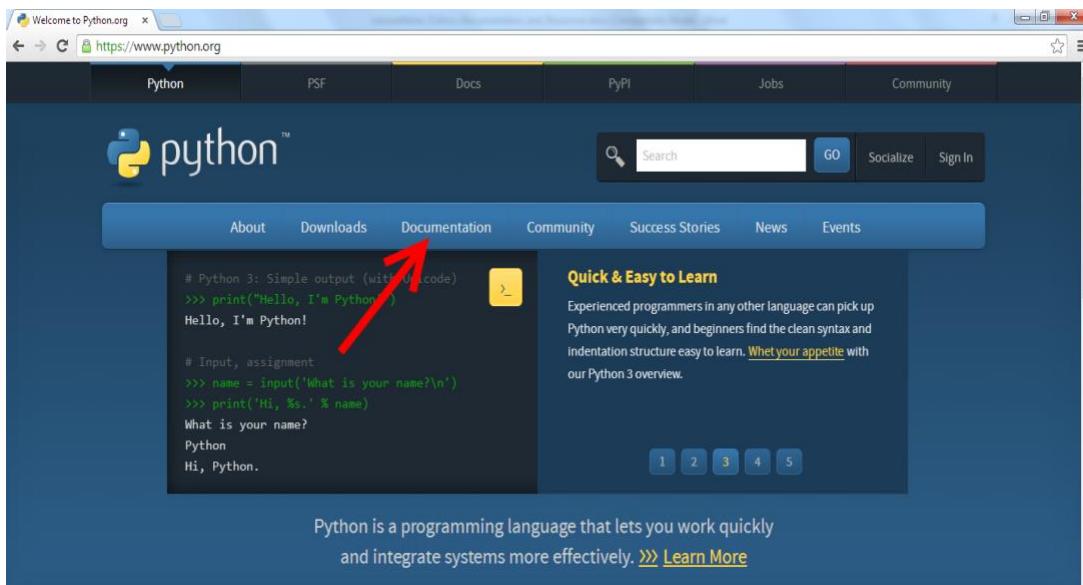
The About section has some starting information on this website and the resources available, but the quotes section is the most interesting part of this section. Here is a quote from <https://www.python.org/about/quotes/>: “Python is fast enough for our site and allows us to produce maintainable features in record times, with a minimum of developers,” said Cuong Do, Software Architect, YouTube.com. There are many interesting quotes on this website to point out the power of Python.

Downloads



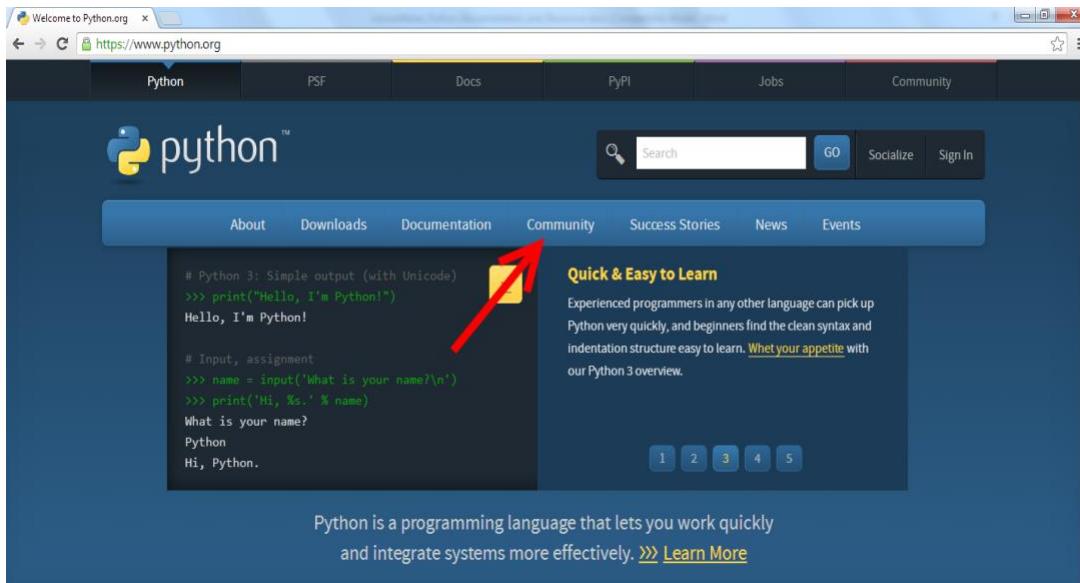
In this course, Anaconda software will be used but, if you wish to install all the libraries by hand, then the download section will give you the basic Python software to get started. In this section, you will find many different releases of the software. Realize that old releases are kept on the site because some of the newer releases may have software bugs.

Documentation



This is a very useful section for Python documentation. For this course, the Python 3.x Docs will be your best resource. If you are struggling with the syntax or how to use a particular function, then this page should be your starting point. Sometimes doing a web search on www.google.com will help with syntax, but the code found might be written in Python 2.x and, **unfortunately, Python 3.x is not backward compatible**, thus the code may not work. This can be frustrating. Watch your versions of documentation and examples, whether it is 2.x or 3.x.

Community Section



Open source does not necessarily mean that the software is well designed or supported. SourceForge (<http://sourceforge.net/>) is a repository of open source projects. Many of these projects (software) are not updated on a regular basis and do not have community support. Imagine a software product that is being used in a business on a regular basis. If there is a software issue, then it can be frustrating if there is no support. One huge advantage of Python is that the software and many of the libraries are well supported. If you have difficulty, you can post your problem in the Community section and get help. Once you develop a comfort level, be sure to support others by contributing!

Other Web Resources

- Python FAQ: <https://docs.python.org/3/faq/general.html#how-do-i-obtain-a-copy-of-the-python-source>
- Python Beginner's Guide Wiki:
<https://wiki.python.org/moin/BeginnersGuide>
- Learn Python Tutorial: <http://www.learnpython.org/>
- The Hitchhiker's Guide to Python: <http://docs.python-guide.org/en/latest/intro/learning/>
- Code Academy, Python: <http://www.codecademy.com/en/tracks/python>

Notes: Python Integration with Other Programming Languages

Programming Language Compatibility

One challenge for a business is the incompatibility of different programming languages. Consider a company that has spent the last five years programming their systems in the C Programming language. For this example, let's assume that the payroll piece is written in C. The payroll department wishes to add a new module for investment options, so they contact the information systems department. The information systems department is happy to implement the new module for them, but the vision is to eventually write everything in the Java programming language.

Perhaps the C code is running on a Windows server and the Java piece needs to be on the server's intranet. The first major hurdle is that these programming languages do not seamlessly interface with each other: C cannot call Java functions, and Java cannot call C functions, without modification. Another potential problem is the hardware component, which can be tricky to solve, as certain languages are designed to run on specific types of machines. This is especially true for newer language system requirements, and companies often ignore this problem by only using newer languages on brand new systems. One particular system, used by a company, had a module of their purchasing system written in Paradox for DOS. The system did not interface with any other system due to this lack of integration. For twenty years this system was used "as is" since

the company did not want to rewrite it from scratch. In 2012, the Paradox for DOS system finally disappeared as the company purchased/installed SAP (Systems, Applications & Products in Data Processing).

Visual Studio was a major attempt at solving this integration dilemma. The hope was that programmers could program in whatever language they wish and then these pieces of code would be able to communicate. Instead of compiling to machine language, it would be compiled to Microsoft Intermediate Language (MSIL) but it would require the .NET framework on the target platform in order to run successfully. This original concept has changed over time and many benefits and flaws of Microsoft's product could be listed, but that is beyond the scope of this class. Visual Studio had many other goals besides the integration with other programming languages.

Python and Other Languages

Python was designed to have the hooks for allowing an easier integration. Keep in mind that this is still not a simple task. The following URL contains information on some of the projects in the work on simplifying integration with other languages:

<https://wiki.python.org/moin/IntegratingPythonWithOtherLanguages>

The languages listed on this page are:

Languages	
<ul style="list-style-type: none">• C• C++• Delphi• Fortran• Lisp• Prolog• Java• C# / .NET	<ul style="list-style-type: none">• Perl• PHP• R• Objective-C• Tcl• Lua• OCaml• Eiffel• Haskell

Some of these open source projects are established and supported, while others are weak. For example, RPy has not seen a release since October 21, 2012, as of this writing. Compare that to the Jython project, which had a release just three months ago as of this writing.

Keep in mind that integration is not an easy task and, if you need to integrate various programming languages with Python, then do the appropriate research and choose a “mature”, supported, open source project. It will save you many headaches in the long run.

Notes: Basic Elements of Python

Variable

A variable is a programmer-defined name that represents a data value in a computer program. In some programming languages, the variable has to be declared before it can be used. In Python, the assignment of a value to a variable name not only declares it but also assigns it with a value.

```
salary = 1000
```

This declares the variable called salary and assigns it a value. The type of data on the right side of the equals determines the data type (integer, float, etc.).

Compare these three scenarios:

Scenario 1	Scenario 2	Scenario 3
<pre>salary = 1000 print(salary)</pre> <p>Result: 1000 The type of this variable is an integer</p>	<pre>salary = 1000.0 print(salary)</pre> <p>Result: 1000.0 The type of this variable is a float (has decimals)</p>	<pre>salary = 'lots of money' print(salary)</pre> <p>Result: lots of money The type of this variable is a string</p>

This can be confusing, but a programmer must put thought into the data type of the variable and the operations.

Note: The print statement displays whatever is inside the parenthesis on the screen; in this case the value of the variable.

Example:

```
num1 = 3  
num2 = 2  
print(num1 / num2)
```

Perhaps the programmer wants to perform integer division. In this case, example 3 divided by 2 would give 1. But the result that is obtained in this program is 1.5. This will be examined in more detail later in the course, but realize that assumptions about data and types should be verified with proper testing!

Variable Naming

Certain restrictions exist on the naming of variables:

1. No Python keywords. For example, “print” is a keyword so you cannot use it as a variable name. Some of the keywords are listed on this website:
https://docs.python.org/3/reference/lexical_analysis.html#keywords

Python is case sensitive, so a programmer could have a program similar to this:

```
Print = 5  
print(Print)
```

This would execute properly and give a result of 5. It is very confusing though, as the only difference between the keyword and the variable is the letter case of the P. If the code looked like this:

```
print = 5  
print(print)
```

This gives the following error message “TypeError: 'int' object is not callable”. Best practices of programming indicate not to use any variable name close to a keyword. It makes your program very confusing.

2. No spaces in the variable name. It might be acceptable in the naming of a word document, for example, but variables cannot have spaces.
3. The first character must either be a letter of the alphabet (uppercase or lowercase) or an underscore (_), though underscores are usually reserved for internal functions – more on those later in the course.
4. Remember that Python is case-sensitive so declaring variables named “salary” and “Salary” would be separate because of the use of the lowercase “s” versus uppercase “S”.

Coding Tip: Variable names are critical to creating a program that self-documents its purpose.

Consider these two programs:

Program 1	Program 2
<pre>salary = 1000 tax_rate = 0.05 tax = salary * tax_rate print(tax)</pre> <p>The print statement could include a bit more about the purpose of the program, but its goal is still clear.</p>	<pre>a = 1000 b = 0.05 c = a * b print(c)</pre> <p>The poor usage of variable names causes the purpose of this program to remain cryptic. Even still, it accomplishes the same task as the program on the left.</p>

I actually saw one program running in production at a company that was programmed similarly to the code on the right. It was actually worse, using the “goto” structure to control the flow of the program and it was not even a structured use of “goto” (which was acceptable use at that company). I could not figure out how to make the correct change that the business unit wished to implement. I ended up rewriting the program from scratch. It took less time to rewrite the code as it would have to modify the existing code. This poorly structured type of code is referred to as “spaghetti code.” The same idea as spaghetti being intertwined and going to random locations is how spaghetti code is designed. Structured programming and proper programming practices will be emphasized in this course.

Coding Tip: Proper use of the underscore and uppercase/lowercase will improve the readability of a program. For example, compare a variable called prevyearssales compared to `Prev_Years_Sales`. Keep in mind that the variable name should be kept as short as possible to achieve readability. You might be able to name `Prev_Years_Sales` as `This_is_the_Previous_Years_Sales` but it would be overkill and complicate the program instead of making it easier to read!

Assignment Operator

The assignment operator is a single equals (=).

Example:

```
tax = salary * tax_rate
```

This code retrieves the values of salary and tax_rate, then multiplies them together, assigning the resulting value to the variable called tax. The right-hand side of the assignment operator follows the order of operations. The left-hand side of the assignment operator must be variable!

Python's Math Operators

Coding Tip: If you do not remember the order of operations (many programmers do not remember), then use parenthesis to indicate an operation that should be done first.

Symbol	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder

**	Exponent
----	----------

Example:

Formula for adding sales tax to the order amount

If you are concerned that the program will first add `order_amt + order_amt`, and then multiply it by `sales_tax_rate`. You can then rewrite the program like so:

Example:

*Total = order amount + (order amount * sales tax rate)*

The parenthesis will make sure that it multiplies `order amt * sales tax rate` first!

In case you are curious, the original statement was fine. Order of operations handles multiplication before addition. A business would probably prefer extra parenthesis to avoid errors. Programming or math errors could potentially cost a company millions of dollars. As an example, the Mars Climate Orbiter went into the atmosphere of Mars and was destroyed due to a simple English/metric conversion error. The estimated cost of this error was 193.1 million dollars.

Example:

$$\frac{\text{jan_sales} + \text{feb_sales}}{2}$$

would be written in a computer program as

`(jan_sales + feb_sales) / 2`

Example:

Consider this interest formula:

$$A = P \left(1 + \frac{r}{n}\right)^{nt}$$

Coding Tip: If there is more than a single number or variable in the exponent then it is also necessary to put parenthesis around the exponent.

would be written in a computer program as:

`A = P * (1 + r / n)**(n * t)`

A variable next to a variable does not indicate multiplication in programming. For example, nt does not mean n times t; instead, the computer program will think this is a new variable called nt. Also, a number or a variable next to the parenthesis in a formula must be programmed with a * to indicate multiplication in the actual computer program.

Coding tip: Do not assume that since the functionality of the current version of the programming language is designed in a certain manner it will always function in the same manner in future versions. For example, consider the following code:

Whenever possible, control the flow of the program using parenthesis or type

Example Python 2.7:

```
num = 3 / 2  
print num
```

Gives the result of 1

Example Python 3.4:

```
num = 3 / 2  
print(num)
```

Gives the result of 1.5

conversions, which will be examined in due course. If the Python program that you are running is for short-term use, these considerations do not typically have to be examined closely; however, if the program is for long-term use, any changes that are introduced in the new versions implemented by the computer department may significantly impact your programs and you should perform comprehensive testing after the upgrade. This is not the same as upgrading to a new version of a word processor!

How do I know this? I was surprised Python had changed integer division between 2.x and 3.x. I performed a Google search of “how to do integer division in 3.0?” and my first result gave me the answer. Do not assume, as a programmer, that you are expected to know all the aspects of programming. You must be willing to explore all the resources available. This does not mean that you contact your instructor every time that you have a question. I once had a computer “expert”

Example Python 3.4

Integer Division

```
num = 3 // 2  
print (num )
```

Gives the result of 1

explain what an “expert” represents in any field. It simply means someone who knows more than you. Certifications in certain computer fields may indicate a

level of expertise but do not mean the individual knows everything. A better view of a true “expert” is that they know where to find the answer!

Comments

Comments are not run by the interpreter but are included in the program to document important items that will help the programmer (or any programmer) to understand the purpose of the code at a later time. A single line comment is indicated by a #, such as this:

Example:

```
# This is a single line comment
```

Multi-line comments can be created by using three double quotes to denote the start of the comment area and three double quotes to denote where it ends.

Example:

```
8# This is a single line comment
9"""
10num = 3 // 2
11print (num )
12"""
13num = 3 / 2
14print (num)
```

This is from Anaconda. Note that the single line comment is grayed out, while the multiple line comment is in green. This particular comment is code that I do not want the Python interpreter to run. This illustrates the most common use of multi-line comments. Instead of deleting the code, it can be commented out so you can refer to the old code or, perhaps, use it if the new code does not work. It can be an invaluable testing and debugging strategy, which we will look at later in this course.

Line Continuation Character

Sometimes the line becomes too long and it is beneficial to use the line continuation character, which is a backslash (\).

For example, the following code:

```
num1 = 2
num2 = 8
num3 = 10
num4 = 20
print("The first number is ", num1, " and the second number
      is ", num2, " and the third number is ", num3, " and the fourth
      number is ", num4)
```

Could be written using the line continuation character to improve readability:

```
num1 = 2
num2 = 8
num3 = 10
num4 = 20
print("The first number is ", num1,
      " and the second number is ", num2,
      " and the third number is ", num3,
      " and the fourth number is ", num4)
```

Notes: Python Syntax

Syntax

Syntax is the form that the various language elements can take. It describes exactly how to write the commands. Consider this syntax form for the print statement from the documentation pages of Python (located at <https://docs.python.org/3.4/index.html>):

```
print('objects', sep='', end='\n', file=sys.stdout, flush=False)
```

Print objects to the stream `file`, separated by `sep` and followed by `end`. `sep`, `end` and `file`, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by `sep` and followed by `end`. Both `sep` and `end` must be strings; they can also be `None`, which means to use the default values. If no `objects` are given, `print()` will just write `end`.

The `file` argument must be an object with a `write(string)` method; if it is not present or `None`, `sys.stdout` will be used. Whether output is buffered is usually determined by `file`, but if the `flush` keyword argument is true, the stream is forcibly flushed.

Changed in version 3.3: Added the `flush` keyword argument.

These are sometimes difficult to read. The more that you program in Python and refer to the syntax, the more these will make sense.

The Print Function

Consider this example:

```
print("happy", "days")
```

The result will be: happy days

Notice in the syntax of the print function that it has an option for **sep**. This is what is known as a keyword argument, and defines the **separator** between the two items. By default, this is a space, but anything can be specified as in this modification:

```
print("happy", "days", sep=":")
```

The result will be: happy:days

While this may not seem to make sense, consider if we wished to put the data in CSV format (comma separated value), then we would want commas between the data and would change the code to look like:

```
print("happy", "days", sep=",")  
print("sad", "days", sep=",")
```

The result will be:

```
happy, days  
sad, days
```

This could be Edit-Copied and pasted directly into Excel!

Notice that the “**end**” option is not specified, but defaults to “\n,” which causes the new line to occur, equivalent to hitting the Enter button in a text editor.

The Bin Statement

Consider another example of syntax for a function called bin:

bin(x)

Convert an integer number to a binary string. The result is a valid Python expression. If x is not a Python `int` object, it has to define an `__index__()` method that returns an integer.

This syntax is straightforward and easy to understand. It appears to convert an integer number to binary. Consider this example code:

```
print("Decimal to Binary Conversion Program")  
num = int(input("Input an integer: "))  
print("The binary number is %s" % bin(num))
```

Without any testing, the syntax told us that the function will “convert an integer to a binary string.” Since it said “string,” the `%s` format specifier was used. The result was:

Decimal to Binary Conversion Program

Input an integer: 153

The binary number is 0b10011001

The result was not entirely expected. With an input of 153, the answer desired was 10011001 but it placed an 0b on the front, which is Python's way of specifying that the number is binary (rather than the integer ten million eleven thousand and one). This 0b, of course, can be removed using string functions if desired.

The syntax can be invaluable in helping figure out how to write a statement. In Python 2.x there was no print function, as it was actually a print statement. It could be that you will be faced with working with a different version of the interpreter and the syntax will help guide you on the changes.

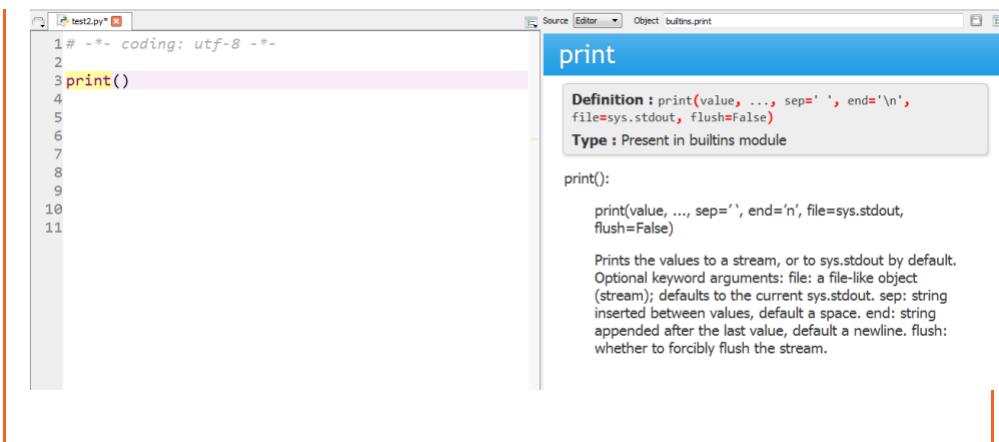
In Anaconda

Consider another example of a syntax for a function called “bin”:

Anaconda has a nice feature. When you type in the Python language element, a popup with the appropriate syntax will be displayed:

```
2
3 print()
4
5 Arguments
6     print(value, ..., sep=' ', end='\n',
7           file=sys.stdout, flush=False)
8
9
10
11
```

However, this is only semi-useful since it disappears from the screen. If you place your cursor on the Python language element and hit CTRL-I, the syntax will appear in the development environment as in this example:



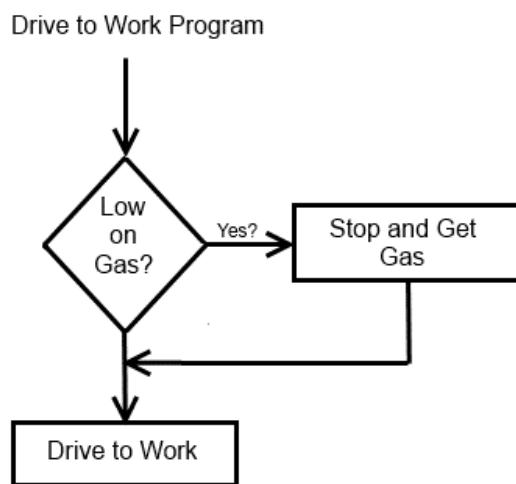
Not all Python editors have this feature so it is important to be able to access the documentation on the Python website.

Notes: Branching Programs

Overview

```
If car is low on gas then  
    Stop and Get Gas  
    Drive to Work  
or else if the car is not low  
on gas then  
    Drive to Work
```

This particular code above could be simplified. This simplification can be seen in this diagram:



This if-then-else structure allows the driver to analyze the situation and branch accordingly. This same concept holds true in a Python program.

General form (IF statement)

```
if condition:  
    statement  
    statement  
    etc.
```

The **if** statement is called a decision structure. It is a special case of a control structure that allows certain statements to be performed only if a certain condition is true. The condition is a Boolean expression and is either “true” or “false”.

Coding Tip: Sometimes it is better to set the initial value (in this case, a bonus was set equal to zero) and then let your control structure modify the value if the condition is met.

Example: Calculation Program

```
bonus = 0  
sales = 80000  
if sales > 70000:  
    bonus = sales * 0.10  
print('The bonus was ', bonus)
```

If the total sales achieved by a salesperson exceeds \$70,000, this person will receive a bonus of 10% of their sales, or else this person will not receive a bonus.

Boolean Expressions

Normally one of the following relational operators is used in a Boolean expression:

Relational Operator	Meaning
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equals
!=	Not Equals

Boolean expression form

Item 1 {relational operator} Item 2

Coding Tip: Consider if a company wishes to give a 20% bonus for a salesperson who makes more than \$100,000 and a 10% bonus for a salesperson who makes more than \$70,000. Assume sales are integers.

Item 1 and Item 2 are typically either variables or numbers. Remember that the concept of the Boolean expression is that it is either true or false, which means that at least one of the items has to be a variable. It would not make

any sense for both to be a number, as this would result in it being *always* true or *always* false.

Example:

Revised Bonus Calculation Program

```
jan_sales = int(input("Please enter january sales"))
feb_sales = int(input("Please enter february sales"))
mar_sales = int(input("Please enter march sales"))
tot_sales = jan_sales + feb_sales + mar_sales
bonus = 0
if tot_sales > 100000:
    bonus = tot_sales * 0.20
if tot_sales > 70000:
    bonus = tot_sales * 0.10
print('bonus was ', bonus)
```

Pause for a few minutes and consider what is wrong with this code.

If the `tot_sales` is greater than \$100,000, then it performs the first if statement BUT also performs the second if statement, which re-calculates the bonus at the 10% level. Now we could re-arrange the if statements such so that the code could work but there is an easier method.

The first form is the if-else statement:

General form (IF-ELSE statement)

```
if condition:
```

```
    statement
```

```
    statement
```

```
    etc.
```

```
else:
```

```
    statement
```

```
    statement
```

```
    etc.
```

Python checks the first condition and, if true, will execute the indented code underneath it. If false, it will skip to, and execute, the code contained underneath **else**.

This decision structure works well in many circumstances, such as the following coding example in which a company wishes to pay time and a half for any hours over 40 hours. Assume that hours and pay rate are integers.

Example:

```
hours = int(input("Please input number of hours worked"))
payrate = int(input("Please input pay rate"))
if hours <= 40:
    salary = hours * payrate
else:
    salary = 40 * payrate + (hours - 40) * payrate * 1.5
print('salary is equal ', salary)
```

While this structure is useful in many cases, Python provides an extension that will help in the coding issues presented above:

General form (IF-ELIF-ELSE statement)

```
if condition_1:
```

```
    statement
```

```
    statement
```

```
    etc.
```

```
elif condition_2:
```

```
    statement
```

```
    statement
```

```
    etc.
```

```
else:
```

```
    statement
```

```
    statement
```

```
    etc.
```

Note: This form only has one elif but Python allows as many as is necessary for your code! Remember that only one block of code will be executed here, that is, if **condition_1** is true, only that chunk of code will be executed and the rest will be skipped. The elif is equivalent to an “else if” in other programming languages.

Now let's revisit our code above. Consider if a company wishes to give a 20% bonus for a salesperson who makes more than \$100,000 and a 10% bonus for a salesperson who makes more than \$70,000. Assume sales are integers.

Example:

Fixed Revised Bonus Calculation Program

```
jan_sales = int(input("Please enter january sales"))
feb_sales = int(input("Please enter february sales"))
mar_sales = int(input("Please enter march sales"))
tot_sales = jan_sales + feb_sales + mar_sales
if tot_sales > 100000:
    bonus = tot_sales * 0.20
elif tot_sales > 70000:
    bonus = tot_sales * 0.10
else:
    bonus = 0
print('bonus was ', bonus)
```

The statements contained in an if, if-else, or an if-elif-else statement can contain decision structures within themselves if necessary. For example, the design might be:

```
if condition_1:  
    if condition_1a:  
        statement  
        statement  
        etc.  
    else:  
        statement  
        statement  
        etc.  
else:  
    statement  
    statement  
    etc.
```

Coding Tip: If possible, keep computer programs simple. Embedded decision structures can be beneficial but one indentation issue can cause code to run incorrectly. Remember, the spacing determines which part of the decision structure that the statement belongs under.

Logical Operators

Often in programming, the basic idea is to **keep it simple**, which works out well since that is also one of Python's main features! Logical operators allow us to reduce the need for complicated nested decision structures.

Operator	Meaning
and	The Boolean expressions on both sides of the “and” must be true for the entire expression to be true.
or	One or both of the Boolean expressions on both sides of the “or” must be true for the entire expression to be true.
not	This is a unary operator. It reverses the truth of the expression that follows it.

Consider if a company wishes to give a 20% bonus for a salesperson who makes more than \$50,000 and less than \$100,000. Assume that sales are integers.

Example:

```
jan_sales = int(input("Please enter january sales"))
feb_sales = int(input("Please enter february sales"))
mar_sales = int(input("Please enter march sales"))
tot_sales = jan_sales + feb_sales + mar_sales
if tot_sales > 50000 and tot_sales < 100000:
    bonus = tot_sales * 0.20
else:
    bonus = 0
print('bonus was ', bonus)
```

This code could be accomplished with a nested structure but again the idea is to keep it simple!

Example:

```
num = int(input("Please input a number: "))
if num != 0 and 9 / num == 3:
    print('we found the answer')
```

Note: Short-Circuit evaluation does exist in Python but is not typically necessary for financial programming.
Although the following code is nonsensical, it does illustrate the idea:

If num is equal to zero then we would not want 9 / num to occur. It would cause the incomputable division by zero. The “and” structure will check num != 0 first which, if false, will not even check the second Boolean expression, since BOTH have to be true for the “**and**” operator. This is short-circuit programming. Usually, there are much simpler ways to program the above code (with more lines of code) and short-circuit evaluation should be avoided.

Boolean Variables

The Boolean expression does not necessarily have to contain a relational operator; it can simply contain a Boolean variable.

Example:

```
bonus = True
if bonus:
    print("congratulations you received a bonus!")
```

Sometimes using Boolean variables can be confusing to another programmer. It makes perfect sense when you are writing the code but may not to another individual. Sometimes it is best to use the following practice:

Example:

```
bonus = True
if bonus == True:
    print("congratulations you received a bonus!")
```

Notes: Strings and String Functions

Introduction

Strings are simply sequences of characters. This sequence is called different names in different programming languages. The sequence of characters in the string "happy" can be seen by this example:

```
for ch in "happy":  
    print (ch)
```

Will give the result:

```
h  
a  
p  
p  
y
```

The code iterates through each character of the string, storing the character in a variable called **ch**. Many programming languages require a more complex manner of traversing through a string character by character. This for loop would allow character by character manipulation/calculation. Consider this example that counts how many a's are in a string:

```
account = 0
for ch in "happy days are here again":
    if ch == 'a':
        account = account + 1
print("The number of a's was %d" % account)
```

Will give the result:

The number of a's was 5

Indexing of a String

As in other programming languages, the character index starts at 0 and goes up to the length of the string minus one. Consider the string “happy”:

'happy'
↑↑↑↑↑
01234

Even though the length of this string is 5, the indexing goes from 0 to 4. To reference a particular letter in this string, the indexing would follow this form: variable[index]. For example, to access the a in happy the program would need to access in the following manner:

```
| str1 = "happy"  
| print(str1[1])
```

The result would be: a

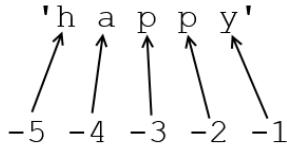
Using indexing, the same result can be achieved as the first example with the following code:

```
| str1 = "happy"  
| for x in range(0, 5):  
|     print(str1[x])
```

The for loop will be covered in more detail later on, but this loops through x, equaling the values from 0 to 4 (one less than the number that you see in the range function). The result would be:

```
h  
a  
p  
p  
y
```

Indexing can also occur from the end of the string in this manner:



The code to access the a from the end of the string would look like:

```
str1 = "happy"  
print(str1[-4])
```

Len Function

In some cases, the length of the string is not known to begin with, but can be determined with the len function. Consider this example to count how many a's are in the string input by the user:

```
aconut = 0  
str1 = input("Please enter the text: ")  
for x in range(0,len(str1)):  
    if str1[x] == "a":  
        aconut = aconut + 1  
print("The number of a's is %d" % aconut)
```

Notice that, in this problem, len(str1) dynamically determines the length of the string at runtime. The result is the following:

```
Please enter the following text: happy days are here again  
The number of a's is 5
```

Concatenation

String variables can be concatenated in the same manner that was presented with the print function. Consider this code:

```
str1 = ""  
str1 = str1 + "happy days are "  
str1 = str1 + "here again"  
print (str1)
```

The result is as follows: “happy days are here again”

String Slicing

The capability exists to slice part of the string out. This is similar to the mid or substr function in other programming languages. The general form is:

```
variable[begin:end]
```

Where begin is the starting index of the string to be extracted and end is the ending index. The string that will be extracted starts at the starting index and goes to one less than the ending index. Consider the following code:

```
datestr = input("Input date (mm/dd/yyyy)")  
month = int(datestr[0:2])  
day = int(datestr[3:5])  
year = int(datestr[6:10])  
print("The month was %d" % month)  
print("The day was %d" % day)  
print("The year was %d" % year)
```

The result would be:

```
Input date (mm/dd/yyyy)12/13/2014  
The month was 12  
The day was 13  
The year was 2014
```

Note that the datestr[0:2] pulls the substring that goes from index = 0 to index = 1 (one less than the ending index). This code has some serious issues with it.

Consider if the date that is entered would be 1/1/2014, what do you think would happen? We will talk about ways to handle this issue shortly.

Other variations of the slicing exist and are demonstrated with these examples:

Python Code	Result
<pre>str1 = "happy days are here again" print(str1[:6])</pre>	happy (Leaving the beginning index blank defaults the beginning

	index to 0, so this example extracts the string from index 0 to index 5.)
<pre>str1 = "happy days are here again" print(str1[11:])</pre>	are here again (Leaving the ending index blank defaults to the length of the string, so this example extracts the string from index 11 to the end of str1.)
<pre>str1 = "happy days are here again" print(str1[-5:])</pre>	again (Inputting a negative index in for the beginning and leaving the ending blank causes the last n characters to be extracted, so this example extracts the last 5 characters.)

“In” and “Not In” Operators

The **in** operator tests whether one string is *in* another string. For example, consider the following code:

```
str1 = "happy days are here again"
if "are" in str1:
    print("it contains the string 'are'")
else:
    print("it does not contain the string 'are'")
```

The result of this function is true or false so, oftentimes, it is used in an “if” statement! The result of this code is:

it contains the string ‘are’

Consider if the string is changed to Are (with an uppercase A):

```
str1 = "happy days are here again"
if "Are" in str1:
    print("it contains the string 'are'")
else:
    print("it does not contain the string 'are'")
```

The result of this code is:

it does not contain the string ‘are’

The not in operator works in a similar manner, but checks to see if the string does not contain the search string.

String Testing Methods

The following functions test various states of a given string:

String Testing Method	Result
isalnum()	True if contains either letters or numbers and is at least one character long.
isalpha()	True if contains only letters and is at least one character long.
isdigit()	True if contains only numbers and is at least one character long.
islower()	True if all letters are lowercase and contain at least one letter.
isupper()	True if all letters are uppercase and contain at least one letter.
isspace()	True if string only has whitespace characters and contains at least one whitespace character.

These are not functions but methods, so they follow the variable name as in this example:

```
str1 = "A12345"
if str1.isupper():
    print("it is upper")
else:
    print("it is not upper")
```

Where begin is the starting index of the string to be extracted and end is the ending index. The string that will be extracted starts at the starting index and goes to one less than the ending index. Consider the following code:

This code gives the following result: it is upper

String Modification Methods

String Modification Method	Result
lower()	Changes all letters to lowercase
upper()	Changes all letters to uppercase
lstrip()	Strips out all leading whitespace characters
rstrip()	Strips out all trailing whitespace characters
strip()	Strips out all leading and trailing whitespace characters
lstrip(char)	Strips out all copies of the given character at the beginning of the string
rstrip(char)	Strips out all copies of the given character at the end of the string
strip(char)	Strips out all copies of the given character at the beginning and end of the string

As before, these are methods and should follow the variable name. Many times in comparisons and searches through strings the programmer does not wish the wrong case to cause issues. If the data was going to be stored in a database, the free-form entry of data can cause database issues. For example, one of my first database applications asked for the state. I allowed it to be entered in freeform. For one state like Kansas, there were many forms entered in: KS, Ks, ks, kansas,

Kansas, etc. I realized my mistake early and corrected the issue. Uppercase versus lowercase will cause major headaches in databases! A simple method like upper() can change all the letters to uppercase.

String Search and Replace Methods

String Search and Replace Methods	Results
startswith(substring)	True if the string begins with the given substring
endswith(substring)	True if the string ends with the given substring
find(substring) find(substring,start_index) find(substring,start_index,end_index)	If the string contains the given substring, then it returns the beginning index of the first substring.
replace(old_substring,new_substring)	replaces all copies of the old_substring with the new_substring

Let's revisit the date entry program:

```
datestr = input("Input date (mm/dd/yyyy)")  
month = int(datestr[0:2])  
day = int(datestr[3:5])  
year = int(datestr[6:10])  
print("The month was %d" % month)  
print("The day was %d" % day)  
print("The year was %d" % year)
```

The code needs to be changed to prevent errors when a single digit is entered for the month or day. Here is the modified code:

```
datestr = input("Input date (mm/dd/yyyy)")  
position_firstrslash = datestr.find("/")  
position_secondslash = datestr.find("/", position_firstrslash + 1)  
  
month = int(datestr[0:position_firstrslash])  
day = int(datestr[position_firstrslash+1:position_secondslash])  
year = int(datestr[position_secondslash+1:])  
  
print("The month was %d" % month)  
print("The day was %d" % day)  
print("The year was %d" % year)
```

Notes: Basic Input/Output

The Input Function

The **input** function allows your Python program to receive input from the user of the program.

General format

```
variable = input(prompt)
```

The prompt is a string that is displayed to the user of the program prompting the

Example:

```
name = input("please input first name: ")
```

input. The input function returns a string value and attempts to place it into the variable. This example prompts the user for their first name, then places it into a variable called name.

The only type that is returned is a string type so an appropriate conversion must be done if another type is desired.

Common Type Conversions	Meaning
int(variable)	Converts the variable type to integer if possible
float(variable)	Converts the variable type to floating point if possible
str(variable)	Converts the variable to a string (not necessary for the input function since it is already a string)

The following example inputs two integers, multiplies them together, and prints the answer.

The int() surrounding the input function converts it from String to Integer. If an

Example:

```
num1 = int(input("input first number: "))
num2 = int(input("input second number: "))
prod = num1 * num2
print(prod)
```

invalid value is entered, an error will occur.

The Print Function

As seen in earlier sections, the print function allows your Python program to print various types of information to the screen.

Statement	Output
<code>print("Have a good day")</code>	Have a good day
<code>print('Have a good day')</code>	Have a good day
<code>print('Have a', 'good day')</code>	Have a good day
<code>print ('Have', 'a', 'good', 'day')</code>	Have a good day

Some items to note:

1. Either the double quotes or single quotes can surround the text message that is to be printed to the screen.
2. Different text can be separated by a comma within one print statement but it puts a space automatically between the text. Typically, in programming, we would expect the last print statement to return the result "Haveagoodday" but spaces are automatically put between text.

The reason why the print function allows either the double quotes or single quotes is because the programmer might actually want to have either one inside the text. For example, if the following string is to be printed: I'm home – then this could be accomplished with the following code:

Python Code	Result
<code>print("I'm home")</code>	I'm home

A single quote was necessary inside the string so we used double quotes to surround all the text.

Consider if we wanted to print: The author said “Good Job!” – then the following code would work:

Python Code	Result
<code>print('The author said "Good Job!"')</code>	The author said, “Good Job!”

The double quotes were necessary inside the string so we used single quotes to surround all the text. The Python programmer may have to use a combination of these items if the desire is to have the string contain both the double quotes and single quotes. Another technique is to use triple double quotes or triple single quotes, as in this example:

Python Code	Result
<pre>print("""The author's quote was "Good Job!" """)</pre>	The author's quote was “Good Job!”

The triple quotes allow the programmer to print items on multiple lines, as in this example:

Python Code	Result

```
print ("""
Four score and seven years ago our fathers
brought forth on this continent, a new nation,
conceived in Liberty, and dedicated to the
proposition that all men are created equal.
Now we are engaged in a great civil war, testing
whether that nation, or any nation so conceived
and so dedicated, can long endure. We are met on
a great battle-field of that war. We have come to
dedicate a portion of that field, as a final rest
place for those who here gave their lives that the
nation might live. It is altogether fitting and proper
that we should do this.
""")
```

Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battle-field of that war. We have come to dedicate a portion of that field, as a final resting place for those who here gave their lives that the nation might live. It is altogether fitting and proper that we should do this.

Obviously, this is beneficial if the programmer has many lines of text to print to the screen.

The print function can also print the value of a variable.

Python Code	Result
-------------	--------

<pre>num = 3 print(num)</pre>	3
<pre>num = 3.0 print(num)</pre>	3.0
<pre>positive_message = "Good job!" print(positive_message)</pre>	Good job!

These three examples show printing variables containing integers, floats, and strings

The print function can also print the value of an expression.

Python Code	Result
<pre>print(5 * 2 + 7)</pre>	19
<pre>print(1 / 3)</pre>	0.3333333333333333

With the basic print function, the formatting may not be the desired format (perhaps you wanted 0.33 as the result of the second statement).

The print function can also print a combination string of literals, variables, and expressions.

Python Code	Result
<pre>n1 = 1 n2 = 3 print(n1,"divided by",n2,"is equal to",n1/n2)</pre>	1 divided by 3 is equal to 0.3333333333 333333

Python's Escape Characters

As seen in earlier sections, the print function allows your Python program to print various types of information to the screen.

Escape Character	Effect
\n	Output advances to the next line
\t	Output advances to the next horizontal tab position
\'	Output has a single quote put into the current position
\"	Output has a double quote put into the current position

\\" data-bbox="172 99 199 120"/>

Output has a backslash put into
the current position

More escape characters exist and can be found at this URL:

https://docs.python.org/2/reference/lexical_analysis.html#string-literals The additional escape characters are for specialized purposes and not used in regular programming. For example, I used many of these escape characters when programming a Zebra printer to print serial tags containing barcodes for spectrum analyzers. Combinations of the above escape characters can be used, as shown in the following examples.

Example:

```
print("Good\n\n\nJob!")
```

The \n repeated three times will cause “job!” to be printed three lines down. The regular print function only goes down one line, as seen below.

Example:

```
print("Good")
print("Job!")
```

The \t is useful to print information in a semi-formatted state.

Example:

```
print("state","sales","sales","sales")
print("", "2008", "2009", "2010")
print("", "----", "----", "----")
print("kansas",2234,5555,7777)
print("washington",8822,1000,3000)
```

This gives the output:

```
state sales sales sales
2008 2009 2010
-----
kansas 2234 5555 7777
washington 8822 1000 3000
```

As can be seen, the formatting is poor. Consider the use of \t in the print functions:

Example:

```
print("state\t\t sales\t\t sales\t\t sales")
print("\t\t\t 2008\t\t 2009\t\t 2010")
print("\t\t\t ----\t\t ----\t\t ----")
print("kansas\t\t",2234,"\\t",5555,"\\t",7777)
print("washington\t\t",8822,"\\t",1000,"\\t",3000)
```

This gives the output:

state	sales	sales	sales
	2008	2009	2010
	----	----	----
kansas	2234	5555	7777
washington	8822	1000	3000

For a text-based output, the \t and proper use of spaces allow an acceptable display. Do not use spaces alone to control output. Different fonts will display differently!

String Concatenation in the Print Function

The plus sign (+) can be used in a print function to concatenate two strings. The purpose of this can be seen in the following examples:

Python Code	Result
<code>print("comp","uter")</code>	comp uter (remember the separator [,] puts a space between the two strings)
<code>print("comp"+"uter")</code>	computer (the use of the + eliminates that extra space if it is not desired in the output)

Variables can be included in this concatenation as long as their type is string.

Example:

```
num=3  
print("happy" + num + "days")
```

gives the following syntax error: TypeError: Can't convert 'int' object to str

Example:

```
num=3  
print("happy" + str(num) + "days")
```

implicitly

The data type of num would need to be converted within the print function, as in this code:

The str(num) returns the value of num as a string value that can then be concatenated with “happy” and “days.” If the type of num is already a string then there is no need to use the str function:

Example:

```
num="3"  
print("happy" + num + "days")
```

gives the following result:

Example:

```
num1 = input("Input first number:")  
num2 = input("Input second number:")  
print(num1,"added to",num2,"gives the answer",num1+num2)
```

Coding Tip: Improper use of the types of the variable is a common mistake for beginning programmers.

```
Input first number:3  
Input second number:4  
3 added to 4 gives the answer 34
```

Obviously, this is not the desired result. The problem in this Python program is that the numbers were not changed from strings to integers. Consider this change:

Example:

```
num1 = int(input("Input first number:"))  
num2 = int(input("Input second number:"))  
print(num1,"added to",num2,"gives the answer",num1+num2)
```

The int function converts the input from the user from a string type to an integer type and now gives the desired result:

```
Input first number:3  
Input second number:4  
3 added to 4 gives the answer 7
```

Keep in mind that the lack of syntax errors does not mean you do not have other errors in your program!

String Formatting

Any programming language should have more functionality than what was presented above. Python is no different. It includes two components: **format**

Example:

```
num1 = 1 / 3
print("The answer is %.2f" % num1)
```

Format Specifier

String Format Operator

Variable to be Placed in the Location of the Format Specifier

specifier and **string format operator**. The format specifier is embedded into the string portion of the print function. The string format operator connects the number to be formatted (although it does not have to be just a number, as we will see in the examples).

In this example, the value of the variable num1 will be placed in the location of the format specifier using the format indicated.

Formatting Floating Point Values

The %f is the format specifier for a floating point number. The .2 will round the number to two decimal places. In order to round the number to three decimal places, %.3f would be used. The decimal point in the format specifier specifies the “actual” decimal point.

The above code would give the following result:

The answer is 0.33

Consider the following examples to better understand the formatting for floating point values.

Python Code	Result
<pre>num1 = 4 print("The answer is %.2f" % num1)</pre>	<p>The answer is 4.00 (Note: the data type of num1 is an integer, but the formatting handled the conversion automatically. Even though there were no actual decimal places, it still put zeros for the two specified decimal)</p>

	places)
<pre>num1 = 4 print("The answer is '%8.2f'" % num1)</pre>	The answer is ' 4.00' (Although the font does not show it, there are actually eight placeholders within the single quotes, of which two are decimals and one is the decimal point). Extra spaces are placed before the number to ensure the total length is the number before the decimal (in this code it would be eight). This size allows decimal points to be lined up in output from a Python program.

Formatting Integer Values

The %d is the format specifier for an integer number. The d represents the decimal integer. This does not mean that it has decimals since an integer does not. The %d does not have as many options as the floating point specifier.

Python Code	Result
<pre>num1 = 4 print("The answer is %d" % num1)</pre>	The answer is 4
<pre>num1 = 4 print("The answer is %8d" % num1)</pre>	The answer is 4 (The 8 in this example makes sure that the width is exactly eight, so it pads seven spaces in front of the 4).

Formatting String Values

The %s is the format specifier for strings. The s represents string. Consider these examples:

Python Code	Result
<pre>state_name = "new york" print("The name was %s" % state_name)</pre>	The name was new York
<pre>state_name = "new york" print("The name was %10s" % state_name)</pre>	The name was new York (The 10 before the s serves the same purpose as in the %d example; it makes sure that it is 10 characters in length.)

Combination of Format Specifiers

A print function can have more than one specifier.

Example:

```
num1 = 41.345
num2 = 8.5917
print("%.2f divided by %.2f equals %.4f" % (num1,num2,num1/num2))
```

which gives the result: 41.34 divided by 8.59 equals 4.8122

Notice in this example that there are three format specifiers and, in the parenthesis, there are three values (num1,num2,num1/num2).

Notice the num1/num2 is an expression. The values following the string format operator can be variable(s), number(s), or expression(s). If there is more than one value, then it must have parenthesis around it, as in the above example.

Notes: Looping

Looping Structure

Many applications in programming would not be possible or even feasible without some form of looping structure. Without looping capability, every computer program would just have a single pass capability. Some mathematical formulas would be extremely difficult to program.

Consider the following example:

```
print("factorial program")
print("-----")
num = int(input("n!, please input n: "))
fact = 1
if num == 0:
    fact = 1
elif num == 1:
    fact = 1
elif num == 2:
    fact = 1 * 2
elif num == 3:
    fact = 1 * 2 * 3
elif num == 4:
    fact = 1 * 2 * 3 * 4
elif num == 5:
    fact = 1 * 2 * 3 * 4 * 5
elif num == 6:
    fact = 1 * 2 * 3 * 4 * 5 * 6
elif num == 7:
    fact = 1 * 2 * 3 * 4 * 5 * 6 * 7
# and so on
```

Imagine if we wanted our program to be capable of calculating up to 1,000 factorial, we would then need to add 1,986 more lines of code, including

multiplication of each and every number beforehand. It would work but what if the user wanted to calculate 1,001 factorial? Clearly, this would be a tedious task.

Consider another example of a program that prints a shipping label for a single customer. It might work perfectly and give the result. But then imagine if your company is as large as Amazon which, according to *Business Insider*, sold 27 million items on Cyber Monday! Your shipping program would need to be run 27 million times to handle just that one day.

Looping allows multiple iterations of a specified chunk of code. They can be designed to run a set number of times, or make use of variables, which may alter the number of iterations. For example, imagine a program that takes your name as an input (e.g. Jane), and then prints out each letter. The loop could run four times and always produce J-a-n-e. To have it work for names more or less than four characters, you'd probably want a variable to store the length of the input name, and have it loop for the value of that variable.

Background

In the past, much of the looping done was with what are known as **goto** statements. Going back to factorials, an example program to calculate 5 factorial might look like:

```
fact = 1
n = 1
label: start
    fact = fact * n
    n = n + 1
    if n <= 5:
        goto start
    print("fact = ", fact)
```

While this initial “looping” helped solve the problem of “single pass” processing, it led to many unstructured programs. Allowing programs to jump anywhere in the code, while useful in some regards, leads to very messy programs as they get larger. Bugs become hard to identify because there is no clear way of telling where the execution will end up next. My first programming job on a Unisys mainframe used this type of looping. The programming language was COBOL and, while it had rudimentary looping constructs, the standards of the company required the programmers to use labels and gotos. As you can imagine, it was not enjoyable.

As programming languages evolved, the concept of “structured programming” became very important. Many textbooks adopted the term “structured programming” in their title. These sorts of programs have a precise structure and make it much easier to see what code is going to be executed next.

The Primary Loop Statements

Two main constructs exist in looping: a **for** loop, and a **while** loop. Their English-language counterparts give hints as to how they operate. Usually, the statement's construct tells:

1. the starting value
2. the increment value, and
3. how long should the looping continue.

A for loop allows looping but in a very confined manner. Consider the following code to calculate 5 factorial:

```
fact = 1
for num in range(1,6):
    fact = fact * num
print("5! = ", fact)
```

The program iterates through the range of one and six, then multiplies the number by the previous result (stored in the **fact** variable). The for loop iterates through a set collection of values (in this case, the range of numbers beginning at one and less than 6).

The second construct is a **while** loop. It iterates a chunk of code *while* a condition is true. As soon as the condition becomes false, the indented code is not executed, and the program continues execution with the remaining code.

Consider the same code to calculate 5 factorial:

```
fact = 1
num = 1
while num < 6:
    fact = fact * num
    num = num + 1
print("5! = ", fact)
```

Many programmers prefer the while construct since it allows complete control; that is, the three pieces of information listed above are clearly defined. There are some downsides to the while construct. It is easier to program an infinite loop where the program continues running forever or until the program halts with an error. Consider the above code if the increment was left off for num:

```
fact = 1
num = 1
while num < 6:
    fact = fact * num
print("5! = ", fact)
```

This program runs for an infinite amount of time, as the value of **num** will always be less than six!

This example illustrates how complete control can be a bit trickier, which can be a benefit or a hindrance in certain cases, in that there are more variables or conditions to manage.

There are many variations of these two types of constructs in different programming languages, including do...while, and repeat...until.

Notes: Brute Force Processing

Overview

In the early days of computing, RAM and hard-drive memory were scarce. As late as 1995, I remember struggling to fit network and system drivers into 640 kilobytes of memory. I would have to rearrange the order of the drivers to get them to work. These were the DOS days when programming had to be done much more carefully. Variables were re-used to save space and unused variables were unheard of.

As time has passed, programmers have seen computers dramatically increase in both RAM and hard-drive space. A typical personal computer now has at least 4GB of RAM (a 6,250-fold increase since my DOS days!). One mainframe process that I evaluated took five hours to process all the database records involved. It had millions of records that it had to sequentially process. As disk space became more inexpensive, I was able to add an additional index that took the processing down to two minutes.

Brute force processing is the sequential iteration through a collection. Consider a sequential search through a database. Looking up an address by an input name in a personnel directory, for example, would start at the first entry and compare the input with the stored value, then continue entry-by-entry, until it finds a match. In the past, brute force processing was costly. Sometimes it had to be done, but its execution was very noticeable. The mainframe would slow down during the time

they ran that five-hour process. It often had to be run at night so that users of the system would not notice the impact.

However, as a result of the increase of processing power, the programming involved in brute force processing is now an accepted practice. In many cases, it is preferred. Consider a case in which an algorithm is programmed inefficiently and this results in a two-second increase in the delay. Would anybody really notice, or care, about an extra two seconds of execution? Obviously, this depends on the context of the program but, in many cases, it would not matter. Especially if the more efficient version requires hours and hours of extra coding effort.

When programming for a business, coding efficiency is often less of a priority than how quickly results can be produced. Programmers tend to want a certain level of “purity” in their programs. I have found myself wanting to redo logic after I got it working, just because I knew there was redundancy in the program. Sometimes, it takes the business to stop the programmer.

Many formulas exist to analyze efficiency. Two seconds does not sound like much if it is applied to the total processing time, but consider if it was 2 seconds **per customer record**. If we had a business that had 10,000,000 customer records, then we are talking about 20,000,000 seconds or approximately 231 days. All of a sudden, that “two seconds” is very significant!

Python Example

Consider the problem of simplifying a fraction by hand in an introductory algebra course:

$$\begin{aligned}\frac{20}{24} &= \frac{2 \cdot 2 \cdot 5}{2 \cdot 2 \cdot 2 \cdot 3} \\ &= \frac{5}{2 \cdot 3} \\ &= \frac{5}{6}\end{aligned}$$

The correct way is to do the prime factorization of each number and then cancel what the numerator and denominator have in common.

With Python, we can use brute force processing to start at 20 (the numerator) and work our way backward to ascertain if both the numerator and denominator are both divisible by the same number:

Are 20 and 24 divisible by 20?

Are 20 and 24 divisible by 19?

Are 20 and 24 divisible by 18?

Are 20 and 24 divisible by 17?

...and so on.

```
numer = 20
denom = 24
for num in range(20,1,-1):
    if numer % num == 0 and denom % num == 0:
        numer = numer / num
        denom = denom / num
print("20/24 = %d/%d" % (numer,denom))
```

This program actually goes down to division by two, which is unnecessary, as four is the least common factor and the program could stop there. But accounting for that would require the programming to be more complex, and the time it saves would not even be noticeable.

Notes: While Loops

Overview – The While Loop

The while loop is a condition-controlled loop. It will continue as long as a certain condition is met.

General Form:

```
while condition:  
    statement  
    statement  
    etc.
```

The condition is checked prior to performing the statements. Once the condition is no longer true, the program execution continues past the last statement in the while loop. The condition is a Boolean expression. Consider this example that adds the numbers from 1 to 10:

```
sum = 0  
num = 1  
while num <= 10:  
    sum = sum + num  
    num = num + 1  
print("The sum =",sum)
```

Often, while loops have a preceding line to initialize the variable that is being checked. In this case, **num** is set equal to 1. Next, the while statement will check

the Boolean condition. Somewhere within the while loop, the variable will have some kind of math operation performed on it.

```
sum = 0           initializes the variable
num = 1          ←
while num <= 10: ← checks the condition of the variable
    sum = sum + num
    num = num + 1 ← performs a math operation on the
print("The sum =",sum)   variable
```

These three elements are crucial to the successful implementation of the while loop. All three of these components will take various forms but must be in the program. Consider the example of finding n-factorial:

```
print("Factorial Program")
n = int(input("Please input n: "))
num = 1
fact = 1
while num <= n:
    fact = fact * num
    num = num + 1
print("%d! = %d" % (n,fact))
```

The three elements are presented here:

```
print("Factorial Program")
n = int(input("Please input n: "))
num = 1 ← Initializes the variable
fact = 1
while num <= n: ← Checks the condition of the variable
    fact = fact * num
    num = num + 1 ← Performs a math operation on the
                     variable
print("%d! = %d" % (n,fact))
```

Both of these above examples added one, but any math calculation could be done. Perhaps you're evaluating a large collection of inputs, three values at a time. In that case, you would likely want to increment **num** by 3, rather than 1.

Consider if the variable is not initialized in the above example:

```
7 print("Factorial Program")
8 n = int(input("Please input n: "))
9
10 fact = 1
Errors → 11 while num <= n:
          ↑
          ↑
          ↑
12     fact = fact * num
13     num = num + 1
14 print("%d! = %d" % (n,fact))
```

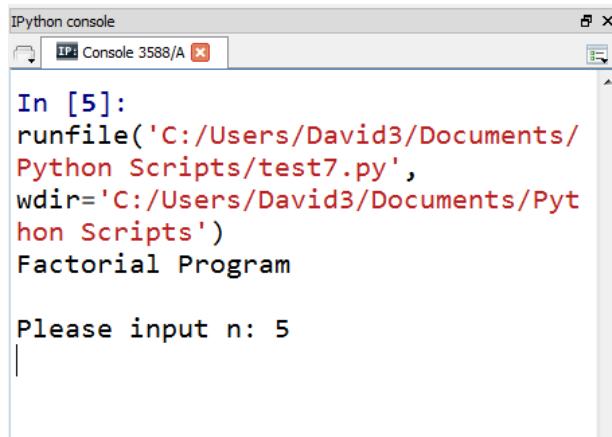
The Anaconda system helps points out issues and recognizes that the initialization was not done.

Consider if the math operation is not performed on the variable:

```
print("Factorial Program")
n = int(input("Please input n: "))

fact = 1
num = 1
while num <= n:
    fact = fact * num
print("%d! = %d" % (n,fact))
```

Notice that there are no errors pointed out by the system. Once the program is run, however, the following occurs:



The screenshot shows an IPython console window titled 'IPython console' with a tab labeled 'IPython Console 3588/A'. The code in the console is as follows:

```
In [5]:  
runfile('C:/Users/David3/Documents/  
Python Scripts/test7.py',  
wdir='C:/Users/David3/Documents/Pyt  
hon Scripts')  
Factorial Program  
  
Please input n: 5
```

The program never comes back since num is never incremented. The loop continues indefinitely. Remember, when programming a while loop, to have all three elements and that the condition will eventually be false!

Notes: For Loops

For Loops

The **for loop** is a count-controlled structure. This means that it will be run *for* a set number of iterations. This total number of iterations might be hard-coded or determined at runtime. Consider this example, if we want to program the binomial probability formula:

$$P(x) = \frac{n!}{(n-x)!x!} \cdot p^x (1-p)^{n-x}$$

The $n!$ would need a for loop since it is count controlled. For example $5!$ would need to be iterated five times, as in:

5x4x3x2x1

General Form

```
for variable in [value1, value2, value3, etc.]:  
    statement  
    statement  
    etc.
```

This general example is written in pseudocode, which is an informal way of describing the procedure and/or structure of a program. It is useful for describing a program or algorithm without a specific language.

Note: Sometimes it is better to introduce some inefficiency in your code to improve readability. Any individual that has taken an introductory statistics course knows that a factorial is found by multiplying the numbers from 1 to the given number. The line ‘for num in [1, 2, 3, 4, 5]’ could have been written ‘for num in [2, 3, 4, 5]’ and would have achieved the same result. Try both and see the result. Reflect why this is the case. The code was written in the form above to match the definition of factorial.

Example Finding 5!

```
fact = 1
for num in [1, 2, 3, 4, 5]:
    fact = fact * num
print('5! = ', fact)
```

Here's an example written in Python, which can be run directly from an interpreter:

The values in the general form do not have to be strictly numbered; perhaps you'd like several different strings to be printed in succession:

Example:

Printing colors with a for loop

```
for color in ['red', 'white', 'blue']:  
    print (color)
```

It would be tedious to add all the numbers from 0 to 20 by applying the general form. Our for loop code would look like:

```
fornum in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,  
19, 20]:
```

But what if you had to iterate over 100 numbers? Or 100,000? Python provides the range function to simplify coding.

General form using the **range** function with an **end number** variable:

```
for variable in range(end number):  
    statement  
    statement  
    etc.
```

This will iterate the variable from 0 to end number – 1.

Note: The usage of 21 in the range function to have the values go from

0 to 20

Example:

Adding all the numbers from 0 to 20

```
sum = 0  
for num in range(21):  
    sum = sum + num  
print('sum = ', sum)
```

General Form using the **range** function from **start number** to **end number**:

```
for variable in range(start number, end number):  
    statement  
    statement  
    etc.
```

This will iterate the variable from start number to end number – 1.

Example:

Adding all the numbers from 10 to 12

```
sum = 0
```

```
for num in range(10, 13):
```

```
    sum = sum + num
```

```
print('sum = ', sum)
```

Example Code:

Adding all the numbers in a range input by the user

```
sum = 0  
  
print('This program sums all the numbers in the range  
specified')  
  
start_string = input('Input starting number:')  
start_num = int(start_string)  
  
end_string = input('Input ending number:')  
end_num = int(end_string)  
  
for num in range(start_num,end_num + 1):  
    sum = sum + num  
  
print('sum = ', sum)
```

Note: The above example was simple. In coding, sometimes it is easier to code a simple version first. For example, if the desired result was the sum of the numbers from 5 to 10,000, it would be very difficult to know the answer prior to writing the program. It may be a better idea to write the program to add the numbers from 10 to 12, which is much simpler to check. Once it is known that the test case works, then the code can be changed to ‘for num in range (5,10001)’ and the result can be trusted. This is not always the case since the sum may exceed the limits of the data type.

General Form using **range** from **start number** to **end number**, with a **step number**:

```
for variable in range(start number, end number, step number):  
    statement  
    statement  
    etc.
```

Note: The start number in the general form does not have to be the smaller number, so can be the larger number. Some

ExampleCode: algorithms require the for loop to decrement the variable in Adding all the even numbers from 4 to 10 each iteration.

```
sum = 0  
for num in range(4, 11, 2):  
    print (num)  
    sum = sum + num  
print('sum = ', sum)
```

This will iterate the variable from start number to end number – 1, changing the value of the variable by the step number for each iteration.

Oftentimes, the exact same manner a process is done by hand is the correct way to implement in a Python program. If the problem was to simplify the fraction 12/20, the appropriate step would be to think of the largest number that divides evenly into both the numerator and denominator.

Example:

Simplifying a Fraction

```
numerator = 12
```

```
denominator = 20
```

```
for num in range(denominator,2,-1):
```

```
    if denominator % num == 0 and numerator % num == 0:
```

```
        denominator = denominator / num
```

```
        numerator = numerator / num
```

```
        print(int(numerator),'/',int(denominator))
```

This code starts with the largest number and goes down to 2, checking to see if the numerator and denominator are divisible by the value of num; if so, then it divides both the numerator and denominator by that number. The answer that this code gives is 3/4. On the other hand, if the for loop is written as “for num in range (2, denominator):” then the result is 6/10. This program would be very difficult, if not impossible if the start number is less than the end number in the general form.

The List datatype has not been covered yet, but when combined with for loops provide powerful mechanisms. Many programming languages refer to lists as arrays. Consider a list of numbers: 10, 12, 17, and 15. These can be put on a Python list using this assignment: `pylist = [10, 12, 17, 15]`. Recall that arrays are indexed starting at zero and this is true with lists in Python also.

Example Code:

Adding Numbers to a List

```
tsv = [10, 12, 17, 15]
sum = 0
for num in range(0,4):
    sum = sum + tsv[num]
print('sum = ', sum)
```

Index	Value
0	10
1	12
2	17
3	15

Note: The indexing of the numbers in the list goes from 0 to 3:

In future topics, the **for loop** will be invaluable in simplifying the code.

Notes: Using Functions

Overview

As you already know from loops, sometimes code needs to be run more than once. But what if the loops themselves need to be executed more than once? For example, consider the case in which our program should calculate two factorials, as in the permutation formula from statistics.

$${}_n P_r = \frac{n!}{(n - r)!}$$

Using only loops, the program to accomplish this formula would be:

```
print("Permutation Program")
n = int(input("Please input n: "))
r = int(input("Please input r: "))

numerator = 1
for num in range(1, n + 1):
    numerator = numerator * num

denominator = 1
for num in range(1, (n - r) + 1):
    denominator = denominator * num

ans = numerator / denominator
print("answer = ", ans)
```

As can be seen, the same factorial logic is being done twice, but with different parameters. The code above is fairly short but consider if this same code has to be

repeated 20 times in a program. All of a sudden it is tedious to keep programming the same code over and over. Consider this same program done with a function:

```
def fact(param):
    ans = 1
    for num in range(1,param + 1):
        ans = ans * num
    return ans

print("Permutation Program")
n = int(input("Please input n: "))
r = int(input("Please input r: "))

numerator = fact(n)
denominator = fact(n-r)

ans = numerator / denominator
print("answer = ", ans)
```

In the above example, the factorial code is contained within a function, **fact()**.

Rather than having to write several loops, the **fact()** function can be *called* instead. Let's examine the various components:

The diagram illustrates the components of the provided Python code. Red arrows point from labels to specific parts of the code:

- A red arrow labeled "Defines Function" points to the `def` keyword.
- A red arrow labeled "Name of Function" points to the identifier `fact`.
- A red arrow labeled "Parameters Pass into Function" points to the parameter `param`.
- A red arrow labeled "Function Called in Different Places in Program" points to two occurrences of the `fact` function call: `numerator = fact(n)` and `denominator = fact(n-r)`.

```
def fact(param):
    ans = 1
    for num in range(1,param + 1):
        ans = ans * num
    return ans

print("Permutation Program")
n = int(input("Please input n: "))
r = int(input("Please input r: "))

numerator = fact(n)
denominator = fact(n-r)

ans = numerator / denominator
print("answer = ", ans)
```

The name of the function can be whatever name you choose, but follow the same naming conventions as variables. In this example, there is only one parameter passed into the function, but you can have any number of parameters, or even none.

The **return** statement returns a value to the location where the function is called, assuming it is a function that returns a value. Consider this example, which calculates the two answers for the quadratic formula:

```
import math

def quadratic_formula(pa, pb, pc):
    ans1 = (-1 * pb + math.sqrt(pb**2 - 4 * pa * pc)) / (2 * pa)
    ans2 = (-1 * pb - math.sqrt(pb**2 - 4 * pa * pc)) / (2 * pa)
    print ("Answers are %.2f and %.2f" % (ans1, ans2))

print("Quadratic Formula Program")
a = int(input("Please input a: "))
b = int(input("Please input b: "))
c = int(input("Please input c: "))

quadratic_formula(a, b, c)
```

Since this function has no return statement, the line that calls the function has no variable assignment. Notice that the variables in the main part of the program are a, b, and c but the parameters were called pa, pb, and pc. The parameters can be called by any name. It is the *values* of the variables in the main body of the code (a, b, c) that gets passed to the parameter variables defined in the function (pa, pb, pc). The concept behind this is referred to as variable scope, which will be covered more in depth later.

Consider this example:

```
import math

def quadratic_formula(a, b, c):
    ans1 = (-1 * b + math.sqrt(b**2 - 4 * a * c)) / (2 * a)
    ans2 = (-1 * b - math.sqrt(b**2 - 4 * a * c)) / (2 * a)
    print ("Answers are %.2f and %.2f" % (ans1, ans2))

print("Quadratic Formula Program")
a = int(input("Please input a: "))
b = int(input("Please input b: "))
c = int(input("Please input c: "))

quadratic_formula(a, b, c)
```

This code uses the same variable names as the main body of the code. This is often done but can lead to confusion. It is best to use different variable names in the function, even though it is possible to use the same variable names.

Python does not limit the programmer to returning one value, it can return many values. Here is the quadratic formula program redesigned to return the answers to the main program and then print them:

```
import math
def quadratic_formula(a, b, c):
    ans1 = (-1 * b + math.sqrt(b**2 - 4 * a * c)) / (2 * a)
    ans2 = (-1 * b - math.sqrt(b**2 - 4 * a * c)) / (2 * a)
    return ans1, ans2

print("Quadratic Formula Program")
a = int(input("Please enter a: "))
b = int(input("Please enter b: "))
c = int(input("Please enter c: "))
ans1, ans2 = quadratic_formula(a, b, c)
print("The answers are %.2f and %.2f" % (ans1, ans2))
```

Python has its own set of built-in functions that can be called. The program up above actually called one of these functions, **math.sqrt()**. There is even a factorial function that would have simplified our code in the while loop. It is important to become familiar with the functions built into the libraries of Python. There is no need to re-invent the wheel if somebody else has already written that function!

Anaconda has a nice feature for libraries that are imported. Consider this example:

```
import math

math.
```



The screenshot shows a code editor with the following code:

```
import math

math.
```

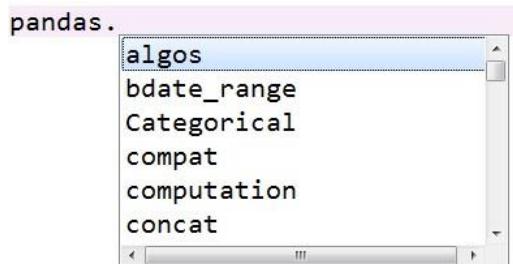
A dropdown menu is open at the end of the word "math.", displaying a list of mathematical functions:

- acos
- acosh
- asin
- asinh
- atan
- atan2
- atanh

Once the import is placed into the program code, you gain access to all of that library's functions. Calling the library name, followed by a dot, will display all of the available functions.

Consider the **pandas** library:

```
import pandas
```



While this list of functions is useful, the programmer will probably have to do some additional research on the usage and meaning of each function.

Notes: Scope of Variables

Overview

Typically, variables can be accessed in the piece of code where they are used. The question becomes whether variables defined in the main program could be used in a function and vice versa.

Local Variables

A variable declared in a function is called a local variable. This means it can only be accessed by that function. Consider this example:

```
7 def test():
8     num = 3
9     print("number =", num)
10
11 # this is the main program
12 test()
13 print("number =", num)
```

Notice that the variable **num** is defined in the function, but the program is trying to print it in the main code body. We have our indicator on the left saying there is a problem. Let's examine what happens if we ignore this and try to run the program anyway:

```
In [7]:  
runfile('C:/Users/David3/Documents/Python  
Scripts/test7.py',  
      wdir='C:/Users/David3/Documents/Python  
Scripts')  
number = 3  
number = 5
```

The first print function returns the number (3) that we would expect. The print function in the main program returns a bizarre result. **Do not ignore the error indicators or you might get bad results!**

This is the reason that the **return** statement was implemented within functions. It allows communication between the function and the main program. Consider this rewrite of the program:

```
def test():  
    testnum = 3  
    print("number =", testnum)  
    return testnum  
  
# this is the main program  
num = test()  
print("number =", num)
```

Notice how the variable declared in the function test was renamed. It could have been left named num but this is not considered good programming practice; it is too easy to confuse the two variables. Now, when this program is run, the following is the result:

```
In [8]:  
runfile('C:/Users/David3/Documents/Python  
Scripts/test7.py',  
wdir='C:/Users/David3/Documents/Python  
Scripts')  
number = 3  
number = 3
```

Keep in mind that, once a function is completed, then the variable is no longer accessible even by the same function. Consider this example:

```
def test(p):  
    if p == 1:  
        testnum = 5  
    else:  
        testnum = testnum + 1  
    print("The number =", testnum)  
  
# this is the main program  
test(1)  
test(2)
```

You might expect that it will print 5 the first time the test function is called, and 6 the second time it is called, but here are the results:

```
wdir='C:/Users/David3/Documents/Python Scripts')
The number = 5
Traceback (most recent call last):

  File "<ipython-input-1-89937c5ea1bc>", line 1, in <module>
    runfile('C:/Users/David3/Documents/Python Scripts/test7.py',
wdir='C:/Users/David3/Documents/Python Scripts')

  File "C:\Users\David3\Anaconda3\lib\site-
packages\spyderlib\widgets\externalshell\sitecustomize.py", line 585, in runfile
    execfile(filename, namespace)

  File "C:\Users\David3\Anaconda3\lib\site-
packages\spyderlib\widgets\externalshell\sitecustomize.py", line 48, in execfile
    exec(compile(open(filename, 'rb').read(), filename, 'exec'), namespace)

  File "C:/Users/David3/Documents/Python Scripts/test7.py", line 16, in <module>
    test(2)

  File "C:/Users/David3/Documents/Python Scripts/test7.py", line 11, in test
    testnum = testnum + 1

UnboundLocalError: local variable 'testnum' referenced before assignment
```

The first number prints correct, but then we get an error on the second pass: “UnboundLocalError: local variable ‘testnum’ referenced before assignment”. Remember that a local variable is only accessible within while that function is running. After it finishes the first run of the function, then the variable **testnum** is no longer available! In the above code, it is only created when **p == 1** is true. Remember, proper communication between the function and main program can be done with returned values. Consider this rewrite of the above code:

```
def test(p, testnum):
    if p == 1:
        testnum = 5
    else:
        testnum = testnum + 1
    print("The number =", testnum)
    return testnum

# this is the main program
num = 0 #dummy value for the variable
num = test(1, num)
num = test(2, num)
```

Now that particular test function can be called as many times in the program and it will work (nonsensical as the program appears to be). Our result with this correct code is as follows:

```
runfile('C:/Users/David3/Documents/Python  
Scripts/test7.py',  
wdir='C:/Users/David3/Documents/Python  
Scripts')  
The number = 5  
The number = 6
```

Global Variables

Variables declared in the main program (global variables) can be accessed directly within a function, but many consider it bad programming practice. Consider this example of the quadratic formula:

```
import math
def quadratic_formula():
    ans1 = (-1 * b + math.sqrt(b**2 - 4 * a * c)) / (2 * a)
    ans2 = (-1 * b - math.sqrt(b**2 - 4 * a * c)) / (2 * a)
    print("The answers are %.2f and %.2f" % (ans1, ans2))

print("Quadratic Formula Program")
a = int(input("Please enter a: "))
b = int(input("Please enter b: "))
c = int(input("Please enter c: "))
quadratic_formula()
```

This program works but notice that the function is using the variables (a, b, and c) directly from the main program. This is a topic that will probably be debated forever but many programmers consider it bad practice to use global variables directly in this manner. An alternative way to program this is listed here:

```
import math
def quadratic_formula(a, b, c):
    ans1 = (-1 * b + math.sqrt(b**2 - 4 * a * c)) / (2 * a)
    ans2 = (-1 * b - math.sqrt(b**2 - 4 * a * c)) / (2 * a)
    print("The answers are %.2f and %.2f" % (ans1, ans2))

print("Quadratic Formula Program")
a = int(input("Please enter a: "))
b = int(input("Please enter b: "))
c = int(input("Please enter c: "))
quadratic_formula(a, b, c)
```

Even when using the same variable names for the global variables and for the parameters, there is no doubt where the variables are coming from. Using the global variables like in the first example:

```
def quadratic_formula():
    ans1 = (-1 * b + math.sqrt(b**2 - 4 * a * c)) / (2 * a)
    ans2 = (-1 * b - math.sqrt(b**2 - 4 * a * c)) / (2 * a)
    print("The answers are %.2f and %.2f" % (ans1, ans2))
```

A programmer coming in would look at this code and question where a, b, and c are coming from, but if they are passed in a parameter to the function, there is no doubt it has simplified coding.

The key point is to pass everything into the function that you will need for that function!

Notes: Comparison of Global Functions, Local Functions, Lambda Functions and Methods

Overview

This lecture will define the different function types and give an example of each type. Subsequent lecture notes will further define these.

Definitions

Global functions – Global functions can be called anywhere from within the Python program. These can return any number of values or none at all. The two main purposes of global functions are for code reuse and modularity. The code reuse comes from a Python program needing to call a piece of code over and over. The modularity comes from breaking a longer program into logical, shorter chunks of code.

Local functions – These can only be called within the function they are declared in. These are supporting functions for that function. They are not meant to be called from outside of that particular function.

Lambda functions – These are small functions that calculate a single expression and return the value. While it can replace a one-line global function, their main purpose is for a call handler such as that used in graphical programming.

Methods – Methods are functions specifically for objects. They are very specific to the functionality of that particular object and cannot be called without the object reference.

Global Function Example

This is an example of using a global function for the quadratic formula.

```
def quadratic_formula():
    ans1 = (-1 * b + math.sqrt(b**2 - 4 * a * c)) / (2 * a)
    ans2 = (-1 * b - math.sqrt(b**2 - 4 * a * c)) / (2 * a)
    print("The answers are %.2f and %.2f" % (ans1, ans2))
```

Local Function Example

Here is an example of using local functions to calculate interest compounded either n -times per year or continuously:

```
import math
def interest(P, r, t, ntype):
    def interest_continuous(): ← Local Functions
        return P * math.e ** (r * t)
    def interest_n_times_per_year(): ←
        cn = float(ntype)
        return P * (1 + r/cn)**(cn * t)
    if ntype == "C":
        return interest_continuous()
    else:
        return interest_n_times_per_year()

# main program
principle = float(input("Input principle: "))
rate = float(input("Input rate (decimal form): "))
time = float(input("Input time (years): "))
ntimes = input("Input number of compounding times (C for continuous): ")

print("Interest = %.2f" % (interest(principle,rate,time,ntimes)))
```

Remember, if **interest_continuous()** is called in the main program, there will be an error.

Lambda Function Example

This is an example of using Lambda functions to calculate interest, either compounded n -times per year or continuously.

```
interest_continuous = lambda P, r, t : P * math.e ** (r * t)
interest_ntimes = lambda P, r, t, n: P * (1 + r/n)**(n * t)

principle = float(input("Input principle: "))
rate = float(input("Input rate (decimal form): "))
time = float(input("Input time (years): "))
ntimes = input("Input number of compounding times (C for continuous): ")

if ntimes == "C":
    print("Interest = %.2f" % (interest_continuous(principle,rate,time)))
else:
    print("Interest = %.2f" % (interest_ntimes(principle,rate,time,float(ntimes))))
```

Methods Example

In Python, which is an object-oriented programming language, much of the data is stored as what are known as *objects*. By now you should be quite familiar with one type of object; the String. Here's an example using one of the built-in string methods:

```
txt = "1023.56      "
print("This program removes spaces at the end of the string")
txt = txt.rstrip(" ")
print("*" + txt + "*")
```

Method for a String

Later in the course, you will learn how to design your own objects for manipulation and data storage.

Notes: Global Functions

Overview

Global functions are functions that are defined so that they can be called from anywhere in the Python program. Consider this example:

```
import math

def interest(P, r, t):
    return P * math.e ** (r * t)

def test():
    A1 = interest(1000,.03,5)
    print("A1 =",A1)

# main program
test()
A2 = interest(2000,.05,10)
print("A2 =",A2)
```

Both the interest function and the test function are global functions. Notice that the interest function is called from the main program as well as from within the test function.

It is easy to identify a local function since it is called from within a global function, as in this example:

```
3 import math
4
5 def interest(P, r, t):
6     return P * math.e ** (r * t)
7
8     def test():
9         A1 = interest(1000,.03,5)
10        print("A1 =",A1)
11
12 # main program
13 test()
14 A2 = interest(2000,.05,10)
15 print("A2 =",A2)
```

Notice that simple spacing turned the test function into a local function, which cannot be called from the main problem (note the error indicator). When writing your Python programs be careful of the spacing, since it will make a huge difference in some cases!

Notes: Local Functions

Overview

Local functions can be contained within a global function. These are functions that you would not normally want the main program to be able to call. Consider this example:

```
def test(num1,num2,operation):
    def add(a, b):
        return a + b
    def subtract(a, b):
        return a - b
    if operation == "+":
        return add(num1,num2)
    elif operation == "-":
        return subtract(num1,num2)

n1 = int(input("Input first integer: "))
n2 = int(input("Input second integer: "))
print("The sum was",test(n1,n2,"+"))
print("The diff was",test(n1,n2,"-"))
```

In this simplistic example, the functions add and subtract would not need to be called from the main function:

```
def test(num1,num2,operation):
    def add(a, b): ← Local Functions
        return a + b
    def subtract(a, b): ←
        return a - b
    if operation == "+":
        return add(num1,num2)
    elif operation == "-":
        return subtract(num1,num2)

n1 = int(input("Input first integer: "))
n2 = int(input("Input second integer: "))
print("The sum was",test(n1,n2,"+"))
print("The diff was",test(n1,n2,"-"))
```

These are simply supporting functions for the function called test to complete its task. Consider this following example that has a function to figure the coefficient of variation based on a sample:

```
import scipy

def CV(a):
    def mean(a):
        tsum = 0
        for num in range(0,len(a)):
            tsum = tsum + a[num]
        return tsum / len(a)
    def std(a):
        tmean = mean(a)
        tsum = 0
        for num in range(0,len(a)):
            tsum = tsum + (a[num] - tmean)**2
        return scipy.sqrt(tsum / (len(a) - 1))
    # this next Line returns the coefficient of variation
    return std(a) / mean(a)

numlist = [11, 50, 80]
print(CV(numlist))
```

The local functions “mean” and “std” are merely supporting functions for the global function CV to use to calculate the coefficient of variation. We do not necessarily want these called from the main program. Consider the following change that tries to call these local functions:

```
7 import scipy
8
9 def CV(a):
10    def mean(a):
11        tsum = 0
12        for num in range(0,len(a)):
13            tsum = tsum + a[num]
14        return tsum / len(a)
15    def std(a):
16        tmean = mean(a)
17        tsum = 0
18        for num in range(0,len(a)):
19            tsum = tsum + (a[num] - tmean)**2
20        return scipy.sqrt(tsum / (len(a) - 1))
21    # this next line returns the coefficient of variation
22    return std(a) / mean(a)
23
24 numlist = [11, 50, 80]
25 print("Mean =",mean(numlist))
26 print("Std Dev =",std(numlist))
27 print("Coefficient of Variation = ",CV(numlist))
```

Notice that the Anaconda system warns us that there is a problem with calling mean and std directly! Once the program is run, the following message appears:

```
In [13]: runfile('C:/Users/David3/Documents/Python Scripts/test7.py',
  wdir='C:/Users/David3/Documents/Python Scripts')
Traceback (most recent call last):

  File "<ipython-input-13-89937c5ea1bc>", line 1, in <module>
    runfile('C:/Users/David3/Documents/Python Scripts/test7.py',
      wdir='C:/Users/David3/Documents/Python Scripts')

  File "C:\Users\David3\Anaconda3\envs\me2\lib\site-
  packages\spyderlib\widgets\externalshell\sitecustomize.py", line 585, in runfile
    execfile(filename, namespace)

  File "C:\Users\David3\Anaconda3\envs\me2\lib\site-
  packages\spyderlib\widgets\externalshell\sitecustomize.py", line 48, in execfile
    exec(compile(open(filename, 'rb').read(), filename, 'exec'), namespace)

  File "C:/Users/David3/Documents/Python Scripts/test7.py", line 25, in <module>
    print("Mean =",mean(numlist))

NameError: name 'mean' is not defined
```

Now, in a program, it might make sense to **WANT** to call the mean and std function. If so, we would want to convert these to global functions, as in this change:

```
import scipy

def mean(a):
    tsum = 0
    for num in range(0,len(a)):
        tsum = tsum + a[num]
    return tsum / len(a)

def std(a):
    tmean = mean(a)
    tsum = 0
    for num in range(0,len(a)):
        tsum = tsum + (a[num] - tmean)**2
    return scipy.sqrt(tsum / (len(a) - 1))

def CV(a):
    return std(a) / mean(a)

numlist = [11, 50, 80]
print("Mean =",mean(numlist))
print("Std Dev =",std(numlist))
print("Coefficient of Variation = ",CV(numlist))
```

Notice that all the functions (mean, std, and CV) are now global functions and can be called from anywhere in the program, even within another function.

Two major reasons to use local functions is that:

1. They are supporting functions that should NOT be called from the main program.
2. The programmers wish to free up the memory once the function is done (remember the variables will no longer be accessible).

Notes: Lambda Functions

Overview

Normal functions provide much power, but there might be certain circumstances in which a lambda function might be used. First, let's compare a regular function to a lambda function:

Regular Function

```
import scipy
def CV(pa):
    return scipy.std(pa) / scipy.mean(pa)

a = [11, 50, 80]
print(CV(a))
```

This code calculates the coefficient of variation from statistics. The scipy library allows access to the functions std (population standard deviation) and mean (population mean).

Lambda Function

```
import scipy
CV = lambda pa: scipy.std(pa) / scipy.mean(pa)

a = [11, 50, 80]
print(CV(a))
```

Lambda functions allow a single expression to be evaluated and automatically returned. Lambda functions are a tool to build function objects. Objects will be discussed later. While we could do everything with regular functions, Lambda functions allow for a more concise and cleaner approach.

While we will not see it in this course, certain graphical constructs require an argument that is a function. Instead of creating a function that is only used one time (which is contrary to the concept of functions), a lambda function can be used as the argument.

The same naming rules that apply to regular functions apply to Lambda functions. As with regular functions, multiple parameters can be passed in; however, only one value can be returned. One tutorial described the main purpose of Lambda functions as for building callback handlers.

Notes: Methods

Overview

The concept of a method is that it is an action performed by a specific object. Consider a student attending a college; our example object. There are various actions that a student can perform: study, attend class, graduate, etc. All of these actions affect the students themselves, though in programming this doesn't have to be the case. If these were written in a Python-flavored form, they could be listed as:

```
student.study  
student.attend_class  
student.graduate
```

These are the generic definitions for any student. Consider some parameters that might be passed into these methods:

```
student.study(hours)  
student.attend(class, date, time)  
student.graduate(date, degree)
```

If a student is going to attend class, it might be important to know which class, the date, and time. Usually, a specific instance is assumed for the student, such as

a student named Bill. Consider these examples with a specific instance and detailed parameters:

```
Bill.study(3)  
Bill.attend("Python Programming", 9/15/2014, 08:00)  
Bill.graduate(5/15/2015, "Computer Science")
```

This is an important concept in object-oriented programming, which will be discussed later in the course. For now, we will use existing methods for Python elements. For example, consider a list of numbers as in the following example:

```
a = [11, 50, 80]  
print(a)
```

This simply prints our numbers. Now consider in the Anaconda system if we start typing **a** and then a decimal:

```
a = [11, 50, 80]  
print(a)
```

```
a.  
append  
clear  
copy  
count  
extend  
index  
insert
```

The various pre-defined methods for a **list object** appear. Many of them are pretty obvious. For example, the append method would add a new number to our list:

```
a = [11, 50, 80]
print(a)

a.append()  
Arguments  
append(self, value)
```

Notice that a help popup will appear once we start typing the first parenthesis. It tells us the arguments (or parameters, as we refer to them) that are possible inputs. Not all of these are required. When the cursor hovers over **append** and CTRL-I is pressed, the following will appear:

append

Definition : append(self, value)
Type : Present in builtins module

append(self, value):
 L.append(object) -> None – append object
 to end

The Python documentation might present a list of the methods and their explanation that is easier to browse:

Table Of Contents

- 5. Data Structures
 - 5.1. More on Lists
 - 5.1.1. Using Lists as Stacks
 - 5.1.2. Using Lists as Queues
 - 5.1.3. List Comprehensions
 - 5.1.4. Nested List Comprehensions
 - 5.2. The `del` statement
 - 5.3. Tuples and Sequences
 - 5.4. Sets
 - 5.5. Dictionaries
 - 5.6. Looping Techniques
 - 5.7. More on Conditions
 - 5.8. Comparing Sequences and Other Types

[Previous topic](#)

[4. More Control Flow Tools](#)

[Next topic](#)

[6. Modules](#)

[This Page](#)

[Report a Bug](#)

[Show Source](#)

5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

`list.append(x)`

Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

`list.extend(L)`

Extend the list by appending all the items in the given list. Equivalent to `a[len(a):] = L`.

`list.insert(i, x)`

Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.remove(x)`

Remove the first item from the list whose value is `x`. It is an error if there is no such item.

`list.pop([i])`

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the `i` in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

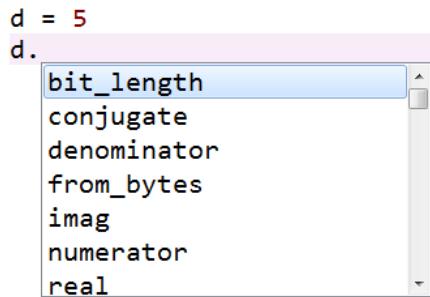
`list.clear()`

Remove all items from the list. Equivalent to `del a[:]`.

If our program is modified to append a new number to the list, the Python program would look like:

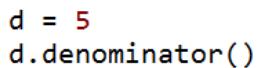
```
a = [11, 50, 80]
print(a)
a.append(99)
print(a)
```

Consider defining an integer variable `d` and typing `d` then a decimal:



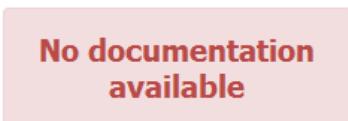
A screenshot of a Python code editor showing code completion for the variable `d`. The code is `d = 5`. A tooltip window lists several methods: `bit_length`, `conjugate`, `denominator`, `from_bytes`, `imag`, `numerator`, and `real`. The method `bit_length` is highlighted.

Elements that do not seem like they have any methods associated with them may have some defined. However, consider this example:



A screenshot of Python code showing `d = 5` followed by `d.denominator()`.

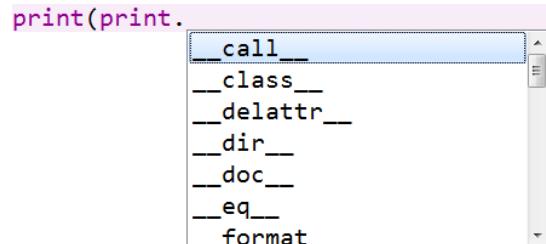
If the cursor hovers over the method name and CTRL-I is performed, the following message appears:



No documentation available

Do not assume that every method will have documentation in the Anaconda system. The Python documentation, as well as other resources, might provide that important missing piece.

Various items will appear once the decimal is entered. Consider this example:



Not everything that appears is a method.

Consider what `__doc__` will bring back with this example:

```
print(print.__doc__)
```

Once this is run it returns the following text:

```
print(value, ..., sep=' ', end='\n',
file=sys.stdout, flush=False)

Prints the values to a stream, or to
sys.stdout by default.
Optional keyword arguments:
file: a file-like object (stream);
defaults to the current sys.stdout.
sep: string inserted between values,
default a space.
end: string appended after the last
value, default a newline.
flush: whether to forcibly flush the
stream.
```

Until object-oriented programming is covered, remember that methods are simply functions for objects.

Notes: Names and DocStrings

Overview

Names and DocStrings are elements used mainly for help and documentation purposes. Consider the following example:

```
import math

def interest_continuous(p,r,t):
    """This function returns the interest when compounded continuously
       p is principle, r is rate (in decimal form), and t is time
       (in years)"""
    return p * math.e ** (r * t)
```

The comment enclosed in triple quotes is called the **docstring**. It is more than just simple documentation. Notice what happens when the programmer types the function name and the first parenthesis:

```
import math

def interest_continuous(p,r,t):
    """This function returns the interest when compounded continuously
       p is principle, r is rate (in decimal form), and t is time
       (in years)"""
    return p * math.e ** (r * t)
```

A screenshot of a Python IDE showing the code above. A tooltip window titled "Arguments" is displayed over the parameter list "interest_continuous(p, r, t)". The tooltip shows the text "interest_continuous()".

It displays the names of the parameters (arguments) required for this function.

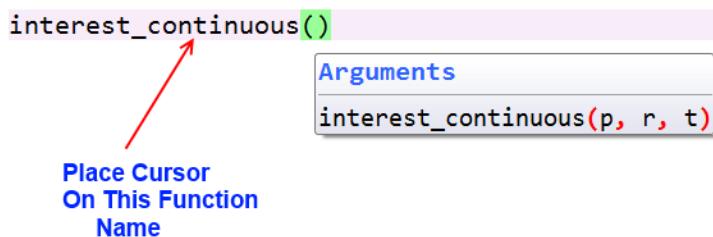
Now put your cursor over the function name:

```

import math

def interest_continuous(p,r,t):
    """This function returns the interest when compounded continuously
       p is principle, r is rate (in decimal form), and t is time
       (in years)"""
    return p * math.e ** (r * t)

```



Now press **CTRL-I** and we get the following documentation page:

interest_continuous

Definition : interest_continuous(p, r, t)
Type : Present in test7 module

interest_continuous(p, r, t):

This function returns the interest when compounded continuously p is principle, r is rate (in decimal form), and t is time (in years)

Notice that this text matches our docstring! We can also access the documentation by using syntax in the example from the previous section, similar to:

```
print(interest_continuous.__doc__)
```

This is just a brief overview of this concept of docstrings. This topic will be explored in greater detail once objects are covered.

Notes: Recursion

Overview

Recursion is a tricky concept. The idea of recursion is that the function calls *itself* over and over to accomplish some task.

Consider our factorial example of calculating 5!:

```
fact = 1
for num in range(1, 6):
    fact = fact * num
print("5!=",fact)
```

Now let's see this example redone with recursion:

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n - 1)

print("5!=",fact(5))
```

First Pass

Pass 1: On the first call to the **fact** function, **n = 5** and the else is performed (plugging 5 in for **n**):

```
return 5 * fact (5 - 1)
```

In the return statement, the fact (5 – 1) becomes fact (4) and calls the function again **before returning anything**.

Second Pass

Pass 2: On the second call to the fact function, **n = 4** and the else is performed (plugging 4 in for **n**):

```
return 4 * fact (4 - 1)
```

The fact (4 – 1) becomes fact(3) and calls the function again...

...and so on...

Once the last pass is called and **n = 1**, the values are plugged in and the following value is returned to the main program:

5 * 4 * 3 * 2 * 1

Recursive functions should have the following components:

1. An **if-else** statement must lead to different cases.
2. A base case to stop the recursion. In our factorial example above, the “if `n == 1:`” is the piece that stops the recursion. Without that part of the `if` statement stopping the process, it would be an infinite loop.

Let's consider an example that adds the number of “A”s in a string:

```
def a_count(str1,n):
    if len(str1) > 0:
        if str1[0:1] == "a":
            n = n + 1
        str1 = str1[1:len(str1)]
        return a_count(str1,n)
    else:
        return n

string1 = "happy days are here again"
print("The number of A's is",a_count(string1,0))
```

If a print function is placed into the program, we will see this program strip off the last characters as it calls itself over and over:

```
def a_count(str1,n):
    print(str1)
    if len(str1) > 0:
        if str1[0:1] == "a":
            n = n + 1
        str1 = str1[1:len(str1)]
        return a_count(str1,n)
    else:
        return n

string1 = "happy days are here again"
print("The number of A's is",a_count(string1,0))
```

It then gives the following display:

```
wdir='C:/Users/David3/Documents/Python Scripts')
happy days are here again
appy days are here again
ppy days are here again
py days are here again
y days are here again
  days are here again
days are here again
ays are here again
ys are here again
s are here again
  are here again
are here again
re here again
e here again
  here again
here again
ere again
re again
e again
  again
again
gain
ain
in
n

The number of A's is 5
```

The basic idea in programming is to keep it simple. If you can program the algorithm in a simple manner, using for and while loop constructs, this is preferable to recursion. Recursion is not simple and, as a result, may introduce unintended errors. Avoid recursion if possible!

Notes: Modules

Overview

Code reuse is an important concept in programming. In previous lecture notes, you have seen example after example dealing with compound interest. This formula does not change and it is redundant to keep programming it over and over. Modules can be thought of as “super-functions” in that they do not **ever** have to be programmed again.

Creating a Module

In a separate Python file, we will write the definitions for the two functions dealing with compound interest. We will save this file as compound_interest.py:

```
import math

def interest_continuous(principle, rate, time):
    """ This function calculates interest
        compounded continuously.
        p is principle, r is rate (decimal form),
        t is time (years)"""
    return principle * math.e ** (rate * time)

def interest_ntimes(principle,rate,time,n):
    """ This function calculates interest n
        times per year.
        p is principle, r is rate (decimal form),
        t is time (years), n is number of times
        compounded per year"""
    return principle * (1 + rate/n) ** (n * time)
```

Notice that it contains docstrings for documentation. The following import statement will allow access to these functions:

General Form:

```
from<python module name> import <function name>
```

Consider this example:

```
from compound_interest import interest_continuous  
ans1 = interest_continuous(1000,.05,5)  
print(ans1)
```

Notice that the **.py** file extension is not necessary for the Python module name. Also notice again that the Anaconda environment brings up the parameters (arguments) once the initial parenthesis is typed:

```
from compound_interest import interest_continuous  
ans1 = interest_continuous()  
Arguments  
interest_continuous(principle, rate, time)
```

Even though longer variable names make the module not as concise, it helps show the programmer what each variable is responsible for. Putting your cursor over **interest_continuous()** in the line `ans1 = interest_continuous()` and press CTRL-I, you will see the following:

interest_continuous

Definition : `interest_continuous(principle, rate, time)`

Type : Present in compound_interest module

`interest_continuous(principle, rate, time):`

This function calculates interest compounded continuously. p is principle, r is rate (decimal form), t is time (years)

This is the same as the docstring that was input into the module. In real life, this documentation should be very detailed, even including examples of usage. The above example just brought in one of our functions, but both can be brought into the program:

```
from compound_interest import interest_continuous  
from compound_interest import interest_ntimes  
  
ans1 = interest_continuous(1000,.05,5)  
ans2 = interest_ntimes(1000,.06,3,4)  
  
print(ans1)  
print(ans2)
```

This kind of importing requires knowledge of the function names. There is another kind of import that makes every function available:

```
import compound_interest  
  
ans1 = compound_interest.interest_continuous(1000,.05,5)  
ans2 = compound_interest.interest_ntimes(1000,.06,3,4)  
  
print(ans1)  
print(ans2)
```

Now it might appear at first glance that it is still necessary to remember the function names, but remember what happens once you type in the decimal:

```
import compound_interest  
  
ans1 = compound_interest.  
    interest_continuous  
    interest_ntimes  
    math
```

The Anaconda system brings up all the global functions defined in that module.

Both types of imports can be used with other libraries. Consider this square root example:

```
import math  
  
print(math.sqrt(2))
```

Instead of referencing the library name before the function, the function can be referred to directly:

```
from math import sqrt  
  
print(sqrt(2))
```

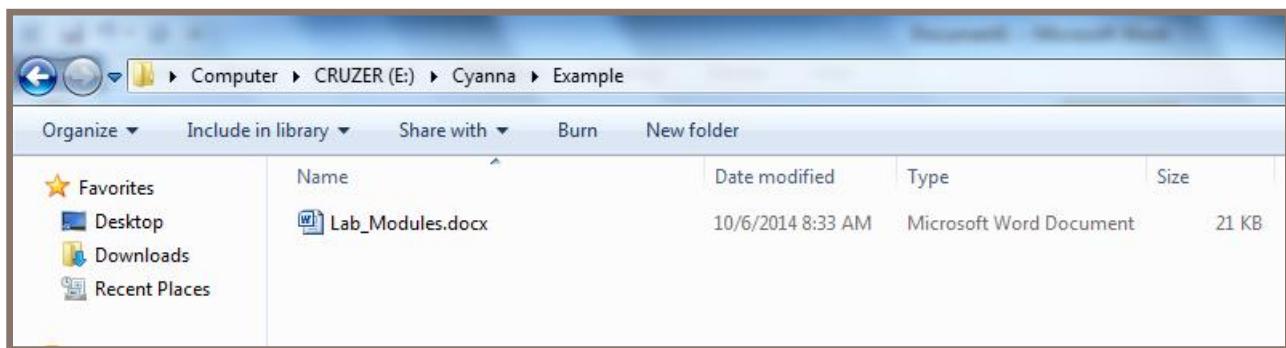
It is ultimately a personal preference to each programmer, but the first method (math.sqrt) identifies exactly where that function is coming from, which should make the program more readable.

Notes: File Handling

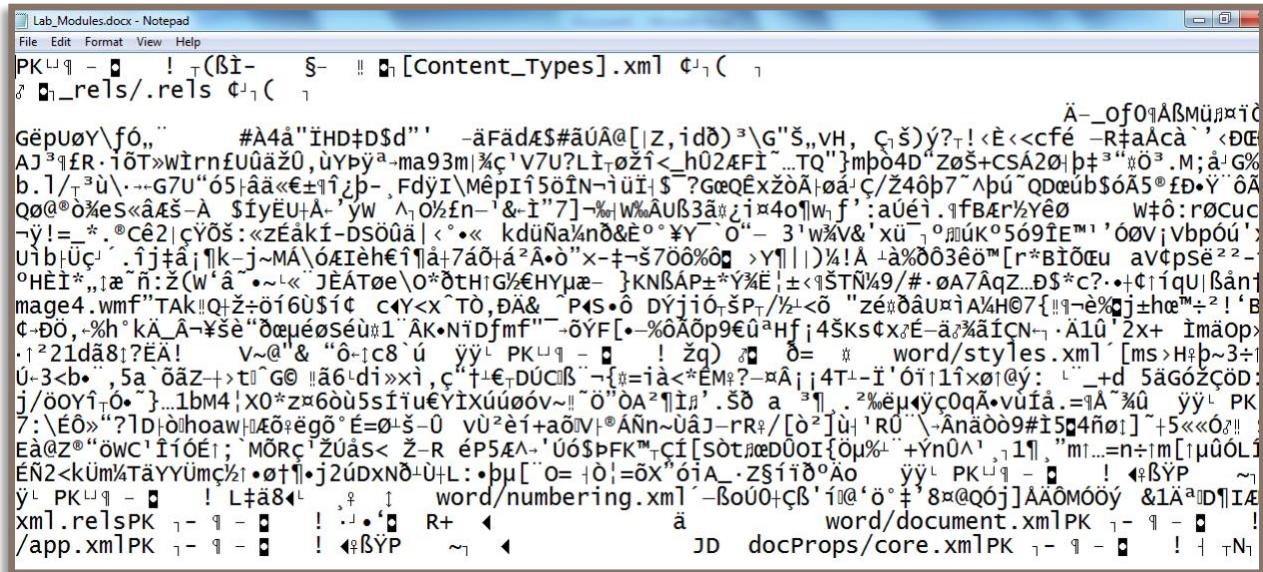
File Handling Overview

Python can save data in two different types of format: text and binary. The data stored in a textual format can be opened and viewed with an editor. Data stored in binary cannot be viewed in this manner since it is stored in binary format. This course will focus on textual. The current trend is for non-database files to be stored in a text-based format. Even Microsoft Office has adopted this strategy.

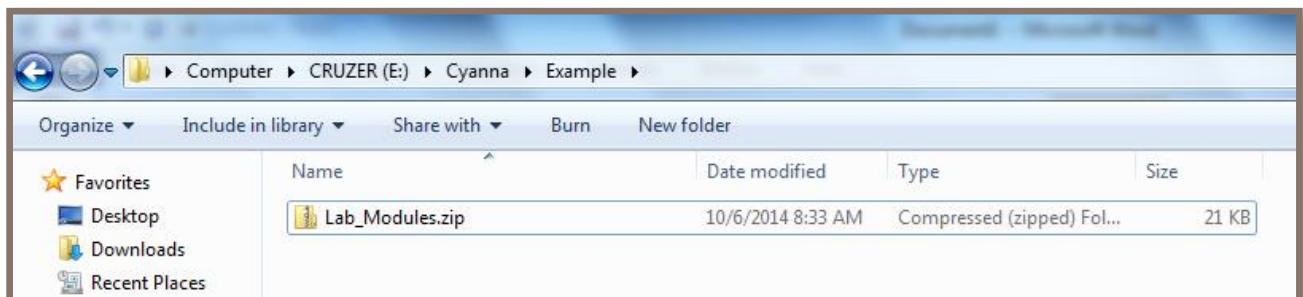
Consider the following Word document:



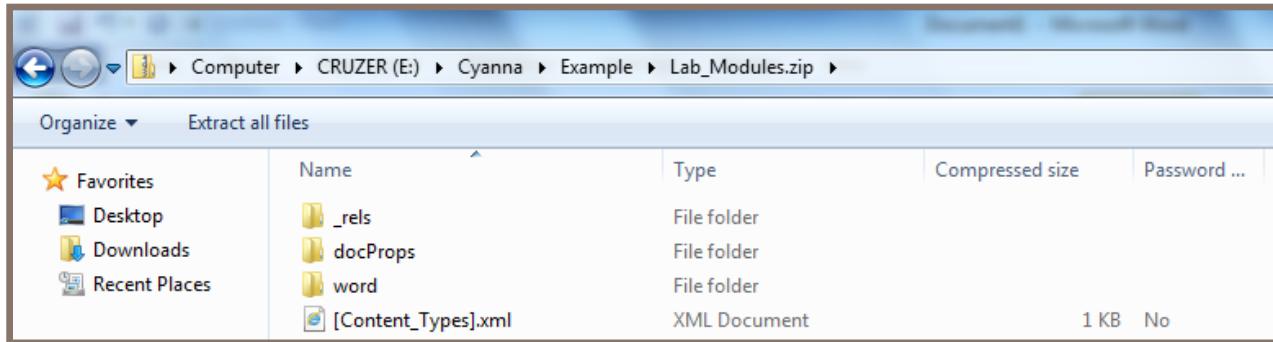
If it is open in Notepad:



It appears binary, but if we rename the file to Lab_Modules.zip:



And then double-click the file, the following appears:



We could explore this further and see that the Word document is actually a zipped folder of XML files, which can be viewed in an editor as in this file:

```

<?xml version="1.0" encoding="UTF-8" standalone="true"?>
- <w:document mc:Ignorable="w14 wp14" xmlns:wps="http://schemas.microsoft.com/office/word/2010/wordprocessingShape"
  xmlns:wne="http://schemas.microsoft.com/office/word/2006/wordml"
  xmlns:wpi="http://schemas.microsoft.com/office/word/2010/wordprocessingInk"
  xmlns:wpg="http://schemas.microsoft.com/office/word/2010/wordprocessingGroup"
  xmlns:w14="http://schemas.microsoft.com/office/word/2010/wordml"
  xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main" xmlns:w10="urn:schemas-microsoft-
  com:office:word" xmlns:wp="http://schemas.openxmlformats.org/drawingml/2006/wordprocessingDrawing"
  xmlns:wp14="http://schemas.microsoft.com/office/word/2010/wordprocessingDrawing" xmlns:v="urn:schemas-microsoft-
  com:xml" xmlns:m="http://schemas.openxmlformats.org/officeDocument/2006/math"
  xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships" xmlns:o="urn:schemas-microsoft-
  com:office:office" xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:wpc="http://schemas.microsoft.com/office/word/2010/wordprocessingCanvas">
  - <w:body>
    - <w:p w:rsidRDefault="00D86712" w:rsidR="001A0453">
      - <w:r>
        <w:t xml:space="preserve">Lab: </w:t>
      - <w:r w:rsidR="00040648">
        <w:t>Modules</w:t>
      - <w:r>
        <w:t>In the previous lab on functions you were g</w:t>
      - <w:r w:rsidR="00FA5CEC">
    - </w:p>
  - <w:p w:rsidRDefault="00040648" w:rsidR="00FA5CEC">
    - <w:r>
      <w:t>In the previous lab on functions you were g</w:t>
    - <w:r w:rsidR="00FA5CEC">
  - </w:body>
</w:document>

```

This textual-based format allows for easier transferring of data, as well as less corruption occurring. A programmer could change one byte (corrupt) in a binary file and the binary file would potentially be unusable. A textual format would

need much more corruption for this to occur (in theory, it all depends on how the original programmer programmed it).

In addition, this course will focus on sequential files. In a previous programming course, the topic of random access files may have been covered. Typically, this method should be avoided. As simple and widespread as databases have become, if some direct access is needed to data, then it should be done in a database.

Opening a File: General Format

The `file_variable` will allow the programmer to access the file object. The same naming conventions used in variables can be used for the filename, which is a string. Extensions are a good idea to identify the type of file! A mode is a string to indicate how the file is to be used: ‘r’ for read only, ‘w’ for writing (file is either created or for an existing file, then all data is removed), and ‘a’ for appending (new data is written at the end).

Syntax:

```
file_variable = open(filename, mode)
```

Examples:

```
F1 = open('sales.txt', 'r')
```

```
F2 = open('purch.txt', 'w')
```

Note: If the file is opened in append mode, then the original data is left in the file, but all new writes will occur at the end of the file. Along the same lines, if we are reading various datatypes into the program, we would need to convert them to an integer (int) or a floating point (float).

Closing a File: General Format

Example:

```
file_variable.close()
```

Writing to a File: General Format

file_variable is the variable that was referenced in the opening of the file. String can be a literal string or a string variable. The \n will go to the next line in the file. Remember that str will convert another data type to string:

Example:

```
num = 3
```

```
sales = 13.53
```

```
name='ABC Company'
```

```
F1.write(str(num) + ',' + str(sales) + ',' + name + '\n')
```

Reading from a File – Entire Content: General Format

This particular method is probably not going to be useful in most programming.

The programmer usually needs to read the file line-by-line, using the next method.

Syntax:

```
string_variable = file_variable.read()
```

Reading from a File – Line by Line: General Format

Syntax:

```
string_variable = file_variable.readline()
```

This method does read the file line-by-line, but would require the programmer to know how many lines are in the file. For example, if there are five lines in the file, the following code would extract that information:

Example:

```
infile = open('video.txt', 'r')
s1 = infile.readline()
s2 = infile.readline()
s3 = infile.readline()
s4 = infile.readline()
s5 = infile.readline()
infile.close()
```

Unfortunately, most of the time the programmer does not know how many lines there are in the file. This problem will be discussed later in this lesson.

Also the '\n' that is at the end may occasionally cause problems. This can be easily stripped out with the following code:

Example:

```
s1 = infile.readline()  
s1 = s1.rstrip('\n')
```

Detecting End of File Using While

Once the readline hits the end of the file, it will be set equal to " indicating the

Example:

```
infile = open('sales.txt', 'r')  
str1 = infile.readline()  
while str1 != "":  
    print(str1)  
    str1 = infile.readline()  
infile.close()
```

end of the file. The while loop is probably the simplest way to detect the end of the file.

Detecting End of File Using For Loop

The for loop provides a simpler interface.

Example:

```
infile = open('sales.txt', 'r')
for str1 in infile:
    print(str1)
infile.close()
```

Record Structure

Records are often stored in a single line in a file. This can be a predefined structure.

Example:

```
ABCCompany 35 123.45
111111111222222223
123456789012345678901234567890
```

The programmer might decide that columns 1 through to 12 are for the company name, columns 13 through to 18 are for the customer number, and columns 19 through to 30 are for the sales amt. Remember that strings are indexed starting at zero so, for a programmer, this structure would need to be viewed in this manner:

Example:

ABCCompany 35 123.45
111111111222222222
012345678901234567890123456789

The positions would change! Using string methods the information can be extracted, extra spaces removed, and a data conversion to the appropriate data type can be done.

Sometimes the data may be stored in CSV format and may look like this:

Example:

"ABC Company",35,123.45

While not as simple as a fixed column format, the information can be extracted using string methods. Even simpler, there is a library that would allow easier extraction of this data, as will be shown in a demonstration video.

Modifying an Existing File

It can be tricky to modify an existing file. The following scenarios present the methodology to accomplish the given tasks.

Case 1 – Add new records at the end of the file

In this case, simply open the file up with append mode and it is handled automatically.

Case 2 – Insert a new record in the file

1. Open a temporary file for writing.
2. Open the file for reading.
3. Loop through the file until you reach the insertion point. As you are looping, write the records straight from the file to your temporary file.
4. Write your new record to the temporary file.
5. Continue looping through the file, writing the remaining records to the temporary file.
6. Delete the file.
7. Rename the temporary file as the name of the file.

Case 3 – Delete a record in the file

1. Open the file for reading and the temporary one for writing.
2. Loop through the file until you reach the record to be deleted. As you are looping, write the records straight from the file to the temporary file.
3. Ignore the record to be deleted (do not write it to the temporary file).

4. Continue looping through the file, writing the remaining records to the temporary file.
5. Delete the file.
6. Rename the temporary file as the name of the file.

Case 4 – Changing a record in the file

1. Open the file for reading and the temporary one for writing/reading.
2. Loop through the temporary file until you reach the record to be changed.
As you are looping, write the records straight from the temporary file to the file.
3. Extract/change the information from the line as read from the temporary file.
4. Write this new record to the file.
5. Continue looping through the temporary file, writing the remaining records to the file.

Notes: Python Sequences

Overview

The most basic data structure in Python is the **sequence**. Python has several built-in types of sequences; the most common ones are strings, lists, and tuples.

- **Lists** are the most versatile sequence type. The elements of a list can be any object, and lists are mutable. Mutable objects can change their value but keep their `id()`. Elements in a list can be reassigned or removed, and new elements can be inserted.
- **Tuples** are like lists, but they are immutable. An immutable object has a fixed value. If a new value is needed, a new object has to be created.
- **Strings** are a special type of sequence that can only store characters, and they have a special notation. However, all of the sequence operations described below can also be used on strings.

Sequence Operators

There are standard sequence operations that are used in Python, which include:

- Slicing
- Length
- Concatenation of Sequences
- Checking for membership
- Repetitions

- Indexing

Indexing

Indexes are numbered from 0 to n-1, where n is the number of items (or characters) that is defined in the string, list or tuple.

Example:

```
str = 'Hello world'
```

To reference values in the string, its index looks like this:

-11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

H e l l o w o r l d

0 1 2 3 4 5 6 7 8 9 10

To access the values to the following:

str[3]

returns: 'l'

sStr[-1]

returns: 'd'

Slicing

Python makes it very easy to slice apart a string, list or tuple by using its slice operator. A **Slice** is used to extract a subset of a sequence.

- A slice has a starting point, an ending point, and an increment.
- Each parameter is optional.
- If not provided:
 - start defaults to 0
 - end defaults to the length of the list
 - increment defaults to 1
- The first colon is always necessary (to indicate Python should make a slice rather than access a single element), but if the increment isn't specified the second colon can be omitted.

Syntax:

```
mySlice[start:end:increment]
```

Examples:

```
s='abcd'
```

```
s[::-2]
```

returns: 'ac'

```
s[::-1]
```

returns: 'dcba'

Note: Print every second character of this string.

Length

The length of a sequence can be determined with the function `len()`.

- For strings, it counts the number of characters.
- For lists and tuples, the number of elements is counted.
- A sub-list counts as 1 element.

Example:

```
txt = "I love Python"
```

```
len(txt)
```

```
returns:13
```

```
random = ["this", 88, 3.14, "red"]
```

```
len(random)
```

```
returns: 4
```

Concatenation

Combines two or more sequences of the same type into a new sequence object.

Uses the + operator.

Example:

```
firstname = "Homer"  
surname = "Simpson"  
name = firstname + " " + surname  
print name  
returns: Homer Simpson
```

Checking for Membership

It's easy to check if an item is contained in or is a member of a sequence. This can be done using the “in” or the “not in” operator.

Example:

```
abc = ["a","b","c","d","e"]
```

"a" in abc

result: True

"a" not in abc

result: False

"e" not in abc

result: False

"f" not in abc

result: True

```
str = "Python is easy!"
```

"y" in str

Result: True

"x" in str

Result: False

Repetitions

To repeat a sequence, use the `*` operator to repeat the sequence n number of times, as shown in the following examples.

The augmented assignment for `"*"` can be used as well:

`s *= n` is the same as `s = s * n`.

Example:

To repeat a string 4 times, you can do:

(1) `str + str + str + str` or

(2) `str * 4`

(3) `3 * "xyz-"`

result: `'xyz-xyz-xyz-'`

(4) `3 * ["a","b","c"]`

Result: `['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']`

Notes: Lists

Lists Overview

A list can store multiple data items. For example, it can store 100 numbers. In your previous programming courses, “lists” were called “arrays.” Many of the same topics from arrays apply to Python lists.

Lists are objects which are covered later in this course, but some of the terminologies are mentioned in this lesson.

Creating Lists

- Lists are “constructed” using the “list” keyword.
- Elements in the list are enclosed in brackets and separated by commas.

Simple List Examples:

```
L1 = list();  
L2 = list([1,2,3,4]);  
L3 = list(range(1, 5));  
  
L1 = [];  
L2 = [1,2,3,4];
```

List creation can be simplified in the above manner.

The examples above were limited to just integers but any data type can be used in a list, as shown below.

Example:

```
L1 = ["Bill", "Sally", "Henry"];  
L2 = list("henry");
```

Example 2:

```
L2 = list("henry");  
print(L2);
```

L1 is fairly obvious but L2 requires a little bit more explanation. The string “henry” is passed into the list and each letter is split up as an element.

This returns the following:

```
['h', 'e', 'n', 'r', 'y']
```

The “list” in front of (“henry”) is what provides the conversion from the string to a list. Strings will not be covered in much depth in these lecture notes since it was covered previously. Remember that we treated strings as an array of characters and could do various functions with this idea, such as slicing.

List Indexing

Just as with arrays and strings, lists are indexed starting at 0. The string “happy” would have the following indexing.

h = position 0

a = position 1

p = position 2

p = position 3

y = position 4

This is the same concept that was demonstrated with strings and in your previous studies of arrays. The indexing always goes from 0 to the 1, less than the length of the string or list.

Index Operator

Example:

L1 = [3,5,7,9];

To access individual items in the list, the index operator can be used.

If the 5 needs to be accessed, the following form would be used: L1[1] (Remember the 3 is the 0 position while the 5 is the 1st position!).

Even though brackets indicate a list, to access the individual entries in the list also requires brackets. Consider this example, as seen in previous programming courses of a simplistic sort:

Example:

```
L1 = [5,2,1,8];  
for p1 in range(0,3):  
    for p2 in range(0,3):  
        if L1[p2] > L1[p2 + 1]:  
            temp = L1[p2];  
            L1[p2] = L1[p2 + 1];  
            L1[p2 + 1] = temp;  
    print(L1);
```

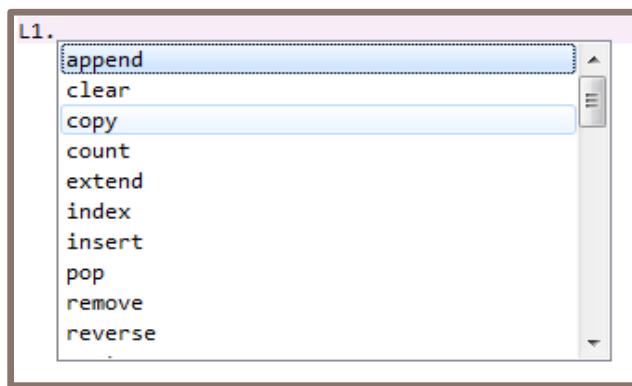
Note: The use of the brackets is used to refer to the entries inside the list.

Here is another example showing adding all the numbers in a list:

```
L1 = [];  
resp = "Y";  
while resp != "N":  
    resp = input("Input next number, N to indicate no more  
numbers: ");  
    if resp != "N":
```

Note: The use of the **append** method.

Remember that lists are objects which are different to the concept of “arrays”. If you find yourself wondering how to do something with lists, then try entering the list name followed by a decimal:



The benefits of a robust IDE is that the need for memorization of different programming topics is minimized.

Functions for Lists

Consider a list: L1 = [3, 5, 7, 9];

len(L1) would return 4

max(L1) would return 9

min(L1) would return 3

sum(L1) would return 24

These are actually functions not methods.

Methods act on an object such as a list.

Unlike methods, functions do not easily appear in an IDE by pressing a decimal.

Out of Bounds

Consider a list: L1 = [3, 5, 7, 9]:

One very common error is to try to access outside of the “bounds” for the list.

Remember, indexing starts at zero, so trying to access L1[4] will return an error.

Sometimes these types of errors will occur by attempting array programming that is carried out in other languages.

For example: L1 = [3, 5, 7, 9] dynamically determines the length of the list by the entries. A similar programming error:

```
L1 = [];
```

```
L1[0] = 5;
```

Note:

L1 - Creates an empty list

L2 – Creates an out of bounds error

The first line creates an empty list.

A logical assumption is that the second line would add a new element to the list and dynamically increase the size. It does not. It gives an out of bounds error. The proper code would be:

Example:

```
L1 = [];
```

```
L1.append(5);
```

When working with lists, a programmer will have less difficulty if they remember that lists are objects!

For Loop

As seen in the earlier example, a for loop must go from 0 to one less than the length of the list! This can be simplified in Python using the following code:

Example:

```
for index in range(len(L1)):  
    print(L1[index]);
```

Remember what the range function does and it handles this situation perfectly for lists!

Alternative for Loop

For each pass through the loop, u will contain the value of the next element in the

Example

```
L1 = [3, 5, 7, 9];  
for u in L1:  
    print(u);
```

list, starting at the 0 position.

Many of the same programming topics that were covered in strings also apply to lists, which makes sense if you think of a string as a list of characters.

List Slicing

Form: list[start : end]

This will return a list starting at the index = start and ending at the index = end – 1:

Example:

```
L1 = [1, 5, 13, 15, 21];  
L1[2 : 4] would return [13, 15]
```

Note: This returns a list that could then be assigned to another list variable.

Leaving off the start or end will extra all list entries from the beginning of the list or the end of the list.

+ Operator

Form: list1 + list 2

This will concatenate two lists.

Example:

```
L1 = [1, 2, 3];
```

```
L2 = [4, 5, 6];
```

```
L3 = L1 + L2;
```

L3 would not contain [1, 2, 3, 4, 5, 6]

* Operator

Form: scalar * list

Example:

```
L1 = [1, 2, 3];
```

```
L2 = 2 * L1;
```

L2 would not contain [1, 2, 3, 1, 2, 3]

This will replicate the list as many times as the scalar.

Keep in mind that this does multiply everything in a list by a number but also replicates the list!

in/not in Operator

This will check to see if an item is on or not on the list.

Example:

```
L1 = [1, 2, 3];
if 2 in L1:
    print("2 was on the list!");
```

The in operator provides a much easier interface than writing a search routine to search through the list.

List Methods

As has been mentioned, a list is an object that has various methods associated with it. Here is a table of the methods:

list	
append(x: object): None	Adds an element x to the end of the list.
count(x: object): int	Returns the number of times element x appears in the list.
extend(l: list): None	Appends all the elements in l to the list.
index(x: object): int	Returns the index of the first occurrence of element x in the list.
insert(index: int, x: object): None	Inserts an element x at a given index. Note that the first element in the list has index 0.
pop(i): object	Removes the element at the given position and returns it. The parameter i is optional. If it is not specified, list.pop() removes and returns the last element in the list.
remove(x: object): None	Removes the first occurrence of element x from the list.
reverse(): None	Reverses the elements in the list.
sort(): None	Sorts the elements in the list in ascending order.

Sometimes the programmer may find it of benefit to split a string up into a list, to take advantage of the methods of the list object.

Example:

```
colors = "red blue yellow".split();
```

The split method for a string will break a string into a list based upon the delimiter.

The split method with no delimiter assumes spaces.

Example with a Delimiter:

```
colors = "red|blue|yellow".split("|");
```

Consider the last line of code that uses the pipe symbol as a delimiter:

Example:

```
colors = "red|blue|yellow".split("|");
print(colors);
```

would return: ['red', 'blue', 'yellow']

As can be seen, this is a very powerful feature. Consider the following code:

Example:

```
L1 = "3 | 'david' | 5".split(" | ");
print(L1);
sum = L1[0] + L1[2];
print (sum);
```

Since 3 and 5 are numbers, then we expect that our sum would equal 8 but the result gives:

[3, "david", 5]

35

Python is a powerful programming language, but it is important to keep the data types in mind when designing and testing the program.

Receiving input from the user is simplified using this concept.

Example:

```
s = input ("input 5 numbers with spaces between them");
L1 = s.split();
L1Numbers = [eval(x) for x in L1];
print(L1Numbers);
```

Note: Five numbers were input using just ONE input!

When this program is run, it gives the following results:

input 5 numbers with spaces between them: 3 5 7 15 21

[3, 5, 7, 15, 21]

Copying a List

The easiest way is to copy the list item by item.

Example:

```
L1 = L2;
```

This may appear to copy one list to another but instead creates references that will complicate the Python program.

Functions and Lists

The procedure for passing in a list/returning a list versus a variable is the same, but lists by default come in as mutable (which means the function can change the value of the list directly). Sometimes creating a function that does not know the type (in the design mode of the Anaconda system) causes the methods to be unavailable. Consider this code once I press the decimal:

```
def sort(L):  
    L.|
```

No methods appear as the function does not know the data type. This does not mean that you cannot use the methods of the function, just that you will not get that extra help from the IDE.

```
def sort(L):
    L.sort();

L = [5, 2, 1, 3];
sort(L);
print(L);
```

As can be seen, it works fine but I had to remember the sort method of the list. Notice how confusing this code example would be. I have a function called sort and a method called sort.

Now we have a function, method, and parameter called sort. Very confusing. A better way to write this code example would be:

```
deflistsort(L1):
    L1.sort();

L = [5, 2, 1, 3];
listsort(L);
print(L);
```

Naming can make a world of difference in programming!

Multidimensional Lists

Lists can be embedded into another list for multidimensional lists (arrays):

Example:

```
numbers = [  
    [1, 3, 5, 7],  
    [9, 11, 13, 15]  
];
```

This creates a list that is 2 rows by 4 columns.

List items have to be referenced by row and column designation.

Example:

```
numbers = [  
    [1, 3, 5, 7],  
    [9, 11, 13, 15]  
];
```

To access the 9, the following code would be used: numbers[1][0].

Notes: Tuples

Overview

A tuple is a Python sequence object that is similar to a list. The main differences between a list and a tuple are:

- A tuple is an immutable object
- When it is defined, the whole set of elements is enclosed in parentheses instead of square brackets
- Any set of multiple objects, comma-separated, written without identifying symbols, i.e. brackets for lists, parentheses for tuples, etc., default to tuples

Why Use a Tuple?

- Tuples are used to store collections of heterogeneous data
- Tuples implement all of the common sequence operations (slice concatenate, etc.)
- Tuples have no `append` or `extend` method
- Tuples have no `index` method
- Tuples have no `remove` or `pop` method

Creating Tuples

To create a tuple, insert comma-separated data next to the tuple name. It is optional to put parenthesis around the values. The following are all valid ways to construct a tuple:

- Using a pair of parentheses to denote the empty tuple: ()
- Using a trailing comma for a singleton tuple: a, or (a,)
- Separating items with commas: a, b, c or (a, b, c)

Syntax:

```
tuple = ([data]);
```

Examples:

```
tuple ('abc')
```

```
returns ('a','b','c')
```

```
tuple ([1,2,3])
```

```
returns (1,2,3)
```

```
tuple ();
```

```
returns nothing
```

```
tuple (1,);
```

Note: To write a tuple containing a single value you have to include a comma, even though there is only one value.

Tuple Operations

We learned about the Sequence Operations in the previous lecture. The following is the list of Tuple operations and how they apply to Tuples.

Operation	Meaning
$t := [t_1, t_2]$	Concatenate tuples
$i := t $	Get number of elements of tuple t
$v := t[i]$	Select element i of tuple t ; $0 \leq i < t $
$t := t[i_1:i_2]$	Select from element i_1 to element i_2 of tuple t
$t := \text{subset}(t, i)$	Select elements specified in i from t
$t := \text{select_mask}(t_1, t_2)$	Select all elements from t_1 where the corresponding mask value in t_2 is greater than 0
$t := \text{remove}(t, i)$	Remove elements specified in i from t
$i := \text{find}(t_1, t_2)$	Get indices of all occurrences of t_2 within t_1 (or -1 if no match)
$t := \text{uniq}(t)$	Discard all but one of successive identical elements from t
$t := [i_1:i_2:i_3]$	Generate a sequence of values from i_1 to i_3 with an increment value of i_2

Operation	Meaning
<code>t := [i1:i2]</code>	Generate a sequence of values from <code>i1</code> to <code>i2</code> with an increment value of one

Operation Examples:

Length

```
Tuplen((1, 2, 3))
```

result 3

Concatenation

```
Tupconc(1, 2, 3) + (4, 5, 6)
```

result (1, 2, 3, 4, 5, 6)

Repetition

```
Tup Rep ('Hi!') * 4
```

result ('Hi!', 'Hi!', 'Hi!', 'Hi!')

Membership

```
TupMemb3 in (1, 2, 3)
```

Result True

Iteration

```
Tup Iterationfor x in (1, 2, 3): print x,
```

result 1 2 3

Tuple Indexing and Slicing

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings and lists.

- Tuples indices are zero-based, just like a list, so the first element of a non-empty tuple is always `t[0]`.
- Negative indices count from the end of the tuple, just as with a list:

Examples:

Assuming following input: Data = ('test', 'Test', 'TEST')

Data[2]

result = TEST

Data[-2]

result = 'Test'

Data[1:]

result = ['Test', 'TEST']

Note: Offsets start at zero

Negative: count from the right

Slicing fetches sections

Example 2:

```
Months = ('Jan','Feb','Mar','Apr','May','June', \
'Jul','Aug','Sep','Oct','Nov','Dec')
```

Note: The \ symbol allows the first line to carry over to the next line. It allows for readability.

The code is then organized in a numbered list starting with zero, in the order the values were entered. The list Months would look like this:

Index	Value
0	Jan
1	Feb
2	Mar
3	Apr
4	May

5	Jun
6	Jul
7	Aug
8	Sep
9	Oct
10	Nov
11	Dec

No Enclosing Delimiters

Any set of multiple objects, separated by a comma without identifying symbols:

Examples

```
print'abc',-4.24e93,18+6.6j,'xyz'; x, y =1,2;
```

```
result: abc-4.24e+93(18+6.6j) xyz
```

```
print"Value of x , y : ",x,y;
```

```
result: Value of x , y :12
```

i.e. brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples.

Updating Tuples

Tuples are immutable, which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples, as the following example demonstrates:

```
tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

tup1[0] = 100;

tup3 = tup1 + tup2;
print tup3;
result: (12, 34.56, 'abc', 'xyz')
```

Note: **tup1[0]** is not valid for tuples; a new tuple needs to be created called **tup3**.

Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded. To explicitly remove an entire tuple, just use the **del** statement:

Example:

```
tup=(python,'physics',2015,2016);
printtup;
deltup;
print"Deleting tuple : "
printtup;
result: ('python', 'chemistry', 2015, 2016)
```

Deleting tuple:

```
Traceback (most recent call last):
File "test.python", line 9, in <module>
printtup;
NameError: name 'tup' is not defined
```

Note: An exception is raised and this is because, after **del**tup, tuple does not exist anymore.

Notes: Higher Order Functions

Higher Order Functions Overview

Higher order functions imply that function names can be passed into functions.

This feature is powerful in certain mathematical and engineering applications. The average programmer would not see the benefit of using this feature.

Consider the composition of functions from college algebra. Given the following functions: $f(x) = 8x + 2$ and $g(x) = 9x + 3$, the composition of f and g is given by the following notation: $f \circ g$. In order to solve this problem, one function needs to be passed into another function, as in these steps:

$$\begin{aligned}f \circ g \\= f(g) \\= 8(9x+3)+2 \\= 72x+24+2 \\= 72x+26\end{aligned}$$

Consider another example of solving equations that are quadratic in form. Such as the following example:

$$x^{\frac{2}{3}} - 5x^{\frac{1}{3}} + 6 = 0$$

The u substitution results in the following equation:

$$u^2 - 5u + 6 = 0$$

The substitution of $x^{\frac{1}{3}}$ for u at the end of the problem is another example of passing one “function” into another “function” (in programming terms).

Python has a powerful library called SimPy. The goal of this library is to have a full-featured computer algebra system (CAS). Higher-order functions allow for a simpler development of certain mathematical topics.

Let's consider an example based on the composition of functions from above:

Example:

```
def f(x):
    return 8 * x + 2
def g(x):
    return 9 * x + 3
def composition(f1,f2,val):
    return f1(f2(val))
print(composition(f,g,4)) # f(g(4))
```

Note: The calling within the print function:
`composition(f,g,4)`.

The f and g are actual function names. Within the composition function, the return statement has:

`f1(f2(val))`

This applies the same concept as in algebra.

Example 2:

Note: This passes various SymPy functions as parameters to the function printresults.

```
importsympy;

defprintresults(n1,n2,f):
    print("answer is ", f(n1,n2));

printresults(8,12,sympy.gcd);
printresults(8,12,sympy.lcm);
printresults(8,12,sympy.primefactors);
```

Typically, the idea in programming is to keep it simple, so the feature exists if the algorithm is tricky and can only be done with higher order functions or mathematical programming, then you can apply the concept. Otherwise, it is best to avoid higher order functions and use a few more lines of code to make the program easier to read.

Notes: Mapping Types – Dictionaries

Overview

Dictionaries are objects that provide a key/value lookup capability. Typically these are values that are often accessed in a Python program and usually never change. For example, a state lookup might use dictionaries.

KS would be the key for the value Kansas

MO would be the key for the value Missouri

These are static values that typically never change or are loaded in at runtime; it is assumed that no change will occur during the time that the program is up and running.

- State sales tax might be an example of this situation.
- While sales tax may change, it would never change in the middle of the day.
- Assuming that the program is run in the morning, then it would load in the sales tax at startup and always have the current rate.
- On the other hand, if it is a system that is up and running 24 hours a day, then using dictionaries for the sales tax might not be a wise decision.

Obviously, a value that changes often, such as store sales, would be a poor choice for a dictionary lookup. Assuming your system has some kind of database capability, then database lookups are usually the preferred choice. A huge dictionary in a Python program would consume memory and be inefficient.

Dictionary: Creation

A dictionary can have as many key:value combinations as necessary but realize that practical considerations have to come into the design, such as how much memory would one million key:value combinations occupy? Assuming Python allowed that many, it is reasonable to assume that nobody has tried to load that many into a dictionary, as a database is the most practical solution to large volumes of data.

Syntax:

```
dictionaryname = {key:value, key:value, etc.}
```

Examples:

```
states = {"KS": "Kansas", "MO": "Missouri"};  
rate = {};
```

Note: The line of code: `rate = {}` creates an empty dictionary which then can have data loaded.

Dictionary: Adding and Modifying Entries

Syntax:

```
Dictionaryname[key] = value
```

Examples:

```
states["OK"] = "Oklahoma";
```

```
states["MO"] = "Missouri";
```

These examples will look for the appropriate key ("OK", "MO", etc.).

- If it finds that key then it will change the current value to the specified value.
- For example: states["OK"] = "Oklahoma"; looks for the key "OK" and if it finds it then it replaces the value with "Oklahoma".
- If it does not find the key, then it adds the key/value combination of "OK"/"Oklahoma" (without the quotations, of course).

These do not have to be hardcoded but could be loaded in from a file, database, or user input. Imagine a website that had a parts inquiry screen. The viewer of that webpage could input a part and it would return the quantity on hand or perhaps the price. Dictionaries would allow this lookup. Though obviously a large retailer with a million items on their webpage would not use dictionaries but a full database system.

Dictionary: Deleting Entries

Syntax:

```
del Dictionaryname[key]
```

Example:

```
del states["OK"];
```

Deleting entries is fairly self-explanatory. Keep in mind that, while these examples are hardcoding the state, the Python program could be driven off from user input!

Dictionary: Retrieving Values

Syntax:

```
Dictionaryname[key]
```

Example:

```
print(states["OK"]);  
str = states["KS"];
```

Dictionary values can be retrieved and used in other functions, or in various parts of a Python program.

- The print state looks up the key value “OK” and returns the value to the print function.

- The second line assigns the value to another variable.

Consider the following code:

Example:

```
states = {"OK":"Oahoma","KS":"Kansas"};
print(states["OK"]);
str = states["KS"];
print(states["MO"]);
```

“MO” is not in the dictionary and returns the following error message:

```
File "//exchsrv2003/Docs$/HaysD357147/My
Documents/Python Scripts/untitled3.py", line 11, in
<module>

    print(states["MO"]);

KeyError: 'MO'
```

Dictionary: Looping Through all Entries Example

Example:

```
for key in states:  
    print(states[key]);
```

Note: There is no hardcoding of the keys, but the for loop simply goes through all the “keys” and prints the values.

Dictionary: Determining the Size of the Dictionary

Syntax

```
len(dictionaryname)
```

In some cases, the size of the dictionary should be monitored to make sure that it is not becoming too large.

- Programmers are not limited to the standard try/except/finally construct.
- If some issue is arising, then the computer program could email the support team to let them know what is occurring.
- One program that I wrote automatically emailed me (based on a flag in the database) if the users were using my program.

- Sounds kind of silly, but parts of the production floor were very resistant to change and many of our programs that were installed were not being used.

Dictionary: Checking to see if the Dictionary Contains a Certain Entry

Example

```
states = {"KS": "Kansas", "MO": "Missouri"};
if "KS" in states:
    print("Found Kansas!");
```

It is not a bad idea to check to see if the key exists in the database prior to using it in the program. This avoids key errors or having to implement exception handling for this issue.

Dictionary: Methods

dict	
keys(): tuple	Returns a sequence of keys.
values(): tuple	Returns a sequence of values.
items(): tuple	Returns a sequence of tuples. Each tuple is (key, value) for an item.
clear(): None	Deletes all entries.
get(key): value	Returns the value for the key.
pop(key): value	Removes the item for the key and returns its value.
popitem(): tuple	Returns a randomly selected key/value pair as a tuple and removes the selected item.

Most of these entries are self-explanatory. The entry on the right side of the colon is the data type returned. For example, the clear() method returns nothing.

Notes: Exception Handling – Types

Overview

Simply put, exception handling refers to something going wrong in a Python program and how the program is going to handle the situation.

General ledger entries are read into a Python program and loaded into the database. The record structure should look like these examples:

1234ABC Company 33.41

2019XYZ Company 44.39

One record comes in truncated:

1234ABC Company 33.41

2019XYZ Compa

- The Python program fails to execute properly!
- How do you handle the situation?
- Open the input file, delete or correct that line, and run the program again?
- Fix the problem temporarily and then code exception handling into the program.

At one company where I worked, the programmer would temporarily fix the problem and get the program running again. No effort was put into solving the reason that it happened. This was not laziness because of the priority of other projects. The company made a determination that the correction of this “error”

was a priority. As a consequence, the number of programs that “crashed” significantly went down! Many companies do not understand this importance. In the long run, this approach reduces downtime for systems and programmers due to these errors, which in turn gives more time for “priority projects.”

Types of Exception Handling

Programmer-defined

Programmer-defined exception handling is done with programming constructs. For example, the programmer might check to see if the user has input an invalid state code, or if the number that is input does not fall within a pre-defined range.

System-defined

System-defined exception handling (try/except) handle the cases that programmer defined exception handling cannot check.

Assertions

Assertions are simple one-line exception handling meant to check the status of a Boolean condition, print the result, and then terminate the program.

For example, a `RuntimeError` is a `StandardError` which is a `BaseException`. All the properties associated with the higher level are inherited down.

Notes: Exception Handling – Usage

Exception Handling – Usage

Recall that exception handling is how to handle an unexpected error in your program. The unexpected error refers to a system error, not necessarily a programming error. Although the programming error could cause the Python program to terminate with a system error. This usually refers to an error that causes a program to terminate unexpectedly. An example is a conversion to an inappropriate data type, as in the following example.

Example:

```
num = "two"  
n = int(num)
```

- The first line creates a string variable called num and assigns the value “two” to the variable.
- The next line attempts to convert it to an integer.
- Normally it is fine to convert a string variable to an integer variable, but the value contained in the string variable should be an integer.
- The value “two” is not an integer, so the program would terminate with a system error.
- Exception handling allows the programmer to handle this system error in an appropriate manner.

- The Python program might give a user-friendly message about the data type and then terminate, or perhaps ask the user to re-enter the number.

Common Errors

Some common locations for exception handling in a program are during data conversions, file functions, and database access.

The reason for this specific focus is that these are where the majority of the errors occur. Access to outside components can bring in unexpected results. The main reason is that these components are outside the control of the Python program.

Outside Components

The following are situations of outside components that can cause errors:

- A file that a particular Python program reads in might be written out by another program.
- An external program might not have run or was programmed incorrectly.
- Perhaps the file was removed by accident by the user.
- The drive was full.
- Many conditions like the above mentioned could occur.

Real Life Example:

Three of your friends are meeting at your house to paint the house.

You have control of your own self:

- You know that you will be there.
- You will know how to handle an emergency that might come up.

Your friends are another issue:

- Will they show up?
- Will they work as you might expect?

These are outside components that can bring in unexpected results.

The same idea holds true in our Python programs.

In some data conversions, a strategy is to run the program and cause it to terminate by entering unexpected data. This allows the programmer to see the error code. Some programmers may look up the documentation, but causing the program to terminate allows the programmer to copy the error message exactly and paste it into the appropriate position.

In many cases, this method does not work. For example, how can you simulate a database reaching the limits of the size? In many cases, the programmer must use standard exception handling code and individually handle those cases when they occur. This is perhaps the exception handling code that is recommended by the vendor of the software product. In some cases, it is simple enough that it does

not need to be tested. Unfortunately, this idea causes many headaches. Code simple enough for no testing often fails but sometimes faith is necessary, as in the database example above.

Example:

A Progress database became too large and the entire weekend was spent correcting the issue. It had not happened in the previous ten years and never occurred again. In many cases, exception handling is not meant to handle the error that happens once in the lifetime of the system, but the more common issues. The criticalness of the system will determine the level of exception handling. Obviously, programs controlling ATMs or medical equipment would require more exception handling.

General Form:

```
try  
<code>  
exceptExceptionType as ex:  
<handler>  
else:  
<code>  
finally:  
<code>
```

The indented code following the try is performed. If a system error occurs, then the program immediately goes down to the lines containing “except.” The general form above only has one except, but the Python program may have numerous except – one for each system error. The “ex” is a variable name that allows access to further detailed information. The form could just have “except” with no exceptiontype. The else clause handles all the exception handling that is not covered by the exceptiontypes. The finally clause is code that occurs after the particular clause is performed.

Example:

```
try:  
    num = int(input("Enter a Number:"))  
    print(num)  
except NameError as e1:  
    print("Exception occurred: ", e1)
```

This particular example checks for a “NameError” and handles the error. Keep in mind that the appropriate error must be checked. If the programmer wanted to check to see if the data type was correct, then the error that should be checked is “ValueError.”

```
Try:  
    num = int(input("Enter a Number:"))  
    print(num)  
except ValueError as e1:  
    print("Exception occurred: ", e1)
```

If the programmer just wishes to generically handle all errors, then the error code can be left off, as in this example:

```
Try:  
    num = int(input("Enter a Number:"))  
    print(num)  
  
except:  
    print("Exception occurred ")
```

If the exception handling can be done without try/except, it should be done by the programmer. It is bad programming practice to let the program raise an exception when simple coding can prevent the situation.

Example

If your program is expecting a file to be read in, then check for the existence of the file prior to opening it.

Poor programming would be to let the Python program raise an exception on the missing file and use exception handling to correct the issue.

In the early days of Internet access, a program was written to send out emails for customer leads. If the Internet connection was down, the program would store the emails in a database and keep checking the status of the Internet connection. This exception handling was necessary at that time due to the uptime of the Internet connection.

The larger question concerns how to handle these emails in the database. There are many questions that arise. Why did the emails not go out? Was it just the Internet connection or could there have been other reasons? Did the exception handling take care of all the cases?

An exception handling solution placed in the program was to:

- Find and write all emails that were improperly formed into a file on the computer (not even into a database table!).
- The file eventually became too large and crashed the program.
- What are some considerations on this program?
- The exception handling was done right but the handling of the bad data was not considered.

Exception Handling – Control Flow Mechanism

This concept is introduced to emphasize the proper use of exception handling.

Although this could be debated, the main idea in programming is to keep it simple and exception handling to control flow can violate this idea. Over-using exception handling to control the flow of the program is considered poor programming practice. Many programmers consider this usage as equivalent to “gotos” in a

Try:

```
F1 = open(filename, 'r')
for str1 in F1:
    print(str1)
F1.close()
except FileNotFoundError as E1:
    print("File is missing " + filename + "Err: " + E1.strerror)
    choice = input("Do you wish to create the file? (Y or N): ")
    if choice == "Y":
        F1 = open(filename,'w')
        F1.close()
```

program. One example in the demonstration illustrates the proper usage of exception handling as a control flow mechanism:

This example will be discussed in the demonstration, but the idea is that we are treating the exception handling like a simple if statement:

```
if "the file is present":  
    code  
else if "the file is missing":  
    code
```

This code does not seem to reflect a goto situation but more of an if-else condition. The usage of exception handling should not cause “astonishment.”

The code above does not cause any astonishment.

It might be tempting to use exception handling for clever coding, but remember that another programmer may have to look at your code in the future. Perhaps even yourself!

The Principle of Least Astonishment states: “The result of performing some operation should be obvious, consistent, and predictable, based upon the name of the operation and other clues.”

Notes: Exception Handling – Control Flow Mechanism

The Overuse of Exception Handling to Control the Flow of a Program

“Keep it Simple”

When a Python script comes across a situation it is not “coded” to handle, it raises an exception. First, we need to understand what an exception is. An exception is an event that occurs during the execution of a program, which interrupts the otherwise normal flow of the program’s instructions. When the script raises an exception, it has to be handled immediately or the script will terminate. It is, therefore, considered as “bad programming” to use exception handling to control the flow of your program.

Python will raise an exception if your program attempts to do something inaccurate or pointless. To handle errors, you can set up exception handling blocks using the keywords *try* and *except*. When an error comes along, within a *try* block, Python will look for an *except* block. If it finds one it will jump there to look for a means to handle the error.

Exceptions should be reserved for exceptional situations, not for everyday normal flow control. Most programmers agree that you should use exception handling for handling unanticipated/exceptional situations, not as normal program flow.

The overuse of exception handling is an anti-pattern:

- Exceptions are basically sophisticated goto statements
- Overuse of exceptions makes the code difficult to read and understand
- Python, as well as other languages, has control structure built in and is better designed to solve the problems

Pros to the use of Exception Handling Control the Flow

Usually, exceptions are useful for allowing the programmer to concisely state two control flows, which are independent of each other, in the same piece of code: one if the error happens and one if the error doesn't occur.

There are a few occasions when it is “more” acceptable to use exceptions than others. For instance, during a busy day, you encounter a situation you don’t have time to deal with right now. You don’t have enough information to deal with the issue correctly the first time. In this case, if writing in a try/except works to bring everything back online and working, all well and good. However, you should write quality documentation on what you did and why, to allow for you or someone else to further investigate and correct the problem later.

It is also reasonable to use exception handling as a quick way to advert a control flow problem that has the potential to cause serious data corruption or some other lasting damage. Keep in mind, if you aren't using exception handling as a quick fix to avoid disaster or bridge to hold up the program until you have more time, there is probably a better way to achieve your objectives.

Notes: Assertions

Overview

Assertions are very short pieces of code used to check the internal status of your Python program.

- They might check the status of a dictionary entry or the type of a variable.
- These are one-line pieces of code to check a Boolean condition and, if false, then print a message and terminate.

Syntax:

```
assert <Boolean condition>, "programmer written error  
message"
```

If the Boolean condition is false, the programmer written error message is displayed on the screen and the program terminates.

Example:

```
class MyDB:  
    def __init__(self):  
        self._id2name_map = {}  
        self._name2id_map = {}  
  
    def add(self, id, name):  
        self._name2id_map[name] = id  
        self._id2name_map[id] = name  
  
    def by_name(self, name):  
        return self._name2id_map[name]  
  
    def by_id(self, id):  
        assert self._id2name_map[id] == name  
        return id
```

The last function can be changed to incorporate an assertion:

Note: The assert statement: assert self._id2name_map[id] == name is checking the integrity of the data.

- Do not worry too much about the details of this code.
- The assert statement: assert self._id2name_map[id] == name is checking the integrity of the data.
- No message is displayed in this example, but the programmer wants to do the quick check and just terminate the program if the integrity is inconsistent.
- This causes the program to terminate.
- This is the idea of asserts: they check a simple condition and terminate if the condition is “false.”

Example:

```
def add(self, id, name):  
    assert type(id) is IntType, "id is not an integer: %r" % id  
    assert type(name) is StringType, "name is not a string: %r" %  
        name
```

- These asserts check the type and handle it appropriately.
- Notice that these asserts have a message displayed.

- Proper programming usually requires some display of debugging information, such as the value of variables that might have caused the assertion error.

According to the Python documentation:

- Checking parameter types, classes, or values.
- Checking data structure invariants.
- Checking “can’t happen” situations (duplicates in a list, contradictory state variables).
- After calling a function, to make sure that its return is reasonable.

Notes: Efficiency Considerations

Overview

Efficiency is sometimes important and sometimes unimportant. For example, one mainframe process done inefficiently had to be run at night due to the hours required for it to be run. It was changed to use keys in the database and it ran in just a few minute's time. The process was going through millions of records sequentially and efficiency became very important!

At a particular company, I spent time improving the efficiency of programs running on the mainframe. We leased the mainframe and paid a set amount per month. This leased price was then charged out to the internal customers per CPU cycle. I ended up saving the company approximately \$250,000. I received a memo praising my efforts.

A Real-Life Example:

I took a report that cost an internal customer \$500 per run and reduced it to \$2 per run. I felt good about my work, but when I went to another customer, who was more knowledgeable on business concepts, and told him what I had done and wanted to examine their programs for efficiency improvements, he asked me if the mainframe was leased. I replied that it was. He then asked if the leased price had gone down. It had not. He pointed out that I merely caused the cost of the CPU cycle to increase and that I did

not save the company \$250,000 since we were still paying the same price to the vendor. He was correct! He said that sometimes a company getting their result faster might result in profit but, otherwise, my efforts were not beneficial. He was not being mean but helping me to understand. He said that he had other projects that he would rather I work on, which would be profitable for the company. It taught me a valuable lesson that efficiency has to be viewed in terms of the larger picture of the company.

Efficiency has to be viewed in terms of the larger picture of the company.

Consider another process that takes 5 seconds to run. Perhaps it is programmed inefficiently. After maybe five hours of changes in the program, then the process now takes 1 second to run. Was it worth the effort? Does a company want to pay you for this improvement?

- 5 seconds versus 1 second; is this significant?
- 5 seconds to bold an item in Microsoft Word would seem like an eternity!
- 5 seconds to wait for a process that is run only on Monday mornings would be acceptable!
- 5 seconds for a prompt on an ATM machine would seem like an eternity!
- 5 seconds for a printer process to start printing is acceptable!

Sometimes programmers are short-sighted! The process perhaps takes 1 minute to run when there are 100 records in the database. This seems acceptable. How

about in 5 years' time, when there are 10,000,000 records in the database?

Proper consideration of efficiency might prevent major headaches in the future!

Theoretical Approach to Efficiency

It is difficult to anticipate what the effects will be in the future since they have never occurred before! A theoretical approach allows for a measure of the efficiency. The big O notation is a function that measures the complexity of the algorithm based on input size.

Linear Search

- Each number is compared to the search value.
- If it does not match, then it goes to the next one.
- If the number of entries is n , then:
 - If the search value is not on the list, then n searches have to be performed.
 - If the search value is not on the list, then $n/2$ searches on average have to be performed.
- $O(n)$ refers to the time.
- In analyzing the algorithm: $O(n) = O\frac{n}{2} = O\frac{1}{2n}$, constants are ignored.
- An algorithmic analysis is based on growth rate.

$T(n)$ refers to the time complexity. In general: $T(n) = c \times n = O(n)$

Example:

```
for i in range(10):  
    sum = sum + i;  
  
for i in range(n):  
    for i in range(30):  
        sum = sum + i;
```

$$T(n) = \mathbf{10} \times c + \mathbf{30} + c \times n = O(n)$$

10 × c refers to the first for loop which loops from 0 to 9

- This is a total of 10 iterations.
- If the for loop was “for i in range(20),” that piece would be $20 \times c$

30 + c × n refers to the second main for loop which loops from 0 to n-1

- N is the number of iterations within the loop
- It goes from 0 to 29 or 30 iterations for each value of the outer for loop.
- This explains $30 + c \times n$

Consider the following for loop:

```
for i in range(n+1):
    for j in range(n):
        for k in range(n+5):
```

This would consist of $(n + 1) \times n \times (n + 5) \times c$ for the efficiency.

What is C?

The question becomes: what is c and how to figure it? This is a hard concept.

- How long will one iteration of a for loop take?
- It is a simple programming task to find this on a particular computer, but then how long will this take on an older or newer model?
- Industry or company specific charts may help in this determination.
- Will the multi-tasking capability of the computer affect c?
- What are the typical applications that will be running when the Python program is run?

These are just some of the considerations that must be addressed. Some analysis just assumes a perfect world with nothing else running, as there are too many variables to consider. Many users of computers have purchased new computers and noticed the startup time decreasing significantly when logging into their computers in the same operating system. Some of this can be explained by programs running at startup on the old computer but not on the new computer. Much of the improvement is due to increased hardware speed.

Notes: Black Box Testing

Overview

Black Box Testing is a testing method where no knowledge of the inner workings or code is known. The easiest way to think of this concept is for a user to use the computer program. This testing might be done by a user of the system or by the actual programmer. Either way, the tester assumes no knowledge of the program, just the interface.

The black box refers to the tester not being able to see inside the box.

The tester is working from specifications and requirements. These might be written in a document or done on the fly.

For example, if the program is meant to calculate the factorial, the user input would be a number, as in this prompt:

Input a number: _____

The tester will then see if the program works according to the specifications and how it reacts in various scenarios.

Examples:

What happens when I put in a negative number?

What happens when I put a letter in the program?

What happens when I put in too large of a number?

What happens when I put in a decimal number?

It is important that the tester document the exact inputs and the result that occurred so that the programmer can recreate the problem.

Test Design Techniques

- Decision table testing
- All-pairs testing
- State transition analysis
- Equivalence partitioning
- Boundary value analysis
- Cause-effect graph
- Error guessing

Not all the techniques will be used in every Black Box Testing scenario. The criticalness of the application will help determine the depth of the testing.

For example, if there is one user who knows what a factorial means, the testing may be minimal. However, if there is going to be 300,000 users who have no clue what a factorial is, then improper Black Box Testing may generate 30,000 support calls!

Notes: Glass Box Testing

Overview

Glass Box Testing is a method of testing the application by assuming knowledge of the programming language and internal design.

The glass box name implies that the tester can see “inside the box” and know all the details of the programs, as well as the supporting components such as files, databases, etc.

The tester is again working on specifications and requirements. These might be more general or, in some cases, more specific. An example of a general requirement is that all test cases are included that encompass all the lines of a computer program.

For example, a Python program might have an intricate “if” structure with many decision paths. Proper testing would make sure that each decision path functions as required.

Example:

```
if grade >= 90:  
    print("You received an A");  
elif grade >= 80:  
    print("You received a B");  
elif grade >= 70:  
    print("You received a C");
```

And so on

It would be important for the tester to have test cases to check all the different branches of the if structure. For example, the tester might input 92, 83, 75, 62, 53, etc., to check all the different paths possible in the if structure.

Perhaps the application is based on a database. The tester might want to check all the conditions that would happen if something is wrong with the database. This is easier said than done. How do you simulate corrupting a database? Sometimes the error conditions that happen cannot be explained and, as a consequence, cannot be recreated.

Although many individuals will indicate that testing should be thorough and complete, this is not actually the case. The criticalness of the application, delivery date, support issues and many other factors help to dictate the depth of the testing.

Consider how much testing is required for an ATM program compared to a simple Excel VBA macro for one user.

Improper testing on the ATM program could cost a bank a huge loss, while improper testing on the VBA macro may be a minor loss. Keep in mind that a VBA macro error could be very costly, depending upon the criticalness of the spreadsheet!

Test Design Techniques

- Control flow testing
- Data flow testing
- Branch testing
- Statement coverage
- Decision coverage
- Modified condition/decision coverage
- Prime path testing
- Path testing

Most programmers do most of the above techniques but do not actually write them down.

This method keeps programmers accountable. Many programmers assume that, if it works for one value, it will work for any value. Often, this is not due to laziness but the time demands of the occupation. Although robust testing can be tedious and some programmers have to be forced to do proper testing!

Notes: Debugging Techniques

Overview

The Python program is not working correctly; what can the programmer do?

1. **Simple inspection:** The programmer should start off with a simple inspection of their program. Look for any indications from the Anaconda system that there are issues. See if the spacing is correct for the various programming constructs.
2. **Is it running Your code?** Use simple print statements to see if it is even running your code. If it never reaches the code that you are debugging or

Example:

```
if sales > 30000;  
    commission = sales * 0.10;
```

runs it inconsistently, then the flow of the program is incorrect.

3. Modify your code (temporarily)

Example 3.1: If your program is running a loop 500,000 times and it is difficult to debug, then change the loop to run twice and check the results.

Example 3.2: Comment out code that is working and focus on the code that is not working.

4. **Copy your code then modify:** Sometimes it is beneficial to copy your code, comment out the original code and then start changing the copied code. Perhaps the code change is temporary or the programmer is just experimenting.
5. **Keep it simple:** If you are trying to be clever or concise with your code then change the code. Clever code should be left for the times when you truly need to be clever in programming. The majority of programming is repeating the same code over and over for different circumstances. For example, for five years I always did the same generic code for database access.

Example:

```
Tempquery.close;  
Tempquery.clear;  
Tempquery.sql.add("SELECT * FROM SALES;");  
Tempquery.open;
```

- Although there are more clever ways to implement database lookups, I always programmed it in the simplest manner.
- If you are developing your own algorithms, then the keep-it-simple idea may not hold but try to follow this principle as much as possible.

6. **IDE features:** Breakpoints, variable watches, and many other features are available in IDEs. Explore these options if you plan to get serious about a particular development platform. These features could save hours and hours of debugging.

7. **Another pair of eyes:** I have spent hours looking for an error and then breaking down, having another programmer look at the code and immediately seeing the problem.

- Another pair of eyes also means that you set aside the program and take a break from the examination.
- Work on some other project or get a good night's sleep! Sometimes then the problem is obvious.

8. **Incremental development:** One popular approach is to develop small segments of code and to test each segment as you write the code. It is often easier to debug a few lines of code than a hundred lines of code.

9. **Do not reinvent the wheel:** In programming (not necessarily in classwork!) it is perfectly acceptable to copy another person's code. Many companies would rather a programmer spend five minutes copying code off the Internet versus five days writing their own code. If the code you are trying to debug is a common piece of code, then spend a few minutes searching the Internet. Open source code has become popular and the concept of sharing has also become popular in coding! I was debugging an email program written in Perl that used sockets. It was a program that was written by another programmer. The programmer did not keep it simple. Finally, I asked the user if it has to be in Perl using sockets. They had no clue what Perl and sockets were but stated that they just wanted it to work. I found an ASP email example on the Internet and copied it. Then I did some minor testing and the process was working in a very short period of time.

The above are just a few ideas. This is not the complete list but just ones that I used in my years of programming.

Notes: Classes

Overview

Object-oriented programming has simplified certain programming. For example, game development would be extremely difficult without objects.

- Many features of a development environment/programming language have improved with the use of objects.
- Some languages have implemented various programming constructs as objects.
- The programming language Smalltalk intended for everything in the language to be an object.
- This object-based philosophy of Smalltalk helped to influence future programming languages in their implementation of object-oriented programming.

Objects should not be used in every circumstance though (despite what Smalltalk indicates)!

Defining Classes – General Form

Syntax:

classClassName:

initializer

methods

Example:

classRect:

```
def __init__(self,length=1,width=1):
```

```
    self.length = length;
```

```
    self.width = width;
```

self refers to the object invoking the particular method.

```
class Rect:  
    def __init__(self,length=1,width=1):  
        self.length = length;  
        self.width = width;
```

The initializer creates an instance of the object. It is specified by two underscores before and after the “init” part.

```
class Rect:  
    def __init__(self,length=1,width=1):  
        self.length = length;  
        self.width = width;
```

Length and **width** are the parameters. Since these are set equal to one then, if the programmer does not pass in any parameters, the value of length and width will be equal to one.

```
class Rect:  
    def __init__(self,length=1,width=1):  
        self.length = length;  
        self.width = width;
```

Every function in a class definition must have the word “self” as the first parameter. This is not actually passed in when you call it but must be there.

```
class Rect:  
    def __init__(self,length=1,width=1):  
        self.length = length;  
        self.width = width;  
        Instance  
        Variables
```

By using variables with
“**self.**” in front of
them these are
automatically
instance variables.

Note: Assuming the library Rectangle.py contains the definition of
the Rectangle object.

Creating an Instance – General Form

Syntax:

```
objectRefVar = ClassName (arguments)
```

Example:

```
import Rectangle as r  
r1 = r.Rect(3,5);  
r2 = r.Rect();
```

The first instance r1 is created passed in 3 and 5 as parameters, which means its instance variables are self.length = 3 and self.width = 5. The second instance r2 is created passing with no parameters, which means that its instance variables default to the value one 1: self.length = 1 and self.width = 1.

Accessing Instance Variables

Although many consider it poor programming form, instance variables can be accessed directly.

Example:

```
r1 = r.Rect(3,5);  
print(r1.length);  
r1.length = 8;  
.....
```

Example:

```
classRect:  
def __init__(self,length=1,width=1):  
    self.length = length;  
    self.width = width;  
defgetLength(self):  
    return self.length;  
defsetLength(self,param_l):  
    self.length = param_l;
```

Many programmers adhere to the standards of using get and set methods to change the value of the instance variables.

Notice the naming convention of the methods “getLength” and “setLength”.

Typically, these methods start with get and set. The languages do not require it but it is accepted programming practice. The method names could actually be any valid name, as shown below.

The standard allows anybody who has worked with objects to easily understand the purpose of these two methods. The “get” method usually just returns the value of the instance variable, while the “set” method usually just sets the value of the instance variable. The code to then access the instance variables would follow the example below.

Example:

```
classRect:  
    def __init__(self,length=1,width=1):  
        self.length = length;  
        self.width = width;  
    defgL(self):  
        return self.length;  
    defsL(self,param_l):  
        self.length = param_l;
```

Think of the interface of a television. The user could open up the back of the computer and manipulate the circuit boards to change the channel, but the designers did not wish the user to access it in this manner. A remote control and/or buttons were implemented to provide a “safe” interface. The “get” and “set” methods perform the same function.

Methods

“Get” and “set” methods are not the only pieces of code that can be implemented for a class. Various methods can be defined to accomplish various tasks.

Example:

```
classRect:  
    def __init__(self,length=1,width=1):  
        self.length = length;  
        self.width = width;  
    defgetLength(self):  
        return self.length;  
    defsetLength(self,param_l):  
        self.length = param_l;  
    defcalcPerimeter(self):  
        return 2 * self.length + 2 * self.width;  
    defcalcArea(self):  
        return self.length * self.width;
```

Notice that the last two methods have been implemented to calculate perimeter and area. Notice that `self.length` and `self.width` are not passed in as parameters. The instance of the object already knows its own instance variables. This is a similar idea of an instance of a human being already knowing their name, age, etc.

These methods can be called as in the following:

Example:

```
print(r1.calcArea());  
print(r1.calcPerimeter());
```

Should area and perimeter be methods to call or should these be instance variables? These are questions that the programmer must decide upon when designing the method. Benefits exist with both methods. The general idea is that instance variables are “natural” occurring properties that describe the object. For example, a human being might have the properties: first name, last name, birthdate, and age. Age is not something that should be calculated each time it is asked for. Imagine the look on an individual’s face if they asked your age and you had to do the calculation in front of them. How about if you were asked how many blocks you live from a local school? This is a calculated item and would make a good method. Consider if area and perimeter were instance variables:

```
def __init__(self,length=1,width=1):  
    self.length = length;  
    self.width = width;  
    self.area = length * width;  
    self.perimeter = 2 * length + 2 * width;
```

Then each time the length or width have changed the area and perimeter would have to be recalculated:

```
def setLength(self,param_l):  
    self.length = param_l;  
    self.area = self.length * self.width;  
    self.perimeter = 2 * self.length + 2 * self.width;
```

Some true purists of object-oriented programming might argue that a “set” method should only apply to the variable specified. For example, “setLength” should only code affecting the length. In practice, this is not practical.

Notes: Class Inheritance

Overview

Inheritance is seen in public education with the taxonomy of living creatures. In a simplistic view, a primate is a mammal which is an animal. Since a mammal is an animal, it “inherits” all the characteristics of an animal. Since a primate is a mammal, it “inherits” all the characteristics of a mammal.

In programming, this allows the programmer to concisely define the instance variables/methods at any level. In certain programming fields, this is an important concept.

The breakdown can be a logical breakdown that has been seen in past courses or any definition.

Examples:

Company

- Department

- Employees

- Buildings

- Office Equipment

Example:

House

- Rooms

- Dining Room

- Living Room

- Bedroom

- Bathroom

- Residents

A company consists of departments and buildings. Within the department, there are employees and, within the building, there is office equipment, which is further broken down into desks and chairs.

This breakdown does not indicate that there will only be one bedroom or bathroom, it is merely defining the “types” of rooms.

- Another programmer defining this same breakdown might have a different interpretation.

- The key is whether the object definition can be applied to every instance of the object.
- This does not apply to every instance across the entire universe.
- Perhaps there are houses that do not have the same structure.
- Perhaps a better breakdown for the house might be:

House

- Rooms “with instance variables of type (dining, living, sleeping, bath, etc.)”
- Residents

Types

By having the type defined as an instance variable, the object definition is more flexible.

In programming, this allows the programmer to concisely define the instance variables/methods at any level. In certain programming fields, this is an important concept. For example, in certain games, there may be 100 different characters. The basic movement is the same for the characters but perhaps their appearance is different. The hierarchy would look like:

character

hero

villain

Polynomial

Polynomials provide a good example of inheritance. From algebra, we know the breakdown is as follows:

Polynomial

- Monomial
- Binomial
- Trinomial

This could also be defined using a quadratic.

Example:

```
class Polynomial:  
    def __init__(self,degree=1):  
        self.__degree = degree;  
    def getDegree(self):  
        return self.__degree;
```

Stored in a file called Polynomial.py

Note: The `setDegree` method is missing. Perhaps the programmer does not wish for the degree to be changed after the creation of the instance. The `setDegree` would not be necessary.

Trinomial

The trinomial class would be defined as:

```
import math  
from Polynomial import Polynomial
```

```
class Trinomial(Polynomial):  
    def __init__(self,a=1,b=1,c=1):  
        super().__init__(2);  
        self.__a = a;  
        self.__b = b;  
        self.__c = c;  
    def getZeros(self):  
        ans1 = (-1 * self.__b + math.sqrt(self.__b**2 - 4 * self.__a *  
                                         self.__c))/(2 * self.__a)  
        ans2 = (-1 * self.__b - math.sqrt(self.__b**2 - 4 * self.__a *  
                                         self.__c)) / (2 * self.__a)  
        return ans1, ans2
```

Stored in a file called Trinomial.py

The two main pieces that cause the inheritance are:

```
class Trinomial(Polynomial):
```

Note: Inside the parenthesis is the name “**Polynomial**.” This is the class it inherits from and is brought into the code from the “import” line.

The piece is:

```
super().__init__(2);
```

The super refers to the class it is inherited from and the `__init__` calls the initializer method (construct the instance).

The calling of this inheritance scheme is as follows:

Example:

```
import Trinomial as t  
t1 = t.Trinomial(1,-5,6);  
print("zeros are %.2f and %.2f" % t1.getZeros());  
print("The degree was %d" % t1.getDegree());
```

The trinomial class specifically refers to a quadratic. The definition could have gone as follows:

Polynomial

- Monomial
- Binomial
- Trinomial
- Quadratics

But the question becomes whether every quadratic is a trinomial? Can a quadratic be a binomial or even a monomial? Proper design of inheritance takes much time and effort. An improper design could set up an infrastructure that creates many headaches down the road!

Notes: Abstraction and Encapsulation

Overview

Consider a television set. The inner workings of the television set are hidden from the user. Buttons allow the user to turn the television on/off or to change channels. Imagine if the user had to understand all the inner workings of the circuitry prior to use!

Abstraction

Class abstraction is the separation of the class implementation from the use of the class. The creator defines the methods that can be used and how to use them.

Examples:

TV.PowerOn()

TV.PowerOff()

TV.ChangeChannel(channelNumber)

TV.ChangeVolume(volumeNumber)

Our television example would have the methods above.

- To call the method “ChangeChannel,” the user would need to pass in the channelNumber.

- This could also be programmed by increasing or decreasing the channel by one.
- This defines the class implementation (if all the code was placed with the methods).
- The actual usage would be separate.

For example, Nielson states that there are 115.6 televisions in the United States in 2014.

There are 115.6 instances of a television object in the United States.

This is the usage, while the example above would just describe the interface.

The public methods, expected behavior, and usage are referred to as the class's contract with the client.

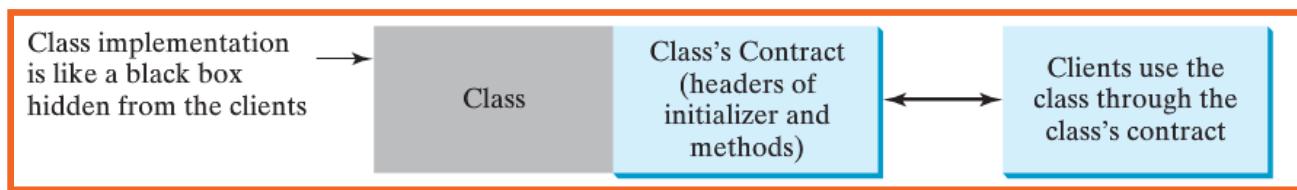
Encapsulation

The details of the implementation are encapsulated (the packing of instance variables and functions into a single component) and hidden from the user. This is referred to as the class encapsulation.

Data Types and Objects

These concepts can also be applied to data types and objects.

- An abstract data type refers to an object that has a very controlled environment, allowing only public methods and hiding the details.
- This separation of the class implementation from the use of the class is referred to as class abstraction. This class abstraction can be seen in the following diagram from the book:



Data Attributes

Data attributes may be referenced by methods, as well as by ordinary users (“clients”) of an object.

- Classes are not usable to implement pure abstract data types.
- Nothing in Python makes it possible to enforce data hiding — it is all based on convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Notes: Information Hiding

Overview

Part of this abstraction/encapsulation is separating the implementation from the definition. Another part is “hiding” everything but the details of the how to access the object. The television is again the perfect example. The details of how the television works are hidden from the user. Most users would just have to say that “magic” is what makes it work. They might give some kind of weak explanation of electricity and channels coming in through a cable, but that is just what they notice going into the back of the television. This simplicity is what allows the television to be popular. Imagine if the user had to understand the details of how a television works in order to use it. The television would probably cease to exist. Our goal in designing an object is to hide as much as possible from the programmer to simplify the usage of the object!

In object-oriented programming, sometimes it is necessary to hide certain aspects of objects, like instance variables.

Example.

```
classRect:
```

```
def __init__(self,length=1,width=1):  
    self.length = length;  
    self.width = width;  
    self.area = length * width;  
    self.perimeter = 2 * length + 2 * width;
```

Note: The instance variable (self.length) is accessed directly.

```
defgetLength(self):
```

```
returnself.length;
```

```
defsetLength(self,param_l):
```

```
self.length = param_l;
```

```
self.area = self.length * self.width;
```

```
self.perimeter = 2 * self.length + 2 * self.width;
```

```
defgetArea(self):
```

```
returnself.area;
```

```
defgetPerimeter(self):
```

```
returnself.perimeter;
```

Class definition Example:

```
import Rectangle as r  
  
r1 = r.Rect(3,5);  
print("area =",r1.getArea());  
  
r1.length = 10;  
print("area =",r1.getArea());  
print("length =",r1.length);
```

In reality, we would only like instance variables to be accessible via “get” and “set” methods. Otherwise, it is possible to directly change the length, but area and perimeter do not get updated! The designer of the object would like to make sure that the object does not have the integrity violated. This can be accomplished with “information hiding.”

Obviously, it is not desired that the programmer access the instance variables directly, but should use get and set methods to change the values. In this manner, the designer of the object can make sure that the object functions as desired. Two underscores before the name of the instance variables and/or methods will make them private, so they can only be accessed within the object and not by programs that have created an instance of the object!

This is misleading. You can still see them from the Python program that implements the object, but they cannot be changed and that is the important component.

There are different ways to actually hide the instance variables.

One is shown in the demonstration – usually, it is not worth the effort.

Python programmers know that “hidden” instance variables should not be accessed directly but only through the “get” and “set” methods.

Notes: Mean and Median

Overview

The main objective of this lesson is to analyze a formula and think in terms of algorithms.

Sample Mean

$$\bar{x} = \frac{\Sigma x}{n}$$

Uppercase sigma means to sum up the values. Summing brings up more questions before this can be programmed.

Questions about Summing:

Is it a finite set of numbers that will not change?

No: Use lists and looping

Yes: Is the count of numbers (n) greater than five?

Yes: Use lists and looping

No: Use different variable names (num1, num2, etc.)

In our case, since the code is intended for a library, lists should hold the numbers!

The uppercase sigma implies looping. For and while loops will accomplish the same task in many cases (but not all). While loops are best when the programmer

does not know the exact count of numbers (n). For loops introduce fewer errors in a program compared to while loops, which give the programmer complete control. For the library that is going to be created, the “for” loop will be used due to the simplicity and reduction of errors.

Consider a simple example. The programmer does not have to do the complete example but just enough to understand the logic of the algorithm.

Example

Find the mean of 3, 8, 16.

$$\Sigma x = 3 + 8 + 16 = 27$$

(This will be accomplished by having a sum variable that is accumulated while the list is traversed with a for loop.)

After that piece of code, the mean is a simple calculation that is returned to the user.

Programming Considerations

No matter what programming language is used, the programmer should ask basic questions (that vary depending on the formula/algorithm/procedure):

1. Numbers need to be added; is there a sum function or method in the language?
2. Is there a library that has a sum function that I can use?
3. Do I put this in a library or an object?

These are items that should be considered no matter what programming languages are used.

Median

If n is odd, the median is the number exactly in the middle.

If n is even, the median is the average of the middle two numbers.

Note: list of numbers must be sorted!

Again our library will deal with a series of numbers. so these will be kept in a list so that our library can be used in many situations. For the library, the sort method of lists will be used to perform the sorting. It would be overkill for our example to use some sorting technique such as bubble sort, quick sort, etc.

Example 1

data: 1, 5, 7, 9, 21

there is an odd number of data values which can be determined by the following code:

$n \% 2 == 0$ means even count of data values

$n \% 2 == 1$ means odd count of data values

In order to determine the middle number in this setup, we would use the following calculation:

$n / 2$ with the decimal truncated. For our example $5 / 2 = 2.5$ or 2 (remember the indexing starts at 0 so this would give us 7). If the programming language rounds once an integer is converted to a float, a different formula might be necessary.

Example 2

data: 1, 5, 7, 9

Much of the code above still works but with $n / 2$ would give $4 / 2 = 2$, which would be the 7 (remember indexing starts at 0). So we would need that position and the one before it. Once we have the 5 and 7 (the middle two numbers), we can add them and divide by two.

Programming Considerations

1. Numbers need to be sorted; is there a sort function? If not, do I use a simple bubble sort or does efficiency need to be considered?
2. In some cases, the average needs to be calculated. Can I use the mean function that I already created?
3. Do I put this in a library or an object?

Sample standard deviation

$$s = \sqrt{\frac{\sum(x - \bar{x})^2}{n-1}}$$

Our same considerations hold true to a dataset coming into the program and lists will again be used. The sample mean will call the function that was already created above. A for loop will again be used to accumulate. The square root function will need to be used for this algorithm.

Programming Considerations

1. Numbers need to be added; is there a sum function or method in the language?
2. Is there a library that has a sum function that I can use?
3. The sample mean is needed; did I design my code for the mean as such that I can easily use it?
4. Is there a function for square root?
5. Does the natural rounding of data types matter?
6. Do I put this in a library or an object?

While a simple example can be done, it is obvious that we will go through the x values in the dataset, calculate the mean and then go back through the x values calculating $(x - \bar{x})^2$ and summing the result.

Standard Variance

$$s^2 = \frac{\sum(x - \bar{x})^2}{n-1}$$

This formula is fairly simple to implement if the sample standard deviation function is working. The data values are stored in a list and then passed into the sample standard deviation function, then the result is squared and returned to the user.

Programming Considerations

1. Numbers need to be added; is there a sum function or method in the language?
2. Is there a library that has a sum function that I can use?
3. The sample mean is needed, did I design my code for the mean as such that I can easily use it?
4. Is there a function for square root?
5. Does the natural rounding of data types matter?
6. Do I put this in a library or an object?

Notes: Standard Deviation and Variance

Standard Deviation

Standard Deviation is a measurement of how spread out numbers are. Variance combines all the numbers in a dataset to produce a measure of spread. Standard deviation is often used in statistics programming to draw inferences about a population from a sample.

The Standard Deviation formula is the square root of the Variance. Variance is the average of the squared difference from the Mean. The variance is computed as the average squared deviation of each number from its mean. All of this means that, if you don't have the "Mean" right, nothing else is going to add up either.

Remember that mean, median, and mode are measures of central tendency. These measures will give you information on the data values of the center of the dataset.

Variance (S^2) = average squared deviation of values from mean

Standard deviation (S) = square root of the variance

Think it through!

Points to remember when using standard deviation or variance in your program:

- Standard deviation is never negative
- Standard deviation is always a positive number because it is always measured in the same units as the original data

- If all of the data in a set is the same value, the standard deviation is zero because each value equals the mean
- Standard deviation is only used to measure dispersion around the mean of a dataset
- Variance is never negative because the variance sum is squared, thus, either positive or zero
- Variance has squared units

These functions calculate a measure of how much the population or sample tends to deviate from the typical or average values.

Functions	Measures of Spread
pstdev()	Population standard deviation of data.
pvariance()	Population variance of data.
stdev()	Sample standard deviation of data.
variance()	Sample variance of data.

Don't recreate the wheel!

When you need to add numbers, check to see if there is a sum function or method already within the language to handle it. Review the list of Python's built-in functions.

Built-in Functions				
abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	apply()

delattr()	help()	next()	setattr()	buffer()
dict()	hex()	object()	slice()	coerce()
dir()	id()	oct()	sorted()	intern()

Notes: Frequency and Probability Distributions

Overview

The frequency distribution merely consists of adding how many numbers are in each class so that there is no formula to program.

Factors to consider

1. One key question is whether the various classes are input by the user or generated by the program. Considerations are code reuse and ease of use by the user of the system.
2. For purposes of this example, the user will input the starting lower class limit, class width, number of classes, and then the actual numbers.

There are different methods that can be used to program a frequency distribution. The data values will be stored in a list for the same reasons above. The lower-class limits (LCL) and upper-class limits (UCL) will be built within a for loop, using the following formulas:

first LCL = LCL input by user

first UCL = LCL + Class Width – 1 (assuming a gap of 1)

After the first one, the next LCL and UCL limits will be calculated using:

LCL = previous LCL + Class Width

UCL = previous UCL + Class Width

After each class is built, then the list of data values will be looped through seeing how many are \geq LCL and \leq UCL.

Programming considerations

1. Although a fairly simple program, does a library already exist that performs the function?
2. Are the numbers going to be input or read in from a file?
3. Do I put this in a library or an object?

Binomial Probability Distribution (for $x = 0, 1, 2, \dots, n$)

$$P(x) = \frac{n!}{(n-x)!x!} \cdot p^x (1-p)^{n-x}$$

Two functions will be created for this distribution:

1. The first one will calculate the probability of exactly x successes.
2. The second one will use a for loop to accumulate all the probabilities from $x=0,1,\dots,n$.

Programming Considerations

1. Numbers need to be added; is there a sum function or method in the language?
2. Is there a library that has a sum function that I can use?
3. Is there a factorial function or does this need to be programmed?
4. Does the natural rounding of data types matter?
5. Do I put this in a library or an object?

Realize that the questions that arise based upon different algorithms are not all valid! For example, a sum function would not benefit the program at all in the binomial probability distribution. But these are questions that should mentally be answered prior to programming.

Notes: Probability Simulations

Probability Simulations

This will not be implemented in a library since every probability simulation is different. A very basic example, given in a statistics course, is the birth of three children and determining the probability of getting x number of boys out of the three children.

Normally a simple example is a very straightforward method to figure out the Python program. This is one example that does not allow for a simple example. A better method will be just to program it and display the tables of values with a general overview. The important part of this program is to figure out how to get random numbers. The program should return either 1 or 0 randomly. The 1 will indicate that the gender of the child is a boy, while a 0 indicates that the gender of the child is a girl. A for loop will simulate the number of trials necessary. Three children will be in each trial and the sum of the random numbers will indicate how many boys there were in those three children.

For example, consider that a trial returns:

0 1 1

This would indicate that the first child was a girl (0), the second child was a boy (1), and the third child was a boy (1). Now adding these numbers: $0 + 1 + 1 = 2$, which would mean 2 of the three children were boys!

Programming Considerations

1. Every programming language has a random feature; how does it work in Python? Do you need to provide a seed? Is it contained in a library?
2. There is implied summing, which works best with either a for or while loop.
3. Each simulation is different, so a library or objects probably does not make sense unless more analysis needs to be done on the simulated data.

Sample linear correlation coefficient

$$r = \frac{\sum \left(\frac{x_1 - \bar{x}}{s_x} \right) \left(\frac{y_1 - \bar{y}}{s_y} \right)}{n - 1}$$

From previous functions, the sample mean and sample standard deviation have already been placed into a function in our library. This formula will require two sets of data that will be placed into lists. The sample mean and sample standard deviation of both lists will be obtained and then a for loop will be used to calculate the numerator of the above formula.

Programming Considerations

1. Numbers need to be added; is there a sum function or method in the language?
2. Is there a library that has a sum function that I can use?
3. The sample mean and sample standard deviation is needed. Did I design my code for the mean and standard deviation so that I can easily use it?
4. Does the natural rounding of data types matter?

5. Do I put this in a library or an object?

Least-squares regression line (or equation)

$$\hat{y} = b_1x + b_0$$

where

$$b_1 = r \cdot \frac{s_y}{s_x}$$

(slope)

(y – intercept)

This will be fairly simple to implement since the correlation coefficient (r), the sample mean, and the sample standard deviation functions have already been written in the library. This will just do basic formulas and return the results.

Programming Considerations

1. The sample mean, a sample standard deviation and a linear correlation coefficient is needed; did I design my previous code so that I can easily use it?
2. Does the natural rounding of data types matter?
3. Do I put this in a library or an object?

Confidence Interval for a Population Proportion

$$\hat{p} - z_{\alpha/2} \cdot \sqrt{\frac{\hat{p}(1 - \hat{p})}{n}}$$

Lower bound:

Upper bound: $\hat{p} + z_{\alpha/2} \cdot \sqrt{\frac{\hat{p}(1 - \hat{p})}{n}}$

This formula is straightforward with $\hat{p} = \frac{x}{n}$. The only tricky part is the critical value. This will be looked up in the following table:

Critical Values	
Confidence Level	Critical Value (Z-score)
0.90	1.645
0.91	1.70
0.93	1.81
0.94	1.88
0.95	1.96
0.96	2.05
0.97	2.17
0.98	2.33
0.99	2.575

There are many options for programming this lookup. This library will use the dictionary component of Python to provide a simple and easy lookup scheme.

Programming Considerations

1. Do the critical values need to be hardcoded into a table or derived by formula or calculus?
2. Does the natural rounding of data types matter?
3. Do I put this in a library or an object?

Hypothesis Testing (Claims)

Our library will calculate only one component of one type of claim. Consider the following steps in testing claims about population proportions:

Classical Approach (By Hand)

1. Write down a shortened version of claim.
2. Come up with a null and alternate hypothesis (H_0 always has the equals part).
3. See if claim matches H_0 or H_1 .
4. Draw the picture and split α into tails:

$H_1: p \neq \text{value}$ Two Tail

$H_1: p < \text{value}$ Left Tail

$H_1: p > \text{value}$ Right Tail

5. Find critical values: Use Standard Normal Distribution table.
6. Find test statistic: $Z_0 = \frac{\hat{p} - p_0}{\sqrt{\frac{p_0(1-p_0)}{n}}}$ where $\hat{p} = \frac{x}{n}$ and p_0 is the number from the claim (in decimal form).
7. If test statistic falls in the tail, Reject H_0 . If test statistic falls in the main body, Accept H_0 . If test statistic falls in the main body, Accept H_0 .
Determine the claim based on step 3.

The test statistic will be programmed. This is a fairly simple program with $\hat{p} = \frac{x}{n}$.

The rest of the information is to be brought in as a parameter.

Programming Considerations

1. Usually, an entire claim is not done in a library, but the library provides various components such as the critical value, test statistic, or p-value, depending upon the approach
2. Do I put this in a library or an object?

Notes: Correlation Coefficient, Linear Regression Equation, and Sum of the Squared Residuals

Linear Correlation Coefficient

Correlation coefficients measure the strength of the association between two variables. In a data sample of the two variables, their covariance is divided by the product of their individual standard deviations. It is a standardized measurement of how the two are linearly associated. Pearson's product-moment correlation coefficient is the most common correlation coefficient and it measures the strength of the linear association between variables.

A Sample refers to a set of observations drawn from a population in statistics. Samples are used during research because it is not practical to study the entire population. For example, if you wanted to know the weight of every three-year-old in America, it just wouldn't be possible to find and weigh all of them.

Correlation Coefficient	
Overview: The sign and the absolute value describe the direction and the magnitude of the relationship between two variables.	
The value	ranges between -1 and 1
The greater the absolute value	the stronger the linear relationship
The strongest linear relationship is indicated	by -1 or 1
The weakest linear relationship is indicated	by equal to 0
A positive correlation	if one variable gets bigger, the other variable tends to get bigger
A negative correlation	if one variable gets bigger, the other variable tends to get smaller

Pearson product-moment correlation coefficient only measures linear relationships.

Correlation	Formulas	Use
Pearson product-moment correlation	$r = \Sigma (xy) / \sqrt{[\Sigma x^2] * [\Sigma y^2]}$	For computing a product-moment correlation coefficient (r)
Linear correlation (sample data)	$r = [1 / (n - 1)] * \sum \{ [(x_i - \bar{x}) / s_x] * [(y_i - \bar{y}) / s_y] \}$	Sample means and sample standard deviations to compute a correlation coefficient (r) from sample data
Linear correlation (population data)	$\rho = [1 / N] * \sum \{ [(X_i - \mu_X) / \sigma_X] * [(Y_i - \mu_Y) / \sigma_Y] \}$	Uses population means and population standard deviations to compute a population

		correlation coefficient (ρ) from population data
--	--	---

Linear Regression Equation

- Linear regression is most often used for any level of data exploration but, if you are looking at the relationship between two variables, chances are you will want to conjure a regression.

Simple Linear Regression	Formulas	Meanings
Simple linear regression line	$\hat{y} = b_0 + b_1x$	b_0 is the intercept constant in a sample regression line
Regression coefficient	$= b_1 = \sum [(x_i - \bar{x})(y_i - \bar{y})] / \sum [(x_i - \bar{x})^2]$	b_1 refers to the regression coefficient in a sample regression line (i.e. the slope)
Regression slope intercept	$= b_0 = \bar{y} - b_1 * \bar{x}$	

Simple Linear Regression	Formulas	Meanings
Regression coefficient	$= b1 = r * (sy / sx)$	
Standard error of regression slope	$= sb1 = \sqrt{[\sum(y_i - \hat{y}_i)^2 / (n - 2)] / \sum(x_i - \bar{x})^2}$	sb1 refers to the standard error of the slope of a regression line

Sum of the Squared Residuals

- The sum of squared residuals (SSR) is also known as residual sum of squares (RSS), as well as the sum of squared errors (SSE) of prediction. A lot of different names to mean the same thing; the amount of the difference between data and an estimation model.

$$RSS = \sum_{i=1}^n (y_i - f(x_i))^2$$

- X,Y – set of values
- α, β – constant values
- n – set value counts

Notes: Confidence Intervals

Confidence Intervals

Statisticians interpreting various results from a set of data need to know how sure they are while dealing with the data. Within statistics, confidence level is used to describe the amount of uncertainty associated with a simple estimate of a population parameter. Simply put, confidence interval is a range within which most plausible values would occur.

If $n \geq 30$, then

$$\text{Confidence interval} = x \pm z_{\frac{\alpha}{2}} \left(\frac{\sigma}{\sqrt{n}} \right) \text{ and}$$

If $n < 30$, then

$$\text{Confidence interval} = x \pm t_{\frac{\alpha}{2}} \left(\frac{\sigma}{\sqrt{n}} \right)$$

Where

n = Number of terms

x = Sample Mean

σ = Standard Deviation

$z_{\frac{\alpha}{2}}$ = Value corresponding to $\frac{\alpha}{2}$ in z table

$t_{\frac{\alpha}{2}}$ = Value corresponding to $\frac{\alpha}{2}$ in t table

$$\alpha = 1 - \frac{\text{confidence level}}{100}$$

In order to express a confidence interval, you need three separate data pieces:

- Confidence level
- Statistic
- Margin of error

Given these three pieces of information, you can calculate the range of the confidence interval by the sample statistic + margin of error. The confidence interval is specified by the uncertainty associated with the confidence level. In most cases, the margin of error is not given but must be calculated separately.

Estimation

- Confidence interval: Sample statistic + Critical value * Standard error of statistic
- Margin of error = (Critical value) * (Standard deviation of statistic)
- Margin of error = (Critical value) * (Standard error of statistic)

When dealing with confidence intervals, the margin of error is the range of values above and below the sample statistic. The critical value is a factor used to calculate the margin error.

NumPy and SciPy

NumPy is the fundamental package for scientific computing in Python. NumPy is an extension module that provides efficient operation on arrays of homogeneous data. NumPy has built-in functions for creating arrays from scratch.

SciPy is a set of open source (BSD licensed) scientific and numerical tools, used by scientists, analysts, and engineers in Python. SciPy has several different modules used for optimization, integration, linear algebra, interpolation, FFT, signal and image processing, ODE solvers, special functions, and others.

NumPy and SciPy are open-source add-on modules to Python. Between them they provide common numerical and mathematical routines in pre-compiled, fast functions. Given the fact they are open-source, they are continually maturing.

The NumPy (Numeric Python) package offers basic routines for manipulating large arrays and matrices of numeric data. The SciPy (Scientific Python) package extends the functionality of NumPy by adding a collection of useful algorithms, like minimization, Fourier transformation, regression, and other applied mathematical techniques.

Visit www.scipy.org/ and www.numpy.org/ for full descriptions of what is available.

Mean Confidence Interval in Python

Using NumPy and SciPy example:

```
#!/usr/bin/env python

from scipy.stats import t
from numpy import average, std
from math import sqrt

if __name__ == '__main__':
    # data we want to evaluate: average height of 30 one year old male and
    # female toddlers. Interestingly, at this age height is not bimodal yet
    data = [63.5, 81.3, 88.9, 63.5, 76.2, 67.3, 66.0, 64.8, 74.9, 81.3, 76.2,
            72.4, 76.2, 81.3, 71.1, 80.0, 73.7, 74.9, 76.2, 86.4, 73.7, 81.3,
            68.6, 71.1, 83.8, 71.1, 68.6, 81.3, 73.7, 74.9]
    mean = average(data)
    # evaluate sample variance by setting delta degrees of freedom (ddof) to
    # 1. The degree used in calculations is N - ddof
    stddev = std(data, ddof=1)
    # Get the endpoints of the range that contains 95% of the distribution
    t_bounds = t.interval(0.95, len(data) - 1)
    # sum mean to the confidence interval
    ci = [mean + critval * stddev / sqrt(len(data)) for critval in t_bounds]
    print "Mean: %f" % mean
    print "Confidence Interval 95%%: %f, %f" % (ci[0], ci[1])
```

Notes: Hypothesis Testing

Overview of Hypothesis Testing

Hypothesis testing is one of the most often used and abused practices in statistics. Formally, the hypotheses are specified, a α -level is chosen, a test statistic is calculated, and it is reported as to whether H_0 or H_1 is accepted.

A statistical hypothesis is an assumption about a population parameter. The assumption may be true or false. Hypothesis testing is a type of formal procedure used to accept or reject the statistical hypothesis.

Hypothesis testing is also called significance testing. It tests the claim about a parameter using evidence data in a sample.

1. Null and alternative hypotheses

- Null hypothesis – Statement regarding the value(s) of unknown parameter(s)(will always contain an equality), Represented by H_0
- Alternative hypothesis – Statement contradictory to the null hypothesis (will always contain an inequality), Represented by H_1

2. Test statistic

- Quantity based on sample data and null hypothesis used to test between null and alternative hypotheses

3. P-value and interpretation

4. Significance level (optional)

Rejection region – Values of the test statistic for which we reject the null in favor of the alternative hypothesis.

Hypothesis Testing	Formulas
Standardized test statistic	= (Statistic – Parameter) / (Standard deviation of statistic)
One-sample z-test for proportions	$z\text{-score} = z = (p - P_0) / \sqrt{p * q / n}$
Two-sample z-test for proportions	$z\text{-score} = z = [(p_1 - p_2) - d] / SE$
One-sample t-test for means	$t\text{-score} = t = (x - \mu) / SE$
Two-sample t-tests for means	$t\text{-score} = t = [(x_1 - x_2) - d] / SE$
Matched-sample t-test for means: t-score	$t = [(x_1 - x_2) - D] / SE = (d - D) / SE$
Chi-square test statistic	$\chi^2 = \sum [(Observed - Expected)^2 / Expected]$

Hypothesis testing is one of the two common forms of statistical inference.

Terms	
Population	all possible values
Sample	a portion of the population
Statistical inference	generalizing from a sample to a population with a calculated degree of certainty
Two forms of statistical inference	Hypothesis testing Estimation
Parameter	a characteristic of a population, e.g. population mean μ
Statistic	calculated from data in the sample, e.g. sample mean ()

Hypothesis Testing		
Test Result –	H_0 True	H_0 False
True State		
H_0 True	Correct Decision	Type I Error
H_0 False	Type II Error	Correct Decision

$\alpha = P(\text{Type I Error})$ $\beta = P(\text{Type II Error})$

- Goal: Keep α, β reasonably small

- α is the probability of Type I error
- β is the probability of Type II error

You can change the α -level for a hypothesis test. This is called the level of significance and changing it can affect the results of the test, whether or not you reject or fail to reject H_0 .

Standard Normal Distribution

A normal distribution is a probability distribution that associates the normal random variable X with a cumulative probability. The standard normal distribution is a special case of the normal distribution. This distribution occurs when a normal random variable has a mean of zero and the standard deviation is one.