



Winter Working Connections 2020

Python for Data Science

Day 1

Module 1

Python Statements, Data Types, and Variables

[1]



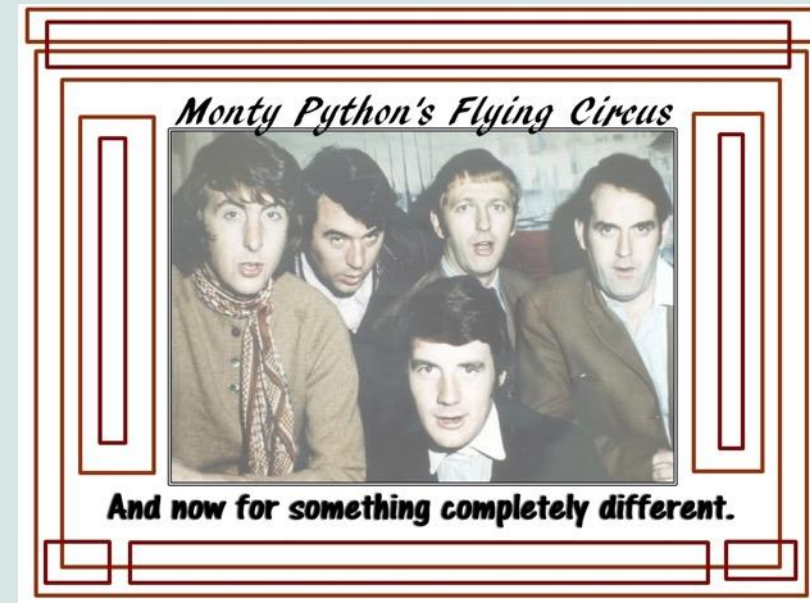
Ch. 1 Getting Started

[2]

The Python Programming Language



- Why is it called Python?
- When he began implementing Python, Guido van Rossum was also reading the published scripts from “Monty Python’s Flying Circus”, a BBC comedy series from the 1970s. Van Rossum thought he needed a name that was short, unique, and slightly mysterious, so he decided to call the language Python.





Characteristics of the Python Language

- Simple syntax
- Many features of traditional programming languages.
- Open source
- Support for different types of applications
 - Console (text-based)
 - scripting and small utilities
 - not necessarily interactive
 - GUI (Graphical User Interface)
 - Web applications



Characteristics of the Python Language

- Python can be used as a scripting language or compiled to bytecode for larger projects
- Python supports dynamic data types and dynamic type-checking
- Python provides automatic garbage collection
- Python can be integrated with other languages such as C, C++, and Java



Python Resources

- Main website:
 - www.python.org/
- Books, Websites, and Tutorials:
 - wiki.python.org/moin/BeginnersGuide/Programmers
- Hitchhiker's Guide to Python:
 - docs.python-guide.org/en/latest/intro/learning/
- Downloading the software:
 - www.python.org/downloads/



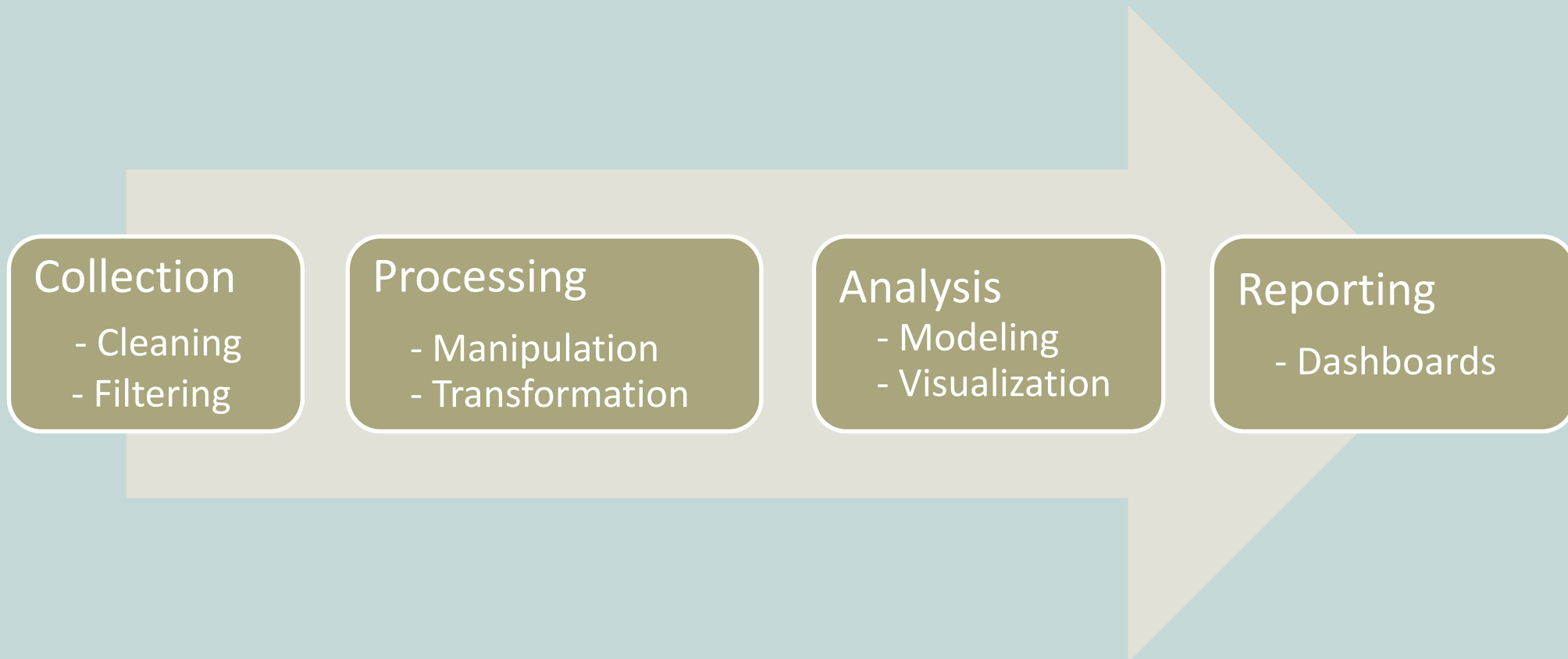
Applications of Python

- Scripting
- Data science
- Machine learning
- Scientific programming
- Game development
- GUI development
- Web application development
- Mobile development



Python and the Data Lifecycle

- The data lifecycle represents stages of data as it progresses through collection and extraction of useful information



Components of the Data Lifecycle in Python



- Python's broad application scope and community-contributed libraries makes it an ideal candidate for implementing components of the data lifecycle
- **Data Collection**
 - Python can access remote data sources, both structured and unstructured, and efficiently collect and clean it for processing
 - Cleaning tasks include fixing malformed data, removing empty entries
 - Some packages designed for this purpose:
 - **Beautiful Soup**
 - **Selenium**
 - **Lxml**
 - **Scrapy**
 - **Requests**
 - **PyODBC**

Components of the Data Lifecycle in Python



- **Data Processing**

- Data processing includes manipulating and transforming data
 - Manipulation tasks include sorting, formatting
 - Transformation tasks include enhancing (e.g. add timestamps), converting from one format to another (e.g. Excel table to XML)
- Many processing packages also serve as analysis tools
- Some packages designed for this purpose:

- **Numpy**
- **Pandas**

- **Query Abstraction Layer**

Components of the Data Lifecycle in Python



- **Data Analysis**
- Packages are provided by the Python community for performing data analysis
 - Modeling
 - Visualization
 - Some of these are used for data processing as well
 - Some packages designed for this purpose:
 - **Numpy**
 - **SciPy**
 - **Matplotlib**

Components of the Data Lifecycle in Python



- **Reporting**

- Reporting includes publishing visuals, tables, and analysis results, frequently as components of a dashboard or web page
 - Some reporting tools are also used for data analysis
 - Some packages designed for reporting:

- **Plotly**

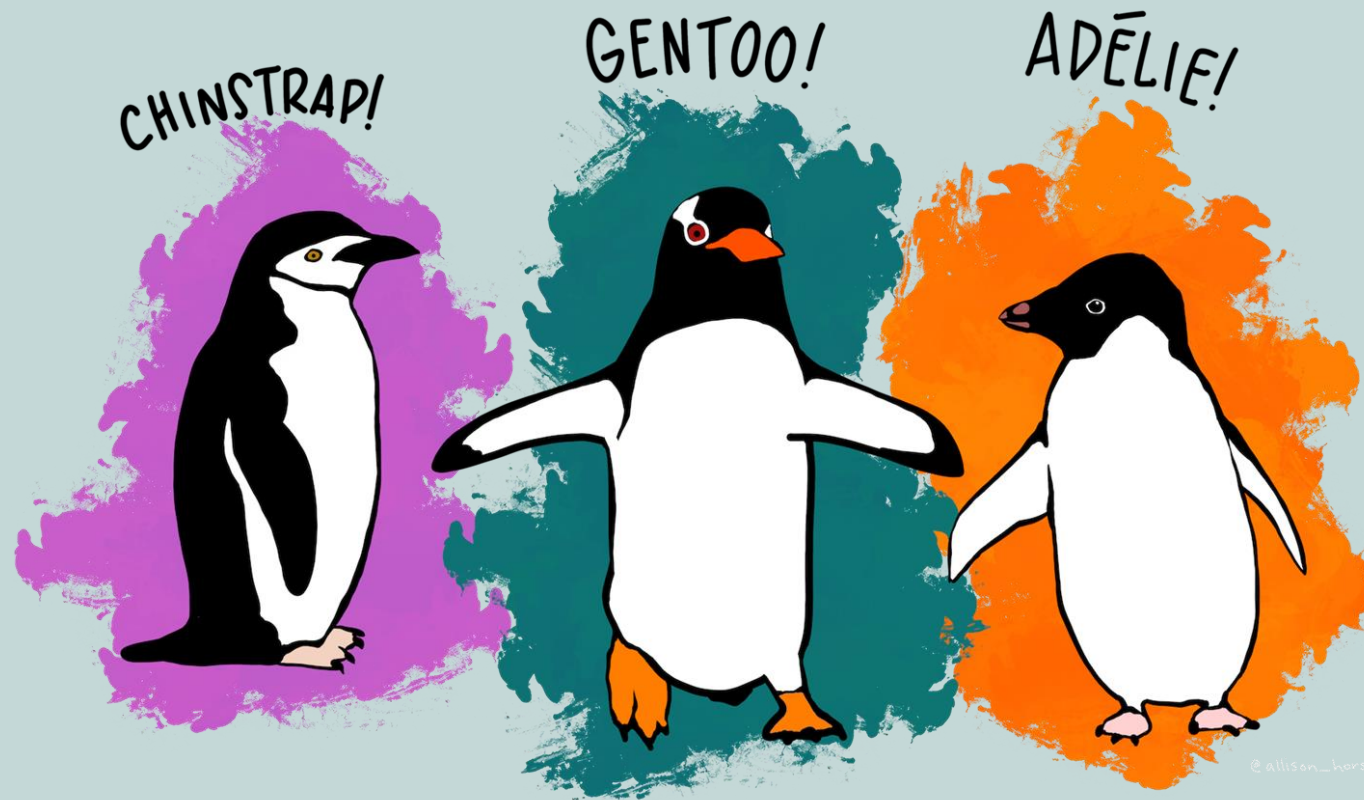
- **Dash**

- **Matplotlib**

Data For This Course: Palmer Penguins



- Some of the content for this course will include Palmer Penguin data



<https://allisonhorst.github.io/palmerpenguins/>

Popular (and Historic) Datasets Used in Data Science



MNIST (1995)



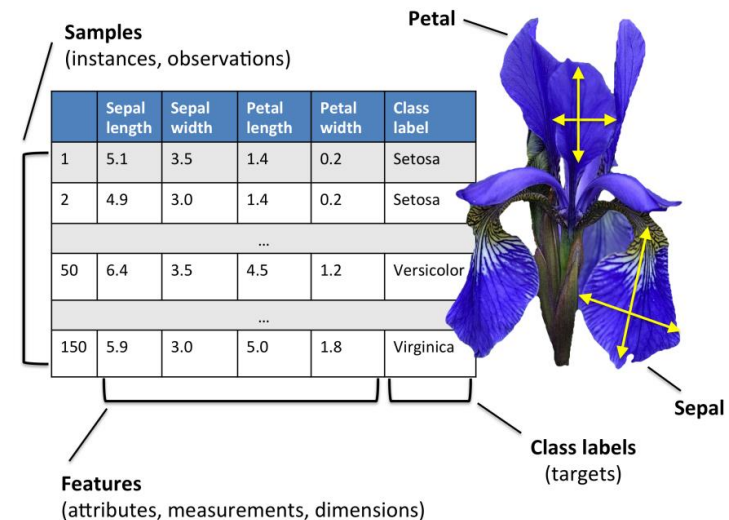
Fashion MNIST (2017)



MTCARS - 32 cars from the 1973-74 model year. Statistics about 1974 cars: fuel consumption, design, and performance.



IRIS: Fisher's Iris data set was introduced by the British statistician, eugenicist, and biologist Ronald Fisher in a 1936 paper.



Enter Palmer Penguins!

- Data includes size measurements for three penguin species observed on three islands in the Palmer Archipelago, Antarctica.
- Collected from 2007 - 2009 by Dr. Kristen Gorman, Palmer Station Long Term Ecological Research Program, part of the US Long Term Ecological Research Network.
- Freely available ("No Rights Reserved").
- "The goal of palmerpenguins is to provide a great dataset for data exploration & visualization"





Species

a factor denoting penguin species (Adélie, Chinstrap and Gentoo)

Island

a factor denoting island in Palmer Archipelago, Antarctica (Biscoe, Dream or Torgersen)

CulmenLength_mm

a number denoting bill length (millimeters)

CulmenDepth_mm

a number denoting bill depth (millimeters)

FlipperLength_mm

an integer denoting flipper length (millimeters)

BodyMass_g

an integer denoting body mass (grams)

Sex

a factor denoting penguin sex (female, male)

Year

an integer denoting the study year (2007, 2008, or 2009)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	StudyName	SampleNumber	Species	Region	Island	Stage	ID	ClutchCompletion	EggDate	CulmenLength-mm	CulmenDepth-mm	FlipperLength-mm	Bod Mass-g	Sex
2	PAL0708	1	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N1A1	Yes	11/11/2007	39.1	18.7	181	3750	MALE
3	PAL0708	2	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N1A2	Yes	11/11/2007	39.5	17.4	186	3800	FEMALE
4	PAL0708	3	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N2A1	Yes	11/16/2007	40.3	18	195	3250	FEMALE
5	PAL0708	4	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N2A2	Yes	11/16/2007					
6	PAL0708	5	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N3A1	Yes	11/16/2007	36.7	19.3	193	3450	FEMALE
7	PAL0708	6	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N3A2	Yes	11/16/2007	39.3	20.6	190	3650	MALE
8	PAL0708	7	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N4A1	No	11/15/2007	38.9	17.8	181	3625	FEMALE
9	PAL0708	8	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N4A2	No	11/15/2007	39.2	19.6	195	4675	MALE
10	PAL0708	9	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg Stage	N5A1	Yes	11/9/2007	34.1	18.1	193	3475	

A Simple Python Program



```
#!/usr/bin/env python3
```

```
# display a welcome message
```

```
print("Welcome to Python!")
```

```
print()
```

```
# get input from the user
```

```
name = input("Please enter your name: ")
```

```
print("Hello,", name)
```

Welcome to Python!

Please enter your name:

Python User

Hello, Python User

[17]

Python Versions



- As of November 2019, the most recent version of Python is 3.8
- Some systems may contain an installation of Python 2.X
 - This may be to support older programs
 - No harm in leaving it alone
- Be sure you are working with a relatively recent version (3.x)
- Run the following from the command line
`python --version`

```
C:\Users> python --version  
Python 3.7.3
```

- Or run IDLE (see next slide) and note the version on the banner:

A screenshot of a Windows command prompt window titled '*Python 3.7.3 Shell*'. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The command prompt shows the output of the 'python --version' command: 'Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 4)] on win32'.

```
*Python 3.7.3 Shell*  
File Edit Shell Debug Options Window Help  
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019,  
4) ] on win32
```

IDLE interactive shell



- IDLE (Integrated Development and Learning Environment) is an IDE for Python development that is included in the Python installation package
 - Lightweight and easy to use
 - Not ideal for larger projects
 - Adequate for this course
 - Always available (useful for supporting code at external sites)
- There are many useful and powerful Python IDE packages
 - Choice is personal, try them and pick for yourself
 - Sublime Text (used in the current course textbook)
 - Adam
 - PyCharm
 - Microsoft VS Code
 - Spyder
 - Jupyter Notebook

IDLE Interactive Shell



- Commands and programs can be entered and executed interactively

```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("Hello out there!")
Hello out there!
>>> 8 + 5
13
>>> 10 / 3
3.3333333333333335
>>> (8 + 2) * 3
30
>>> print(Hello out there!)
SyntaxError: invalid syntax
>>> x = 5
>>> x + 10
15
>>> X + 15
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    X + 15
NameError: name 'X' is not defined
>>> |
```



- Python is an interpreted language
- The IDLE shell can be used as an interpreter
 - Python code is executed line-by-line instead of compiling the entire program
 - Each code statement can be executed immediately 😊
- The Windows IDLE shell does not have a command to clear the interpreter window; things can get messy 😞
 - To create a clear IDLE shell, exit and restart 😞 😞



- To test a statement, type it at the prompt and press the Enter key. Try typing "x = 5" (minus the quotes)
 - You can also type the name of a variable at the prompt to see what its value is (after typing "x = 5", just type "x")
- Any variables that you create remain active for the current session.
 - As a result, you can use them in statements that you enter later in the same session.
- To retype the previous statement on Windows, press Alt+p.
- To cycle through all of the previous statements, continue pressing Alt+p.



- Enter a command in the shell and examine the results:

```
print("Hello, World!")
```

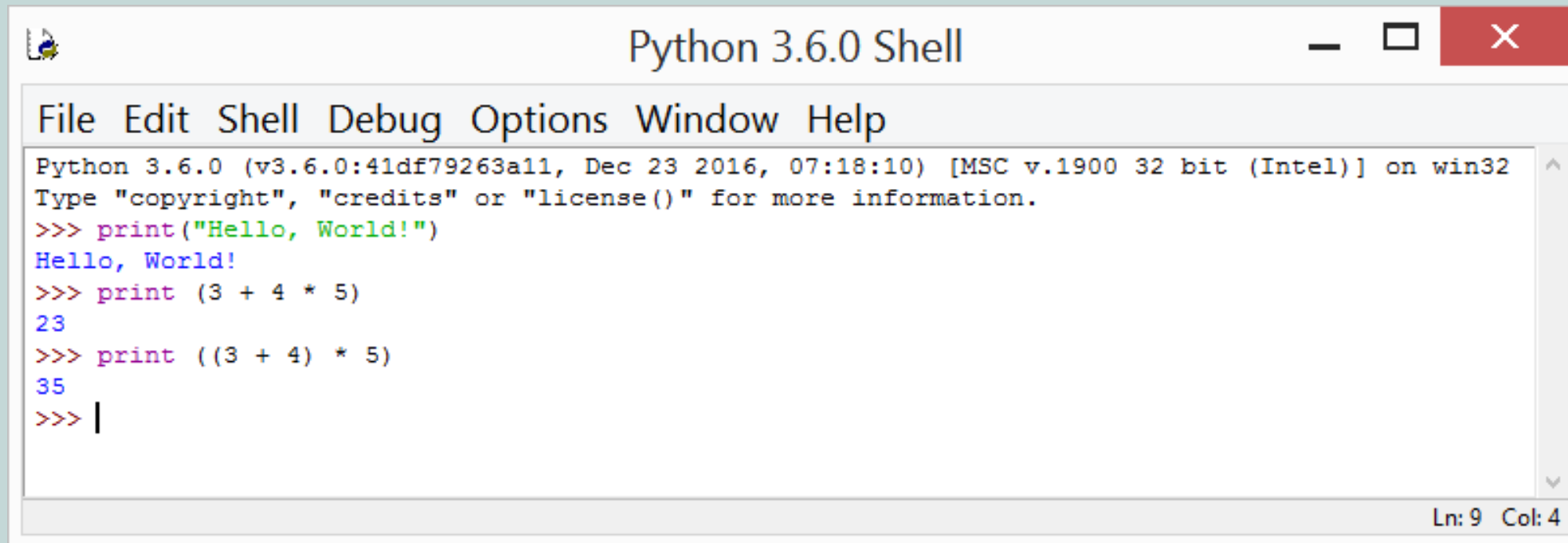
The screenshot shows a window titled "Python 3.6.0 Shell". The window has a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area contains the following text:

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("Hello, World!")
Hello, World!
>>> |
```

The status bar at the bottom right of the window shows "Ln: 5 Col: 4".



```
print(3 + 4 * 5)  
print((3 + 4) * 5)
```



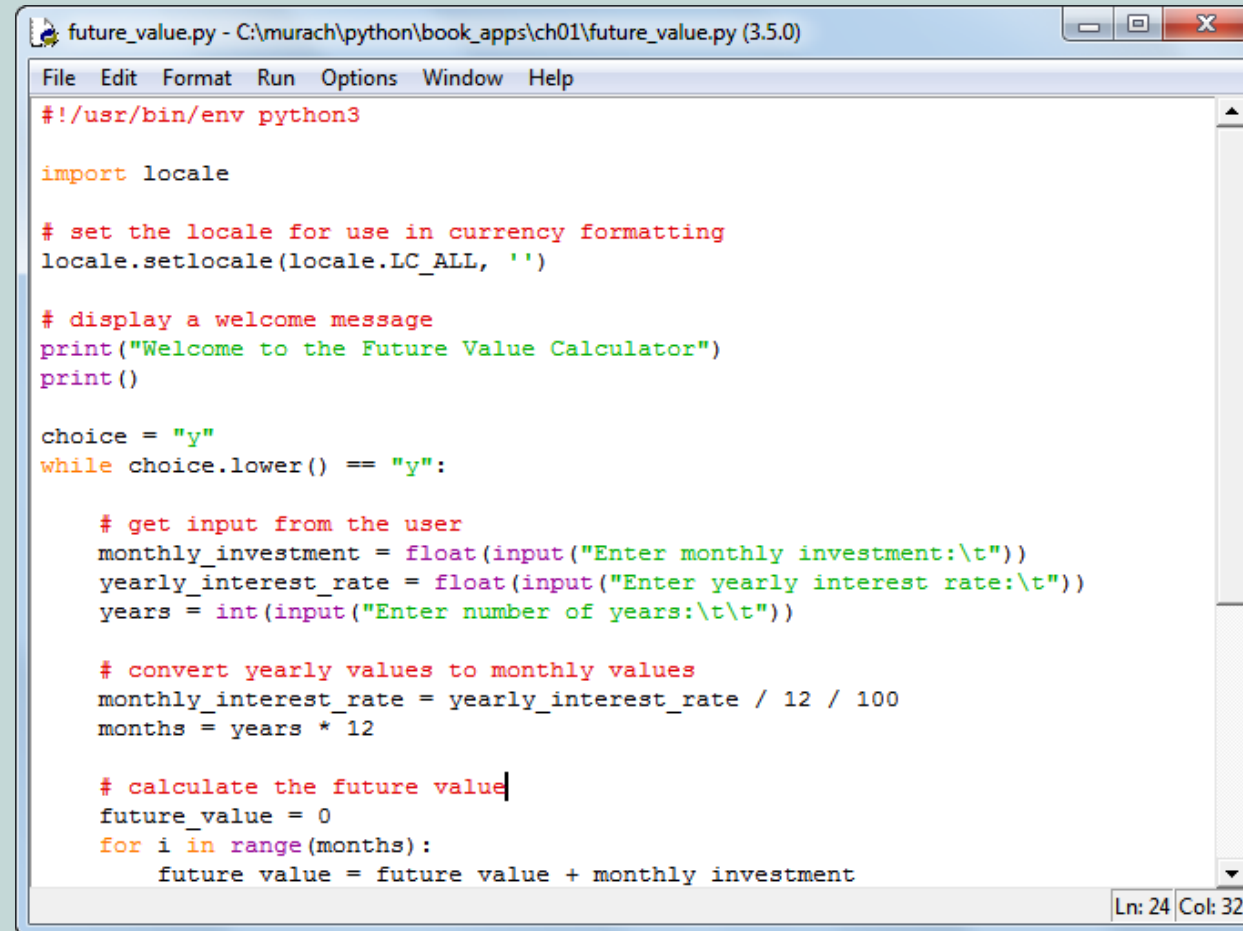
A screenshot of a Python 3.6.0 Shell window. The window has a title bar that says "Python 3.6.0 Shell" and standard Windows window controls (minimize, maximize, close). Below the title bar is a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main area of the window contains the following text:

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) [MSC v.1900 32 bit (Intel)] on win32  
Type "copyright", "credits" or "license()" for more information.  
>>> print("Hello, World!")  
Hello, World!  
>>> print (3 + 4 * 5)  
23  
>>> print ((3 + 4) * 5)  
35  
>>> |
```

At the bottom right of the window, it says "Ln: 9 Col: 4".



IDLE Editor with Source File Displayed



```
future_value.py - C:\murach\python\book_apps\ch01\future_value.py (3.5.0)
File Edit Format Run Options Window Help
#!/usr/bin/env python3

import locale

# set the locale for use in currency formatting
locale.setlocale(locale.LC_ALL, '')

# display a welcome message
print("Welcome to the Future Value Calculator")
print()

choice = "y"
while choice.lower() == "y":

    # get input from the user
    monthly_investment = float(input("Enter monthly investment:\t"))
    yearly_interest_rate = float(input("Enter yearly interest rate:\t"))
    years = int(input("Enter number of years:\t\t"))

    # convert yearly values to monthly values
    monthly_interest_rate = yearly_interest_rate / 12 / 100
    months = years * 12

    # calculate the future value
    future_value = 0
    for i in range(months):
        future_value = future_value + monthly_investment
```

Ln: 24 Col: 32



Running a Source File from IDLE

- From the editor window, press the F5 key or select Run→Run Module.
- If IDLE displays a dialog box that indicates that you must save the program first, click Yes to save it. Then, if the program doesn't have any errors, IDLE runs the program in the interactive shell.

Running a Source File from IDLE



- Select Run / Run Module from Menu Bar

```
future_value.py - C:\murach\python\book_apps\ch01\future_value.py (3.5.0)
File Edit Format Run Options Window Help
#!/usr/bin/env python3

import locale

# set the locale for use in currency formatting
locale.setlocale(locale.LC_ALL, '')

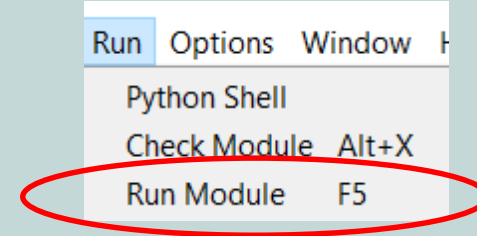
# display a welcome message
print("Welcome to the Future Value Calculator")
print()

choice = "y"
while choice.lower() == "y":

    # get input from the user
    monthly_investment = float(input("Enter monthly investment: "))
    yearly_interest_rate = float(input("Enter yearly interest rate: "))
    years = int(input("Enter number of years: "))

    # convert yearly values to monthly values
    monthly_interest_rate = yearly_interest_rate / 12
    months = years * 12

    # calculate the future value
    future_value = 0
    for i in range(months):
        future_value = future_value + monthly_investment * (1 + monthly_interest_rate / 12)
```



```
*Python 3.5.0 Shell*
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\murach\python\book_apps\ch01\future_value.py =====
Welcome to the Future Value Calculator

Enter monthly investment: 100
Enter yearly interest rate: 12
Enter number of years: 10
Future value: $23,233.91

Continue? (y/n): y

Enter monthly investment: 100
Enter yearly interest rate: 6
Enter number of years: 10
Future value: $16,469.87

Continue? (y/n): |
```



Syntax Errors

- Syntax errors prevent code from building
 - spelling errors, incorrect punctuation, missing parentheses are examples of causes

The screenshot shows a Python IDE window titled "future_value.py - C:\murach\python\book_apps\ch01\future_value.py (3.5.0)". The code in the background is as follows:

```
# get input from the user
monthly_investment = float(input("Enter monthly investment:\t"))
yearly_interest_rate = float(input("Enter yearly interest rate:\t"))
years = int(input("Enter number of years:\t"))

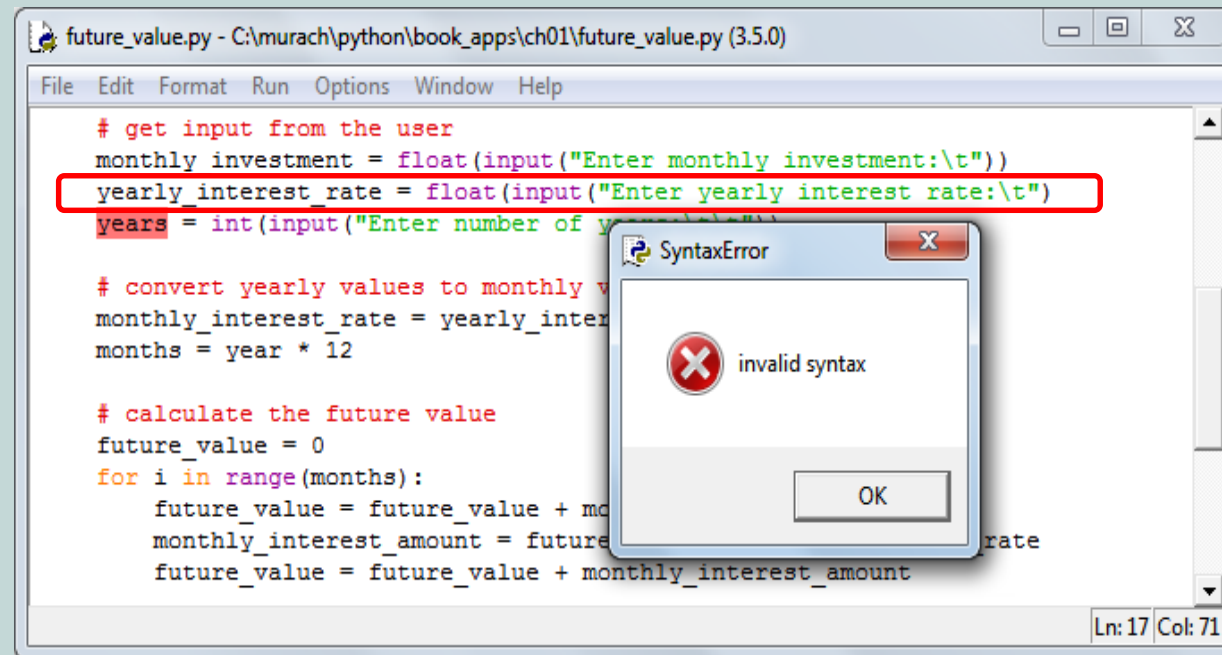
# convert yearly values to monthly values
monthly_interest_rate = yearly_interest_rate / 12
months = years * 12

# calculate the future value
future_value = 0
for i in range(months):
    future_value = future_value + monthly_investment
    monthly_interest_amount = future_value * monthly_interest_rate
    future_value = future_value + monthly_interest_amount
```

A "SyntaxError" dialog box is overlaid on the code, displaying a red "X" icon and the text "invalid syntax". The dialog box has an "OK" button. The status bar at the bottom right of the IDE window shows "Ln: 17 Col: 71".

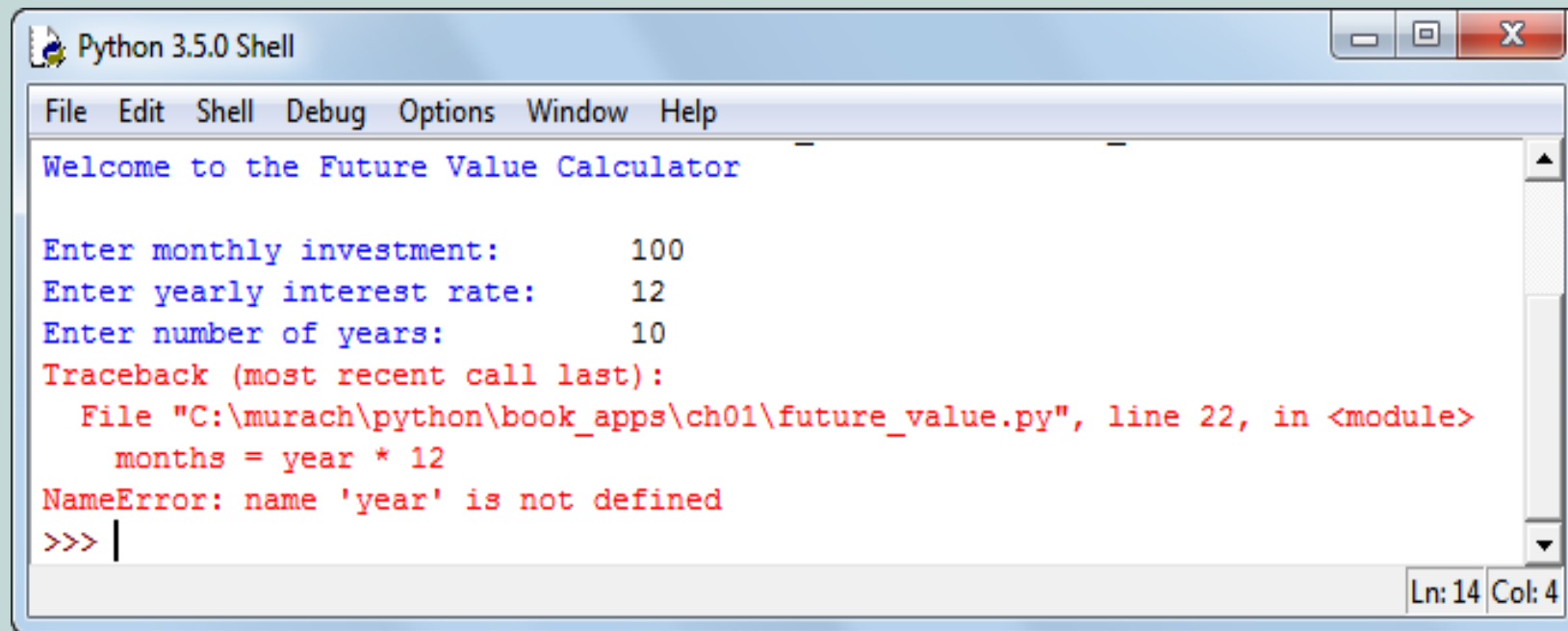
Handling Syntax Errors

- IDLE will highlight where it thinks a syntax error has occurred
- Sometimes the highlighted location will be correct, but sometimes it will be elsewhere (e.g. in the previous statement)



Runtime Errors

- Runtime errors can crash your program
 - There are ways to handle these errors (exceptions) to prevent the crash, but for now we will have to find and fix the code that is causing the crash.



The screenshot shows a Python 3.5.0 Shell window with a menu bar (File, Edit, Shell, Debug, Options, Window, Help). The text inside the window is as follows:

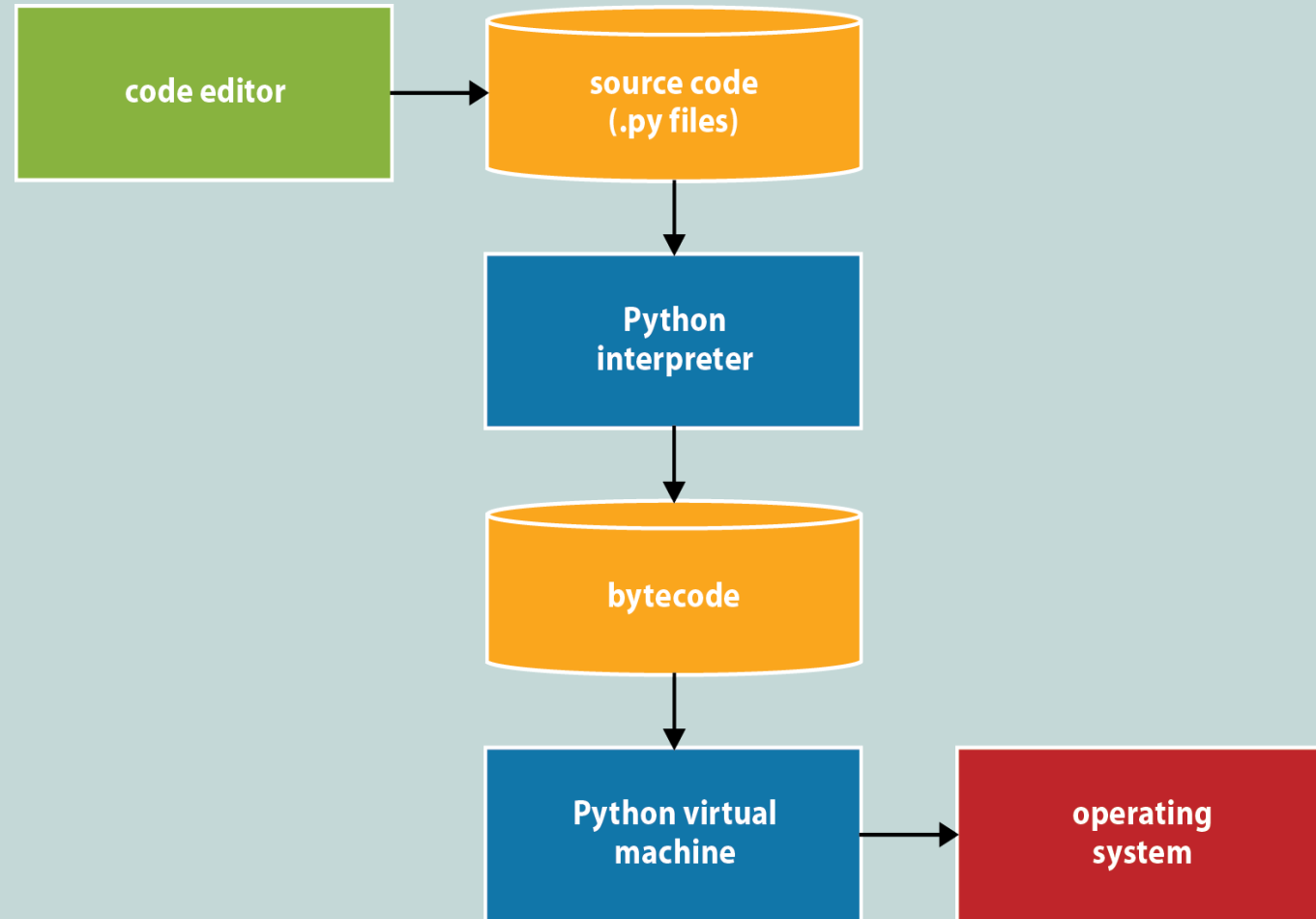
```
Welcome to the Future Value Calculator

Enter monthly investment:      100
Enter yearly interest rate:    12
Enter number of years:        10
Traceback (most recent call last):
  File "C:\murach\python\book_apps\ch01\future_value.py", line 22, in <module>
    months = year * 12
NameError: name 'year' is not defined
>>> |
```

The status bar at the bottom right indicates "Ln: 14 Col: 4".



How Python compiles and runs source code



Python's implementation is "byte-code interpreted".

- Step 1. The programmer uses a text editor or IDE to enter and edit the source code. Then, the programmer saves the source code to a file with a .py extension.
- Step 2. The source code is compiled by the Python interpreter into bytecode.
- Step 3. The bytecode is translated by the Python virtual machine into instructions that can interact with the operating system of the computer.

Python Statements



- A line of Python code is known as a statement

```
print("Total score = " + str(score_total))
```

- Indentation matters! Some statements must be indented, others must not be. Typical indentation is 4 spaces:

```
if test_score >= 0:
```

```
    → score_total = score_total + test_score
```


Python Coding Style (1/2)



- A PEP is a Python Enhancement Proposal, which provides technical specifications for various language features and conventions and should be considered authoritative.
- [PEP 8 -- Style Guide for Python Code](#)
- Use 4-space indentation, and no tabs.
- 4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.
- Wrap lines so that they don't exceed 79 characters. This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.



Python Coding Style 2/2



- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4)`.
- Name your classes, variables, and functions consistently; the convention is to use PascalCase for classes and `lower_case_with_underscores` or camelCase for variables, functions and methods. Always use `self` as the name for the first method argument (classes and methods in a later module).



The "shebang" Line

`#!/usr/bin/env python3`

- Tells some operating systems (e.g. Unix) how to run the program
 - can indicate which version of python to use (e.g. 3)
 - Not required for Windows
 - Not required when using IDLE
 - Good practice to include it so programs can be run on other (non-Windows) systems
- Also known as the "sha-bang", "hashbang", "pound-bang", or "hash-pling"

Comments

- Use the pound sign (hashtag) '#' to comment code

this is a *block* comment

print("Total Score: " + score) # this is an *inline* comment

- Comments have no effect on program operation



Functions



- A function is a group of statements that perform a specific task
- Python includes many built-in functions
 - print is a built-in function

print("Hello out there!")
print()
print("Goodbye!")

argument

Hello out there!
Goodbye!



Ch. 2 Variables and Simple Data Types

[38]

Variables

- A variable is an identifier made up of letters, numbers, and underscores _
- Variables store values that are used in a program
- Variable's values can change during a program's execution
- Variable names must start with a letter or underscore
- Variable names cannot contain spaces, punctuation (other than underscores), or other special characters
- A variable name cannot be a Python keyword (e.g. **while**, or **continue**)





Constants

- A constant is a value that will not change for the life of the program.
- Constants make programs more maintainable and source code less error-prone
- Python does not provide a built-in constant modifier, but we can use naming conventions to "simulate" them
 - "Constants are usually defined on a module level and written in all capital letters with underscores separating words.
 - <http://legacy.python.org/dev/peps/pep-0008/#constants>

THIS_IS_A_CONSTANT

this_is_a_variable

thisIsAlsoAVariable



Basic Data Types and Variables

Data type	Name	Examples	
str	String	"Hello" "50" "Enter a name: "	
int	Integer	21 450	-25 0
float	Floating point	21.9 -92.0	450.65 3.14159



- Variable names are case sensitive
- Variable names should start with a lower-case letter
- Use underscore_ notation or camelCase
 - but not both!

Legal Names	Illegal Names
first_name	10counters
quantity1	notALegal%Name
totalSales	_cannot do this
_systemType	#oneMoreNope



Assigning Values to Variables

- An assignment statement is made up of a variable name, followed by an equal sign, followed by an expression

```
first_name = "Mike"    # set variable first_name to "Mike"
```

```
quantity1 = 3          # set variable quantity1 to 3
```

```
list_price = 19.99     # set variable list_price to 19.99
```

- An expression can be a literal (a "hardcoded" value)
 - A literal can be a string ("**Mike**") or numeric value (**3**, **19.99**)

Numeric Types



- `int`
 - the integer type. Has "unlimited precision" (no theoretical maximum value). You may hit a `MemoryError` exception if you test this (larger than available memory).
- `float`
 - floating point numbers, usually implemented using double in C; information about the internal representation is available in `sys.float_info` (be sure to import the `sys` module first!)
- `complex`
 - complex numbers with a real and imaginary part, which are each a floating point number.

Floating Point Values



- Floating point values are represented using the IEEE-754, as with most languages.
 - <https://docs.python.org/3/tutorial/floatingpoint.html>
- not as precise as we might expect
- can result in inaccurate calculations
- there are ways we can work around this (to be covered later in this course)

[45]

```
sub_total = 74.95
```

```
tax = sub_total * .1      # tax = 7.495000000000000001
```



Working With Numbers in Python

- An arithmetic expression consists of two or more operands (numeric variables or numeric literals) that are operated on by arithmetic operators.

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Integer Division
%	Modulo / Remainder
**	Exponentiation

Examples of Arithmetic Expressions



Example	Result
5 + 4	9
total + fee	depends on variable values
25 / 4	6.25
25 // 4	6
25 % 4	1
3 ** 2	9

Statically-typed
languages (e.g. Java,
C++) do integer
division with '/' based
on operand types

- Spaces around operators are not required, but they make code more readable



Order of Precedence (PEMDAS)

Order	Operators	Direction
(P)arentheses	()	Left to Right
(E)xponent	**	Right to Left
(M)ultiplication (D)ivision	* / // %	Left to Right
(A)ddition (S)ubtraction	+ -	Left to Right

Precedence Examples



Example	Result
$6 * 2 + 5 * 3$	27
$3 + 2 ** 3$	11
$3 ** 2 ** 3$	6561

Changing Precedence With Parentheses



Example	Result
$3 + 4 * 5$	23 (multiplication is first)
$(3 + 4) * 5$	35 (addition is first)
$(\text{subtotal} + \text{fee}) * \text{tax}$	result depends on variable values, but addition is first

Arithmetic Expressions, Variables, and Assignment Statements



- Calculating Sales Tax

```
subtotal = 200.0
```

```
tax_percent = 0.05
```

```
tax_amount = subtotal * tax_percent # 10.0
```

```
grand_total = subtotal + tax_amount # 210.0
```



Mixing Integers and Floating Point Values

- Dividing two numbers will always result in a floating point value, even if both numbers are integers

4/2

2.0

- If an arithmetic operation includes integers both and floating point numbers, the result will always be a floating point number

2 * 3

6

2.0 * 3

6.0

[52]

Underscores in Numbers



- Commas cannot be used to group numbers (in almost all programming languages)
- Python allows the use of underscores to group large numbers to make them more readable

```
x = 1000943563
```

```
y = 1_000_943_563    # same value
```

```
print(x, y)    1000943563 1000943563
```

```
z = 10_00_943_563    # weird, but same value
```

```
print(z)    1000943563
```

Built-In Functions and Type Conversion



- Type conversions are done by using built-in functions

str(1234) # convert 1234 to a string

value = 58.6

print(value)

print(str(value)) # convert to string

num_string = "1234"

num = int(num_string)

print(num)

58.6

58.6

1234

Built-In Functions and Type Conversion



- Output can be concatenated using the '+' concatenation operator

```
num_string = "1234"  
num = int(num_string)  
print(num)
```

```
# print("concatenation error: " + num)  
print("concatenation success: " + str(num))
```

```
num_string2 = "77.888"  
num2 = float(num_string2)  
print(num2)
```

```
#num_error_string = "hello1234"  
#num_error = int(num_error_string)
```

```
1234  
concatenation success: 1234  
77.888
```

Built-in Functions



<u>abs()</u>	<u>dict()</u>	<u>help()</u>	<u>min()</u>	<u>setattr()</u>
<u>all()</u>	<u>dir()</u>	<u>hex()</u>	<u>next()</u>	<u>slice()</u>
<u>any()</u>	<u>divmod()</u>	<u>id()</u>	<u>object()</u>	<u>sorted()</u>
<u>ascii()</u>	<u>enumerate()</u>	<u>input()</u>	<u>oct()</u>	<u>staticmethod()</u>
<u>bin()</u>	<u>eval()</u>	<u>int()</u>	<u>open()</u>	<u>str()</u>
<u>bool()</u>	<u>exec()</u>	<u>isinstance()</u>	<u>ord()</u>	<u>sum()</u>
<u>bytearray()</u>	<u>filter()</u>	<u>issubclass()</u>	<u>pow()</u>	<u>super()</u>
<u>bytes()</u>	<u>float()</u>	<u>iter()</u>	<u>print()</u>	<u>tuple()</u>
<u>callable()</u>	<u>format()</u>	<u>len()</u>	<u>property()</u>	<u>type()</u>
<u>chr()</u>	<u>frozenset()</u>	<u>list()</u>	<u>range()</u>	<u>vars()</u>
<u>classmethod()</u>	<u>getattr()</u>	<u>locals()</u>	<u>repr()</u>	<u>zip()</u>
<u>compile()</u>	<u>globals()</u>	<u>map()</u>	<u>reversed()</u>	<u>__import__()</u>
<u>complex()</u>	<u>hasattr()</u>	<u>max()</u>	<u>round()</u>	
<u>delattr()</u>	<u>hash()</u>	<u>memoryview()</u>	<u>set()</u>	

More Built-In Function Examples



- Functions use 0 or more arguments, some accept different numbers of arguments

function() # 0 arguments

function(x) # 1 argument

function(x, y) # 2 arguments

function("my value is " + int(x)) # concatenated string is one argument

[57]

```
print(round(1234.5678))  
print(round(1234.5678, 3))  
print(float(25))  
print(int(1234.5678))  
print(complex(5,10))  
print(hex(255))
```

```
1235  
1234.568  
25.0  
1234  
(5+10j)  
0xff
```



Write a program that calculates the area of a rectangle

- 1.create and initialize variables to store the length and width (initialize means to set an initial value, always a good programming practice). What type should these variables be?
- 2.create and initialize a variable to store the area (what type?) ($\text{area} = \text{length} * \text{width}$)
- 3.print area variable

[58]



Area Calculator

```
>>> length = 10.0
>>> width = 5.0
>>> area = length * width
>>> print(area)
50.0
>>> |
```



Compound Assignment Operators

- Compound assignment operators (aka augmented assignment operators) are shorthand for common assignment operations

Operator	Description
+=	adds the right side to the variable's value and assigns the result to the variable (can also be used for string concatenation)
-=	subtracts the right side from the variable's value and assigns the result to the variable
*=	multiplies the variable's value by the right side and assigns the result to the variable
/=	divides the variable's value by the right side and assigns the result to the variable
%=	performs remainder division on the variable's value (by the right side) and assigns the result to the variable

Using Compound Assignment Operators



```
counter = 0
counter = counter + 1      # counter = 1
counter += 1              # counter = 2
```

```
score_total = 10
score_total *= 2          # score_total = 20
score_total /= 5          # score_total = 4
```

```
x = 10
x += " # this number is 10" # TypeError: unsupported operand type(s)
```

```
s = "This number is 10: "
s += 10                      # TypeError: unsupported operand type(s)
s += str(10)
print(s)                    # This number is 10: 10
```



Assigning Strings to Variables

- string literals are sequences of characters (including special characters and digits) enclosed in quotes. Use the '=' symbol to assign them to variables.

```
first_name = "Bob"
```

```
last_name = 'Smith' # Python accepts single quotes
```

```
empty_string = ""
```

```
singleSpace = " " # hard to see, but there is a space
```

```
anExcitedString = "Hello!!!"
```

Concatenating Strings



- "concatenation" joins two or more strings together using the '+' symbol:

```
first_name = "Bob"
```

```
last_name = 'Smith'
```

```
full_name = first_name + " " + last_name
```

```
# result is "Bob Smith"
```

```
lastNameFirst = last_name + ", " + first_name
```

```
# result is "Smith, Bob"
```

The str() Function

- The str() function converts numeric data to string data so you can use concatenation

```
print("Total Score: "  
      + str(score_total)  
      + "\nAverage Score: "  
      + str(average_score))
```





Implicit Continuation of a String

- A long string can be continued over multiple lines
- The interpreter looks for continuations on subsequent lines if it doesn't find a complete statement

```
print("Total Score: "          # continued  
      + str(score_total)      # continued  
      + "\nAverage Score: "   # continued  
      + str(average_score))    # done!
```

[65]



Using Special Characters in Strings

- An escape sequence is a character combined with a backslash to give it special meaning

Common Escape Sequences

Sequence	Character
<code>\n</code>	New line
<code>\t</code>	Tab
<code>\r</code>	Return
<code>\"</code>	Quotation mark in a double quoted string
<code>\'</code>	Quotation mark in a single quoted string
<code>\\</code>	Backslash



- **The new line character**

```
print("Title: Python Programming\nQuantity: 5")
```

Displayed on the console

```
Title: Python Programming
Quantity: 5
```

- **The tab and new line characters**

```
print("Title: \t\tPython Programming\nQuantity: \t5")
```

Displayed on the console

```
Title:           Python Programming
Quantity:        5
```



- **The backslash in a Windows path**

```
print("C:\\murach\\python")
```

Displayed on the console

```
C:\murach\python
```

- **Four ways to include quotation marks in a string**

```
"Type \"x\" to exit"    # String is: Type "x" to exit.
'Type \'x\' to exit'    # String is: Type 'x' to exit.
"Type 'x' to exit"     # String is: Type 'x' to exit.
'Type "x" to exit'     # String is: Type "x" to exit.
```

[68]



- **Don't mix the surrounding quotes:**

```
ssmixed = 'mixed quoted string'
```

SyntaxError: EOL while scanning string literal

f-strings

- Python 3.6 introduced "Literal String Interpretation", or "f-strings"
- f-strings provide a way to embed expressions inside string literals, using a minimal syntax
- an f-string is a literal string, prefixed with 'f', which contains expressions inside braces
- any variable found within braces is replaced with its value





f-strings (example)

```
first_name = "john"
last_name = "smith"
full_name = f"{first_name} {last_name}"
print(full_name)
print(full_name.title()) # convert to "Title" case
```

```
john smith
John Smith
```



Examining the print() function

- print accepts arguments in many different forms
- A comma is used as the argument separator

```
print(19.99)
```

```
# 19.99
```

```
print("Price:", 19.99)
```

```
# Price: 19.99
```

```
print(1, 2, 3, 4)
```

```
# 1 2 3 4
```

[72]



- There are different ways to get the same result

- A print() function that receives four arguments

```
print("Total Score:", score_total,  
      "\nAverage Score:", average_score)
```

- A print() function that receives one string as the argument

```
print("Total Score: " + str(score_total) +  
      "\nAverage Score: " + str(average_score))
```

- If score_total is 240 and average_score is 80, this is what is displayed:

Total Score: 240

Average Score: 80



- print also allows "sep" and "end" arguments
 - defaults to a space character

```
print(1, 2, 3, 4, sep=' | ')      # 1 | 2 | 3 | 4
print('foo', 'bar', sep=' -> ')   # foo -> bar
print('eggs', 'ham', sep='\t')    # eggs      ham
```

- using "end"

```
print(1, 2, 3, 4, end='!!!')      # 1 2 3 4!!!
print('ham', 'eggs', 'toast', sep=' & ', end=': Yum!')
                                # ham & eggs & toast: Yum!
```

Getting User Input



- The input function prompts the user and reads input from the keyboard
 - A prompt provides a 'hint' to the user that they must take an action
 - The program waits until the <Enter> key is pressed
 - The user's keyboard entry is saved in a variable

```
first_name = input("Enter your first name: ")  
print("Hello, " + first_name + "!!")
```

- At the prompt, the user types "Mike" and the <Enter> key

```
Enter your first name: Mike  
Hello, Mike!
```



- The prompt can be displayed separately from the input function by calling `input()` with no arguments:

```
print("What is your first name?")  
first_name = input()  
print("Hello, " + first_name + "!" )
```

What is your first name?

Mike

Hello, Mike!

- Notice the difference? The prompt is on a separate line



- We cannot use the input function directly to obtain numeric data from the user
- We have to convert the input from a string to a number type

```
score_total = 0
```

```
score = input("Enter your score: ") # score is a string
```

```
score_total += score # causes an error
```

[77]



Numeric Conversion Functions

`int(data)` # converts a string to an integer type

`float(data)` # converts a string to a floating point type

- This code causes an exception:

```
x = 15
```

```
y = "5"
```

```
z = x + y # TypeError: can't add an int to a str
```

- Using the `int()` function fixes it:

```
x = 15
```

```
y = "5"
```

```
z = x + int(y) # z is 20
```



- The following code reads a string from the user and converts it to an integer
 - Notice that we use the same variable name; we are dynamically changing the type of the variable

```
quantity = input("Enter the quantity: ") # get str type
quantity = int(quantity)                  # convert to int type
```

- How to use chaining to get the value in one statement

```
quantity = int(input("Enter the quantity: "))
```



- The following code reads a string from the user and converts it to an float

```
price = input("Enter the price: ")    # get str type  
price = float(price)                 # convert to float type
```

- How to use chaining to get the value in one statement

```
price = float(input("Enter the price: "))
```


The Round Function

- `round()` takes a numeric argument and rounds it to the specified number of significant digits

```
num1 = 3.14159
```

```
num2 = round(num1, 2)
```

```
print(num1, num2, sep=',')
```

- displays

```
3.14159,3.14
```





- If the number of digits (the second argument) is omitted, it defaults to 0

```
print(round(7.59))
```

- displays **8**



- Rounding behavior as of Python 3.x:

- The return value is an integer if called with one argument, otherwise of the same type as x."

x = round(3.14) # returns an int

x = round(3.0, 0) # returns a float

- "round(x[, n]): ... values are rounded to the closest multiple of 10 to the power minus n; if two multiples are equally close, rounding is done toward the even choice (so, for example, both round(0.5) and round(-0.5) are 0, and round(1.5) is 2).
- <https://docs.python.org/release/3.1.5/library/functions.html#round>



```
>>>print(round(0.5))
```

0

```
>>>print(round(1.5))
```

2

```
>>>print(round(2.5))
```

2

```
>>>print(round(3.5))
```

4



- Remember that floating point values are not always precise:

```
print(round(2.675, 2))
```

displays

2.67

because internally 2.675 is represented as
2.6749999999999999



Write a program that calculates miles per gallon, given mile driven and gallons used

[86]

1. miles per gallon = miles driven / gallons used
2. round the result to 2 places



Miles Per Gallon Program

- "Quick" version: enter necessary code for interpreter using IDLE

```
>>> miles_driven = 150
>>> gallons_used = 5.875
>>> mpg = miles_driven / gallons_used
>>> mpg = round(mpg, 2)
>>> print("Miles per gallon = " + str(mpg))
Miles per gallon = 25.53
>>> |
```

[87]

Python Source Code File

- Enter code in file editor
- Save with ".py" extension.
- Run from command line via
"python filename.py"
- or open file in IDE (e.g. IDLE) and execute.

```
#!/usr/bin/env python3
```

```
# miles_per_gallon  
# D. Singletary  
# 6/12/17  
# calculates miles per gallon
```

```
print("The Miles Per Gallon program")  
print()
```

```
# get input from the user  
miles_driven= float(input("Enter miles driven:\t\t"))  
gallons_used = float(input("Enter gallons of gas used:\t"))
```

```
# calculate and round miles per gallon  
mpg = miles_driven / gallons_used  
mpg = round(mpg, 2)
```

```
# display the result  
print()  
print("Miles Per Gallon:\t\t" + str(mpg))  
print()  
print("Bye")
```





Winter Working Connections 2020

Python for Data Science

[89]

Module 2

Lists, Tuples, and For Loops



Ch. 3 Introducing Lists

[90]

Lists

<https://docs.python.org/3/library/stdtypes.html#list>

- A list stores a list of items
- lists are built-in data types in a category known as sequences
 - square brackets [] denote a list
 - elements are comma-separated
 - items are stored in the order in which they are added
 - lists are similar to arrays in other programming languages, but lists do not require homogeneity (elements of the same type)

```
# a list of 5 elements  
temps = [48.0, 35.0, 20.2, 100.0, 42.0]
```



Lists



```
# a list of 4 string elements
inventory = ["hat", "shirt", "pants", "shoes" ]
# a list of 3 different elements: string, int, float
movie = ["The Holy Grail", 1975, 9.99]
# an empty list
test_scores = []
print(movie)
# don't need print() in IDLE
movie
```

['The Holy Grail', 1975, 9.99]

['The Holy Grail', 1975, 9.99]

- Individual elements of a list can be accessed by inserting a 0-based numeric index in square brackets following the name of the list

```
temps = [48.0, 35.0, 20.2, 100.0, 42.0]
print(temps[0])
print(temps[4])
print(movie[0].upper())
```

48.0

42.0

THE HOLY GRAIL

[92]

List Indexes



- a list index cannot be larger than the size of the list (number of elements) minus 1

```
temps = [48.0, 30.5, 20.2, 100.0, 42.0]
print(temps[5])
```

IndexError: list index out of range

- In Python an index can also be *negative*

```
# -1 is always the last element
print(temps[-1])
# the first element
print(temps[-5])
print(temps[-6])
```

42.0

48.0

IndexError: list index out of range

Use -1 as an index to the last element in a list (will still return an error if the list is empty)

Modifying List Elements



- Modifying elements is similar to accessing them
 - use a 0-based index for assignment

```
# set fourth element
temps[3] = 98.0
# set second element
inventory[1] = "socks"
```

- The asterisk acts as a repetition operator when creating (and initializing) lists

```
scores = [0] * 5      # same as scores = [0, 0, 0, 0, 0]
```



Adding Elements to a List

- The append method adds items to the end of a list
- increases length by 1

```
stats = [48.0, 30.5, 20.2, 100.0]  
stats.append(99.5)  
print(stats)
```

```
[48.0, 30.5, 20.2, 100.0, 99.5]
```



Inserting Elements in a List

- The insert method inserts an item anywhere in a list
- shifts all items right, increases length by 1

```
stats = [48.0, 30.5, 20.2, 100.0]
# insert(index, data)
stats.insert(1, 37.5)
print(stats)
```

```
[48.0, 37.5, 30.5, 20.2, 100.0]
```




Removing Elements From a List

- The remove method removes an element from a list by value
 - shifts all items left, decreases length by 1
 - removes the first item if there are duplicates (loop required to remove all duplicated elements)
 - if item isn't found, raises a ValueError

```
stats = [48.0, 30.5, 20.2, 100.0]
stats.remove(20.2)
print(stats)
```

```
[48.0, 30.5, 100.0]
```



Removing Elements Using del

- The del statement removes elements by index

```
stats = [48.0, 30.5, 20.2, 100.0]
del stats[1]
print(stats)
```

```
[48.0, 20.2, 100.0]
```

[98]



Accessing an Element using index()

- Use the index method to find the index of a specified element
- if item isn't found, raises a ValueError

```
stats = [48.0, 30.5, 20.2, 100.0]
i = stats.index(100.0)
# the index of 100.0
print(i)
```

3

[99]

Popping an Element



- The `pop()` method removes an element from the list
 - if an index is not provided, removes the last element
 - otherwise the element at the specified index is removed
 - decreases length of list by 1
 - if item at a specified index is not found, raises an `IndexError` (“pop index out of range”)
 - if list is empty, pop with no index argument raises an `IndexError` (“`IndexError: pop from empty list`”)

```
stats = [48.0, 30.5, 20.2, 100.0]
# remove 100.0 from end of list
stats.pop()
# remove 30.5 based on specified index
stats.pop(1)
print(stats)
```

[48.0, 20.2]

[100]



Saving the Popped Element

- `pop()` returns the popped element, allowing the removed element to be saved in a variable or printed

```
stats = [48.0, 30.5, 20.2, 100.0]
last_element = stats.pop()
first_element = stats.pop(0)
print(stats)
print(last_element + first_element)
```

```
[30.5, 20.2]
[148.0]
```

[101]

Sorting a List



- the `sort()` method sorts a list in place ("permanently")

```
stats = [48.0, 30.5, 20.2, 100.0]
stats.sort()
print(stats)
```

[20.2, 30.5, 48.0, 100.0]

- the `sorted()` function (not method) returns a sorted copy of the list

- does not modify the original list
- copy can be assigned to a variable

```
sorted_stats = sorted(stats)
print(stats)
print(sorted_stats)
print(sorted_stats)
```

[48.0, 30.5, 20.2, 100.0]

[20.2, 30.5, 48.0, 100.0]

[20.2, 30.5, 48.0, 100.0]

[102]



Reversing a List

- Use the `reverse()` method to reverse a list's order
 - modifies the original list
 - does not sort
 - to restore to original, just call `reverse()` again

```
stats = [48.0, 30.5, 20.2, 100.0]
# reverse the list
stats.reverse()
print(stats)
# restore original order
stats.reverse()
print(stats)
```

```
[100.0, 20.2, 30.5, 48.0]
[48.0, 30.5, 20.2, 100.0]
```

[103]

Length of a List



- Use the `len()` function to obtain the length of a list

```
stats = [48.0, 30.5, 20.2, 100.0]
print(len(stats))
movie = ["The Holy Grail", 1975, 9.99]
print(len(movie))
movie.remove(9.99)
print(movie)
print(len(movie))
stats.pop()
print(stats)
print(len(stats))
```

```
4
3
['The Holy Grail', 1975]
2
[48.0, 30.5, 20.2]
3
```

[104]

Lists of Lists



- A two-dimensional list is a "list of lists"

```
# create a 3-row, 4-column list of student scores
students = ["Joel", 85, 95, 70],
            ["Anne", 95, 100, 100],
            ["Mike", 77, 70, 80]]
print(students)
```

```
[['Joel', 85, 95, 70], ['Anne', 95, 100, 100], ['Mike', 77, 70, 80]]
```

[105]

Accessing Individual Elements in Two-Dimensional Lists



```
print(students)
print(students[0])
print(students[0][1])
# Change one score
students[0][1]=95
print(students[0][1])
```

```
[['Joel', 85, 95, 70], ['Anne', 95, 100, 100], ['Mike', 77, 70, 80]]
['Joel', 85, 95, 70]
85
95
```

[106]

Appending an Element to a Two-Dimensional List



```
print(students)
# create an empty student list
student = []
student.append("Mary")
student.append(100)
student.append(85)
student.append(87)
students.append(student)
print(students)
```

```
[[['Joel', 95, 95, 70], ['Anne', 95, 100, 100], ['Mike', 77, 70, 80]]
 [['Joel', 95, 95, 70], ['Anne', 95, 100, 100], ['Mike', 77, 70, 80], ['Mary', 100, 85, 87]]
```

[107]

Inserting an Element into a Two-Dimensional List



```
student2 = []
student2.append("Jamaal")
student2.append(90)
student2.append(92)
student2.append(88)
students.insert(1, student2)
print(students)
```

```
[['Joel', 95, 95, 70], ['Jamaal', 90, 92, 88], ['Anne', 95, 100, 100], ['Mike', 77, 70, 80],  
['Mary', 100, 85, 87]]
```

[108]

Removing Elements from a Two-Dimensional List



```
# pop removes and returns the item with index = 2
students.pop(2)
print(students)
students[2].pop(3)
print(students)
```

```
['Anne', 95, 100, 100]
[['Joel', 95, 95, 70], ['Jamal', 90, 92, 88], ['Mike', 77, 70, 80], ['Mary', 100, 85, 87]]
80
[['Joel', 95, 95, 70], ['Jamal', 90, 92, 88], ['Mike', 77, 70], ['Mary', 100, 85, 87]]
```

[109]



Ch. 4 Working With Lists

[110]

The for Loop



- A for loop runs once for each value in a collection
 - The range() function produces a collection of integers

```
for number in range(5):  
    print(number, end=" ")  
print("\nThe loop has ended.")
```

0 1 2 3 4
The loop has ended.

- The for loop in Python is also known as a collection-controlled loop (also known as a "count-controlled loop")
- **The colon : is required**
- **Indentation of the loop body is required**
- **Give the temporary variable a meaningful name (e.g. "number")**

[111]



- An optional stop value can be included in the range function

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

- The for loop will iterate from the start value to, but not including, the stop value

```
for number in range(1, 12):  
    print(number, end=" ")  
print("\nThe loop has ended.")
```

```
1 2 3 4 5 6 7 8 9 10 11  
The loop has ended.
```

[112]



- An optional step value can also be included in the range function (but this requires start and stop)

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

- The step value specifies an increment for each loop iteration

```
for number in range(2, 12, 3):  
    print(number, end=" ")  
print("\nThe loop has ended.")
```

```
2 5 8 11
```

```
The loop has ended.
```

[113]



Write a program which uses a for loop to calculate the sum of the values 1 through 5 and prints the final sum

The sum of 1 to 5 is 15

[114]



```
#!/usr/bin/env python3
# sum.py
#
sum_nums = 0 # accumulator
for number in range(1, 6) # range requires 6 to include 5
    sum_nums = sum_nums + number
print("The sume of 1 to 5 is " + str(sum_nums))
```

The sum of 1 to 5 is 15

- **print statement is not indented, not part of the loop body**

[115]



- Alternative version using a list

```
#!/usr/bin/env python3
# sumlist.py
#
sum_nums = 0          # accumulator
num_list = [1, 2, 3, 4, 5] # 6 not required here
for number in num_list:
    sum_nums = sum_nums + number
print("The sum of 1 to 5 is " + str(sum_nums))
```

The sum of 1 to 5 is 15

- We could avoid the loop and just use the function `sum(num_list)`
 - Note: don't name the accumulator "sum"! The function will then be shadowed by the variable and will be unavailable (`del sum` to fix)

[116]



More for Loop Examples

```
magicians = ['alice', 'david', 'carolina']  
for magician in magicians:  
    print(magician)
```

alice
david
carolina

```
for magician in magicians:  
    print(f"{magician.title()}, that was a great trick!")
```

Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!

[117]



More for Loop Examples/Common Errors

```
# calculate squares from 1 to 10, insert into list
# create an empty list
squares = []
for value in range(1,11):
    squares.append(value ** 2)      # append square to list

print(squares)
```

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

[118]

• Common for Loop Errors

- Forgetting to indent all statements in the loop body
- Indenting a statement that should not be in the loop body
- Forgetting the colon

Nested For Loops



```
# get 2 scores for 3 tests
PROMPT = "Enter test score for test " #constant
MAX_TESTS = 3
MAX_SCORES = 2
test_scores = []
for test in range(1, MAX_TESTS+1):
    for score in range(MAX_SCORES): #0, 1
        one_score = int(input(PROMPT + str(test) + " :"))
        test_scores.append(one_score)
print("Test scores:", test_scores)
```

[119]

Enter test score for test 1:60
Enter test score for test 1:70
Enter test score for test 2:90
Enter test score for test 2:95
Enter test score for test 3:100
Enter test score for test 3:80

Test scores: [60, 70, 90, 95, 100, 80]



List Comprehensions

- A list comprehension allows more efficient coding by combining the for loop and creation of list elements into a single line

```
squares = [value**2 for value in range(1, 11)] # no colon after for!
```

```
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
# print first initial of each name in list
```

```
list_of_names = ["Tom", "David", "Sally", "Pamela", "Robert"]
```

```
# take first letter of each name
```

```
initials = [name[0] for name in list_of_names]
```

```
print(initials)
```

```
['T', 'D', 'S', 'P', 'R']
```

[120]



Slicing a List

- A list slice is a specific group of items in a list
- To create a slice, specify the first and last index
 - As with range(), Python stops one item before the last index

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']  
# create a slice that contains elements 0, 1, 2  
print(players[0:3])  
# create a slice that contains elements 1, 2, 3  
print(players[1:4])
```

```
['charles', 'martina', 'michael']  
['martina', 'michael', 'florence']
```

[121]

More Slice Examples



```
players = ['charles', 'martina', 'michael', 'florence', 'eli']  
print(players[:4]) # omitting first index starts at beginning of list
```

```
['charles', 'martina', 'michael', 'florence']
```

```
print(players[2:]) # omitting last index continues to end of list
```

```
['michael', 'florence', 'eli']
```

```
print(players[-3:]) # use a negative value to display last n elements of a list
```

```
['michael', 'florence', 'eli']
```

```
print(players[0:5:2]) # a 3rd value can be included to skip between items
```

```
['charles', 'michael', 'eli']
```

[122]



Looping Through a Slice

- A slice can be used in a for loop

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']  
print("Here are the first three players on my team:")  
for player in players[:3]:      # be careful with the colons  
    print(player.title())
```

[123]

```
Here are the first three players on my team:  
Charles  
Martina  
Michael
```



Copying Lists Using Slices

- Slices can be used to make new lists

```
my_foods = ['pizza', 'falafel', 'carrot cake']  
friend_foods = my_foods[:]  
print("My favorite foods are:")  
print(my_foods)  
print("\nMy friend's favorite foods are:")  
print(friend_foods)
```

My favorite foods are:
['pizza', 'falafel', 'carrot cake']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake']

[124]

Copying Lists Using Slices



- Individualize each list to show they are indeed copies

```
my_foods = ['pizza', 'falafel', 'carrot cake']  
friend_foods = my_foods[:]
```

```
my_foods.append('cannoli')  
friend_foods.append('ice cream')  
print("My favorite foods are:")  
print(my_foods)  
print("\nMy friend's favorite foods are:")  
print(friend_foods)
```

My favorite foods are:

['pizza', 'falafel', 'carrot cake', 'cannoli']

My friend's favorite foods are:

['pizza', 'falafel', 'carrot cake', 'ice cream']

[125]



Copy Lists Using Slices

- Simple assignment cannot be used to copy lists

```
my_foods = ['pizza', 'falafel', 'carrot cake']  
friend_foods = my_foods  # doesn't work
```

- This assignment creates a second reference to the first list, it doesn't create a copy

```
my_foods.append('cannoli')  
print(friend_foods)          # same list object as my_foods
```

```
['pizza', 'falafel', 'carrot cake', 'cannoli']
```

[126]

Tuples

- A Python tuple is similar to a list, but a tuple is *immutable* (cannot be changed), whereas a list is *mutable* (can be changed)
 - Cannot append to tuples
 - Cannot replace tuple elements
 - Cannot delete tuple elements
- Advantages:
 - faster
 - less "side effects" (errors) when coding





- Creating a tuple:
 - Use parentheses instead of brackets

```
stats = (48.0, 30.5, 20.2, 100.0, 48.0)
```

- Accessing elements:

```
scores = ("Jim", 75, 89, 93)
score = scores[0]      # "Jim"
```

- Remember that elements cannot be changed:

```
scores[1] = 97      # TypeError: 'tuple' object does not support item assignment
```




- Given this tuple:

```
scores = ("Jim", 75, 89, 93)
```

- Unpacking a tuple using a multiple assignment statement:

```
# assign each tuple element to a separate variable
name, score1, score2, score3 = scores
print(name)
print(score1)
print(score2)
print(score3)
```

Jim
75
89
93



More Tuple Operations

- As with lists, for loops can be used with tuples

```
dimensions = (200, 50)
for dimension in dimensions:
    print(dimension)
```

200
50

- Although tuples cannot be modified, they can be reassigned

```
# reassign after initial assignment (above)
dimensions = (400, 100)
for dimension in dimensions:
    print(dimension)    # new tuple, new values
```

400
100

[130]



Winter Working Connections 2020

Programming in Python

[131]

Module 3

Selection Statements



Ch. 5 If Statements

[132]



Selection Statements

- Selection statements control the execution of a program
 - “IF (there is plenty of money) THEN go out to eat”
 - “WHILE (there is not enough money) stay home and cook”
- Boolean expressions evaluate to True or False and are evaluated in selection statements
 - (there is plenty of money) is a Boolean expression which evaluates to True or False
 - (there is not enough money) is a Boolean expression which evaluates to True or False



Relational Operators

- Relational Operators are used in Python to construct Boolean expressions

Operator	Name	Description
==	Equal to	Returns True if both operands are equal
!=	Not equal to	Returns True if the operands are not equal
>	Greater than	Returns True if the left operand is greater than the right operand
<	Less than	Returns True if the left operand is less than the right operand
>=	Greater than or equal to	Returns True if the left operand is greater than or equal to the right operand
<=	Less than or equal to	Returns True if the left operand is less than or equal to the right operand



Examples of Boolean Expressions

- Operands should be based on similar types
- All of the following examples evaluate to True or False

```
age == 5
first_name == "John"
quantity != 0
distance > 5.6
fuel_req < fuel_cap
distance >= limit
stock <= reorder_point
rate / 100 >= 0.1
```

- *Don't use = to test for equality, this is for assignment!*



Assigning Values to Boolean Variables

- A Boolean variable is either True or False
 - The literal values True and False can be assigned

```
boolean isActive = False  
boolean rentIsTooHigh = True
```


Logical Operators



- Logical operators are used to combine Boolean expressions
- The combined expressions also evaluate to True or False

Name	Description
AND	Evaluates to True only if <u>all</u> expressions are True
OR	Evaluates to True if <u>any</u> expression is True
NOT	Reverses the value of the expression

- Order of Precedence
 1. NOT operator
 2. AND operator
 3. OR operator

Logical Operator Examples



```
# The AND operator  
age >= 65 AND city == "Chicago"
```

```
# The OR operator  
city == "Greenville" OR age >= 65
```

```
# The NOT operator  
NOT age >= 65
```

```
# Two AND operators  
age >= 65 AND city == "Greenville" AND state == "SC"
```

```
# Two OR operators  
age >= 65 OR age <= 18 OR status == "retired"
```



Logical Operator Examples, cont.

```
# AND and OR operators with parens to clarify sequence of  
operations
```

```
    (age >= 65 AND status == "retired") OR age < 18
```

```
# AND and OR operators with parens to change sequence of  
operations
```

```
    age >= 65 AND (status == "retired" OR state == "SC")
```

[139]

Short-Circuiting

- Short-circuit evaluation uses the behavior of logical operators to avoid unnecessary expression evaluation
 - Can improve performance

- AND operations evaluate to True only if all expressions are True

```
age >= 65 AND city == "Greenville" AND state == "SC"
```

- If `age >= 65` is False, evaluation stops there and result is False
- If `city == "Greenville"` is False, evaluation stops there and result is False





More Short-Circuiting

- OR operations evaluate to True if any expression is True

```
age >= 65 OR age <= 18 OR status == "retired"
```

- If age >= 65, evaluation stops there and result is True
- If age <= 18, evaluation stops there and result is True

- Evaluate more complex expressions last

```
# if flag is False, time_consuming operation won't need to execute  
if flag == True AND time_consuming_operation() == SUCCESS:
```

```
# if flag is True, time_consuming_operation won't need to execute  
if flag == TRUE OR time_consuming_operation() == SUCCESS:
```

[141]

Comparing Strings

- Numbers are compared based on their numeric values
- Strings are compared on a character-by-character basis
 - Character comparisons are based on a numeric encoding called Unicode
 - Each character has an associated numeric value, or code point
 - Python uses an 8-bit Unicode representation known as UTF-8
 - "A character in UTF8 can be from 1 to 4 bytes long. UTF-8 can represent any character in the Unicode standard. UTF-8 is backwards compatible with ASCII."
 - https://www.w3schools.com/charsets/ref_html_utf8.asp





UTF-8 Code Point Examples

- <https://unicode.org/charts/PDF/U0000.pdf>
- These values are equivalent to ASCII symbol values
 - <http://www.asciitable.com/>

Character	UTF-8 Value
!	33
“	34
#	35
\$	36

Character	UTF-8 Value
0	48
1	49
2	50
3	51
...	...
9	57

Character	UTF-8 Value
A	65
B	66
C	67
D	68
...	...
Z	90

Character	UTF-8 Value
a	97
b	98
c	99
d	100
...	...
z	122



String Comparison Examples

Expression	Boolean Result
<code>"apple" < "Apple"</code>	False
<code>"App" < "Apple"</code>	True
<code>"1" < "5"</code>	True
<code>"10" < "5"</code>	True

- Characters are evaluated from left-to-right
 - First characters of both strings are compared, then the second, and so on

Converting Strings

- `lower()` and `upper()` are methods which convert characters in a string to lower (or upper) case
 - A method is associated with an object (e.g. a string).
 - Methods are called using the dot operator

```
>>> string1 = "Mary"
>>> string2 = "mary"
>>> string1 == string2
False
>>> string1.lower() == string2
True
>>> string1.upper() == string2.upper()
True
```



Simplifying User Input Validation



```
entry = input("Enter x to exit: ")  
entry == "x" or entry == "X"    # this works, but  
entry.lower() == "x"           # this is more efficient
```

```
>>> entry = input("Enter x to exit: ")  
Enter x to exit: x  
>>> entry == "x" or entry == "X"    # this works, but  
True  
>>> entry.lower() == "x"           # this is more efficient  
True  
>>>
```

[146]



Selection Statements: The if Statement

- Boolean expressions by themselves aren't very useful or interesting; we need to associate them with some behavior in our code
- The **if** statement controls the execution path based on the results of a boolean expression:

```
if boolean_expression:
```

```
    statements...           # this is a block of statements
```

[147]



- The boolean expression in the if statement must be terminated with a colon ‘:’
- The statements in the if block* must be indented.

```
>>> age=21
>>> if age >= 18:
    print("You may vote")

You may vote
```

[148]

* - Python officially refers to a block as a “suite”



```
>>> age = 17
>>> if age < 18:
    print("You may NOT vote")
```

```
You may NOT vote
```

[149]



Alternative Paths Using the else Statement

- What if we want to take an alternative action when the if statement fails?
- Use the **else** statement:

```
if boolean_expression:  
    statements...  
else:  
    statements...
```

(150)



- **else** uses the same rules as **if**:
 - terminate with colon
 - use indentation for block

```
>>> name = "Smith"
>>> if name.lower() == "smith":
    print("You have a very common name")
else:
    print("Your name may not be as common as Smith")
```

```
You have a very common name
.
```

[151]



```
>>> name = "Jones"
>>> if name.lower() == "smith":
    print("You have a very common name")
else:
    print("Your name may not be as common as Smith")
```

```
Your name may not be as common as Smith
```

(152)

Multiple Paths Using an elif Chain

- If we have multiple expressions to evaluate, use **elif**:

```
if boolean_expression:
```

```
    statements...
```

```
elif boolean_expression2:
```

```
    statements...
```

```
elif boolean_expression3:
```

```
    statements...
```

```
else:    # not required, but use if necessary
```

```
    statements...
```





```
>>> name = "Jones"
>>> if name.lower() == "smith":
    print("You have a very common name")
elif name.lower() == "jones":
    print("Your name is slightly less common than Smith")
else:
    print("Maybe you should change your name?")
```

```
Your name is slightly less common than Smith
```

[154]



```
>>> invoice_total = 375.79
>>> if invoice_total >= 500.00:
    discount = .2
elif invoice_total >= 250.00:
    discount = .1
elif invoice_total >= 100.00:
    discount = 0.5
else:
    discount = 0.0
```

```
>>> print("discount = " + str(discount))
discount = 0.1
```

Watch the order!
What would happen
if

`invoice_total >= 100.00`
was evaluated first?

[155]



Write a program which uses an **if** statement that asks the user for a number of sides (up to 5) and identifies the shape associated with that number. Use **elif** and **else** as necessary.

Sides	Shape
0	error
1	line
2	polyline
3	triangle
4	rectangle (could also be a square)
5	pentagon
> 5	error



```
#!/usr/bin/env python3
# shapes.py
# prompt user for sides and display corresponding shape

# get user input and convert to an integer
sides = input("Please enter a number of sides from 1 to 5: ")
sides = int(sides)

# print the shape associated with that number of sides
# or print an error if the number is out of range
if sides == 1:
    print("1 side is a line")
elif sides == 2:
    print("2 sides is a polyline")
elif sides == 3:
    print("3 sides is a triangle")
elif sides == 4:
    print("4 sides is a rectangle - and maybe a square")
elif sides == 5:
    print("5 sides is a pentagon")
else:
    print("that number is out of my range")
```

Nested IF Statements

- We sometimes need to make a secondary decision within a primary one. We can use a nested if for this:

```
if a customer chose an apple
    if they chose a Red Delicious
        show Red Delicious price
    else if they chose a Granny Smith
        show Granny Smith price
    else if they chose a McIntosh
        show McIntosh price
else if a customer chose an orange
    if they chose a Valencia
        show Valencia price
    else if they chose a Maltese
        show Maltese price
```





```
#!/usr/bin/env python3
# nested-if.py
#
fruit = "apple"
appleType = "McIntosh"
orangeType = "Unknown"
#
if fruit == "apple":
    if appleType == "Red Delicious":
        print("price is $1.49")
    elif appleType == "Granny Smith":
        print("price is $1.79")
    elif appleType == "McIntosh":
        print("price is $2.11")
    else:
        print("unknown apple type")
elif fruit == "orange":
    if orangeType == "Valencia":
        print("price is $1.19")
    elif orangeType == "Maltese":
        print("price is $1.29")
    else:
        print("unknown orange type")
else:
    print("unknown fruit type")
```



Write a program which applies customer discounts based on a customer type code and the total sales (invoice) amount

Type code	Invoice total	Discount percentage
r (retail)	< 100	0
	>= 100 and < 250	0.1
	>= 250	0.2
w (wholesale)	< 500	0.4
	>= 500	0.5

```
customer_type = "w"
invoice_total = 125
...
Total cost with discount is 75.0
```

[160]



```
#!/usr/bin/env python3
# nested-if-discounts.py
#
customer_type = "w"
invoice_total = 125

if customer_type.lower() == "r":
    if invoice_total < 100:
        discount_percent = 0
    elif invoice_total >= 100 and invoice_total < 250:
        discount_percent = .1
    elif invoice_total >= 250:
        discount_percent = .2
elif customer_type.lower() == "w":
    if invoice_total < 500:
        discount_percent = .4
    elif invoice_total >= 500:
        discount_percent = .5
else:
    discount_percent = 0

print("Total cost with discount is " +
      str(invoice_total - (invoice_total * discount_percent)))
```

To Nest Your Ifs ... or Not ...



This code gives the same results, without using nested If statements.

Which approach is preferable?

```
# the discounts for Retail customers
if customer_type.lower() == "r" and invoice_total <
100:
    discount_percent = 0
elif customer_type.lower() == "r" and (
    invoice_total >= 100 and invoice_total < 250):
    discount_percent = .1
elif customer_type.lower() == "r" and invoice_total
>= 250:
    discount_percent = .2
# the discounts for Wholesale customers
elif customer_type.lower() == "w" and invoice_total
< 500:
    discount_percent = .4
elif customer_type.lower() == "w" and invoice_total
>= 500:
    discount_percent = .5
# all other customers
else:
    discount_percent = 0
```

[162]



Note the duplicated check for customer type. This code is harder to maintain; if we needed to change to something other than “r”, what if we miss one of the “r” literals in the elif statements?

```
# the discounts for Retail customers
if customer_type.lower() == "r" and
invoice_total < 100:
    discount_percent = 0
elif customer_type.lower() == "r" and (
    invoice_total >= 100 and invoice_total <
250):
    discount_percent = .1
elif customer_type.lower() == "r" and
invoice_total >= 250:
    discount_percent = .2
# the discounts for Wholesale customers
elif customer_type.lower() == "w" and
invoice_total < 500:
    discount_percent = .4
elif customer_type.lower() == "w" and invoice
# all other customers
else:
    discount_percent = 0
```

Using If Statements with Lists



```
# loop through pizza toppings, use If statement to handle missing
# topping
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese' ]
for requested_topping in requested_toppings:
    if requested_topping == 'green peppers':
        print("Sorry, we are out of green peppers right now.")
    else:
        print(f"Adding {requested_topping}.")
print("\nFinished making your pizza!")
```

[164]

Adding mushrooms.
Sorry, we are out of green peppers right now.
Adding extra cheese.

Finished making your pizza!

Checking That a List Is Not Empty



```
# if list is empty, don't process it
requested_toppings = [ ]
if requested_toppings:
    for requested_topping in requested_toppings:
        print(f"Adding {requested_topping}.")
    print("\nFinished making your pizza!")
else:
    print("No toppings!")
```

[165]

No toppings!

Using "in" to Find a List Element

```
inventory = ["staff", "hat", "bread"]  
item = "bread"  
if item in inventory:  
    inventory.remove(item)  
# inventory = ["staff", "hat"]
```



Using Multiple Lists



```
# check for existence in one list of each element in another list
available_toppings = ['mushrooms', 'olives', 'green peppers',
                     'pepperoni', 'pineapple', 'extra cheese']
requested_toppings = ['mushrooms', 'french fries', 'extra cheese' ]
for requested_topping in requested_toppings:
    if requested_topping in available_toppings:
        print(f"Adding {requested_topping}.")
    else:
        print(f"Sorry, we don't have {requested_topping}.")
print("\nFinished making your pizza!")
```

[167]

Adding mushrooms.
Sorry, we don't have french fries.
Adding extra cheese.

Finished making your pizza!



Coding Style for Selection Statements

- PEP-8 recommends a single space around comparison operators

```
if age < 4:
```

- is preferable to

```
if age<4:
```

```
if x == y and y <= 10:
```

- is preferable to

```
if x==y and y<=10:
```




The break Statement

- The break statement breaks out of a loop by causing execution to jump to the statement following the loop

```
while True:      # while loops will be covered in an upcoming module
    data = input("Enter a number to square, 'exit' when done: ")
    if data == "exit":
        break      # while True
    i = int(data)
    print(i, "squared is", i * i)
print("Exiting")
```

```
Enter a number to square, 'exit' when done: 1
1 squared is 1
Enter a number to square, 'exit' when done: 2
2 squared is 4
Enter a number to square, 'exit' when done: 3
3 squared is 9
Enter a number to square, 'exit' when done: 4
4 squared is 16
Enter a number to square, 'exit' when done: exit
Exiting
```

[169]

The continue Statement

- The continue statement causes execution to jump to the top of the loop (causing the condition to be reevaluated)

```
more = "y"
while more.lower() == "y":
    miles_driven = float(input("Enter miles driven: "))
    gallons_used = float(input("Enter gallons of gas used: "))

    # validate input
    if miles_driven <= 0 or gallons_used <= 0:
        print("Both entries must be greater than zero. Try again.")
        continue # while more.lower() == "y":

    mpg = round(miles_driven / gallons_used, 2)
    print("Miles Per Gallon:", mpg)

    more = input("Continue? (y/n): ")
    print()

print("Okay, bye!")
```

```
Enter miles driven: 500
Enter gallons of gas used: 23
Miles Per Gallon: 21.74
Continue? (y/n): y

Enter miles driven: 400
Enter gallons of gas used: -2
Both entries must be greater than zero. Try again.
Enter miles driven: 400
Enter gallons of gas used: 18
Miles Per Gallon: 22.22
Continue? (y/n): n

Okay, bye!
```



[170]



Use Break and Continue With Discretion

- break and continue are acceptable if not over-used
 - writing clean and efficient (short and to the point) loops is critical
- long, complex loops are difficult to read and maintain
 - break and continue exacerbate problems here
- break and continue apply to the inner-most loop when nested loops are used
 - It is always helpful to add a comment indicating which loop you are breaking

[171]



An Else Clause for a Loop???

```
for n in range(2, 10):  
    for x in range(2, n):  
        if n % x == 0:  
            print(n, 'equals', x, '*', n//x)  
            break        # inner loop break  
    else:                # inner loop else  
        # loop ended without finding a factor  
        print(n, 'is a prime number')
```

2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3

- The else runs when no break occurs (\Rightarrow n is prime), when all items in the range are exhausted



Winter Working Connections 2020

Programming in Python

[173]

Module 4

Dictionaries



Ch. 6 Dictionaries

(174)

Dictionaries



- A Python dictionary is a form of associative array (also known as a map)

- Each entry contains a key and a value

```
dictionary_name = { key1 : value1, key2 : value2, ... }
```

- To create a dictionary:

```
alien_0 = { 'color' : 'green', 'points' : 5 }
```

- A value can be looked up using the key

```
print(alien_0['color'])  
print(alien_0['points'])
```

green
5

Key-Value Pairs



- Dictionaries store **key-value pairs**
 - Each key is connected to (associated with) a value
 - A key must be of an immutable data type such as a string, number, or tuple
 - Values can be of any type, including lists and other dictionaries
 - Keys are unique within a dictionary, values are not

[176]

```
alien_0 = { 'color' : 'green', 'points' : 5 }
```

Diagram illustrating key-value pairs in a dictionary:

Under 'color' : 'green':

↑
key : value pair

Under 'points' : 5:

↑
key : value pair

Key-Value Pair Data Types



```
# strings as keys and values
countries = {"CA": "Canada", "US": "United States", "MX": "Mexico"}

# numbers as keys, strings as values
numbers = {1: "One", 2: "Two", 3: "Three", 4: "Four", 5: "Five"}

# strings as keys, values of mixed types
movie = { "name": "The Holy Grail", "year": 1975, "price": 9.99}

# an empty dictionary
book_catalog = {}
```

Printing a Dictionary

```
print(countries)
```

```
{'MX': 'Mexico', 'CA': 'Canada', 'US': 'United States'}
```





Accessing Dictionary Values Using Keys

- `dictionary_name[key]` is used to access a value associated with "key"

```
countries = {"CA": "Canada",  
            "US": "United States",  
            "GB": "Great Britain",  
            "MX": "Mexico"}
```

```
print(countries["MX"])  
print(countries["IE"])
```

Mexico
Traceback
KeyError: 'IE'

- If a key is already present, we can modify the existing value:

```
countries["GB"] = "United Kingdom"
```

- If a key is not present, we can add a key/value pair:

```
countries["FR"] = "France"
```

Checking if a Key Exists



```
code = "IE"
if code in countries:
    country = countries[code]
    print(country)
else:
    print("There is no country for this code: " + code)
```



Using the get() Method with a Dictionary

- get() returns the value of the item with the specified key

dictionary_name.get(key [, default_value])

```
print(countries.get("MX"))  
print(countries.get("IE"))  
print(countries.get("IE", "Unknown"))
```

Mexico
None
Unknown

Difference Between get() and Brackets []



- The difference while using get() vs. square brackets [] is that get() returns None instead of raising a KeyError, if the key is not found

```
country = countries["IE"]
```

```
Traceback (most recent call last):  
File "<pyshell#3>", line 1, in <module>  
    country = countries["IE"]  
KeyError: 'IE'
```

```
country = countries.get("IE")  
print(country)
```

```
None
```

[182]

Deleting an Item from a Dictionary

- To delete using a key:

```
del countries["MX"]  
del countries["IE"]
```

Traceback:
KeyError: 'IE'



Checking For a Key Before Deleting



```
code = "IE"
if code in countries:
    country = countries[code]
    del countries[code]
    print(country + " was deleted.")
else:
    print("There is no country for this code: " + code)
```




Other Methods for Deleting

- `pop()` returns the value of the deleted item, instead of just deleting it

```
country = countries.pop("US")
print(country)
print(countries.pop("IE"))

print(countries.pop("IE", "Does Not Exist"))
```

United States
Traceback:
 KeyError: 'IE'
Does Not Exist

- The `clear()` method removes all items from a list

```
countries.clear()
print(countries)
```

{}

Getting All Keys and Values



- A for loop can be used to iterate through a dictionary

```
for code in countries.keys():  
    print(code + "      " + countries[code])
```

MX	Mexico
US	United States
CA	Canada

Implicit Key Iteration



- `countries.keys()` does not need to be specified, Python provides an implicit key iterator when using a for loop

```
for code in countries.keys():  
for code in countries:           # inherent key iterator  
    print(code + " " + countries[code])
```

MX	Mexico
US	United States
CA	Canada

Looping Through Values



- The `items()` method returns tuples of (key, value)

```
for code, name in countries.items():  
    print(code + "    " + name)
```

MX	Mexico
US	United States
CA	Canada

- `values()` returns only the values

```
for name in countries.values():  
    print(name)
```

Mexico
United States
Canada

Sort a Dictionary's Keys - Ordered Output



```
favorite_languages= { 'jen' : 'Python', 'sarah' : 'C++',  
                      'edward' : 'ruby', 'phil' : 'Python', 'robert' : 'Java' }
```

```
for name in favorite_languages:  
    print(name, favorite_languages[name], sep=', ')
```

```
jen, Python  
sarah, C++  
edward, ruby  
phil, Python  
robert, Java
```

```
for name in sorted(favorite_languages):  
    print(name, favorite_languages[name], sep=', ')
```

```
edward, ruby  
jen, Python  
phil, Python  
robert, Java  
sarah, C++
```

[189]

Creating a List from a Dictionary's Keys



- `list(countries.keys())` is a constructor, one of the object-oriented topics to be covered in a subsequent module

```
countries = {"CA": "Canada", "US": "United States",  
            "MX": "Mexico"}
```

```
codes = list(countries.keys())
```

```
codes.sort()
```

```
for code in codes:
```

```
    print(code + "    " + countries[code])
```

CA	Canada
MX	Mexico
US	United States

Dictionary View Objects



- A view is a "window" which represents components of a dictionary
- Views are not copies; modifying the original dictionary also modifies the view

```
dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
keys = dishes.keys() # keys is a view
values = dishes.values() # values is a view
```

```
print(keys)
print(values)
```

```
dict_keys(['eggs', 'sausage', 'bacon', 'spam'])
dict_values([2, 1, 1, 500])
```

[191]

View Objects Are Dynamic



- view objects are dynamic and reflect dictionary changes

```
print("deleting eggs!")  
del dishes['eggs']      # delete from full dictionary  
print(keys)   # no more eggs (key)  
print(values) # no more eggs (value of 2)
```

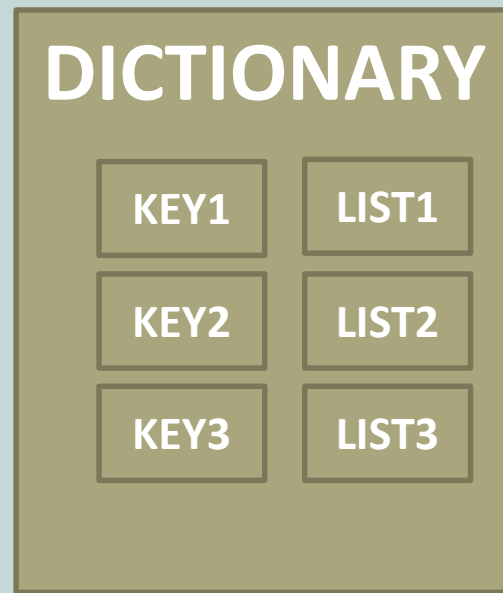
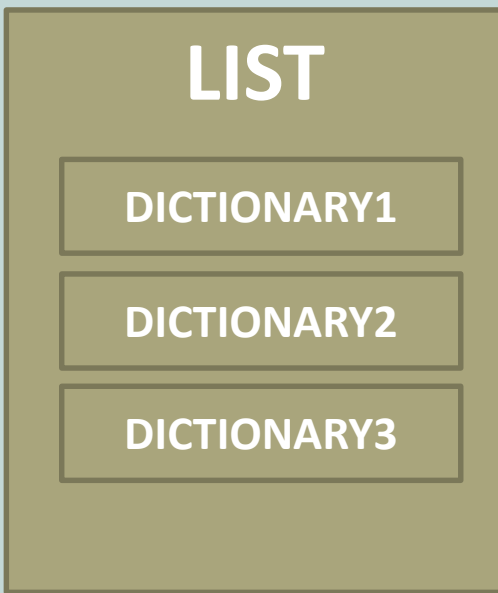
```
dict_keys(['eggs', 'sausage', 'bacon', 'spam']) (previous slide)  
dict_values([2, 1, 1, 500])                     (previous slide)  
deleting eggs!  
dict_keys(['sausage', 'bacon', 'spam'])  
dict_values([1, 1, 500])
```

[192]

Nesting



- Dictionaries can be nested in a list
- A list can be nested in a dictionary (as a value, not as a key)
- A dictionary can be nested in another dictionary (as a value)



A List of Dictionaries



```
alien_0 = { 'color' : 'green', 'points' : 5 }  
alien_1 = { 'color' : 'yellow', 'points' : 10 }  
alien_2 = { 'color' : 'red', 'points' : 15 }
```

```
aliens = [ alien_0, alien_1, alien_2 ]
```

```
for alien in aliens:  
    print('alien:', alien)
```

```
alien: {'color': 'green', 'points': 5}  
alien: {'color': 'yellow', 'points': 10}  
alien: {'color': 'red', 'points': 15}
```

[194]

A Dictionary of Lists



```
pizza = { 'crust' : ['thick', 'stuffed'],  
          'toppings' : ['mushrooms', 'extra cheese'],  
          'sauce' : ['tomato', 'fennel', 'basil'] }
```

```
for p in pizza:  
    print(p, pizza.get(p))
```

```
crust ['thick', 'stuffed']  
toppings ['mushrooms', 'extra cheese']  
sauce ['tomato', 'fennel', 'basil']
```

[195]



A Dictionary of Dictionaries

```
users = { 'aeinstein' : { 'first' : 'albert', 'last' : 'einstein',  
                          'location' : 'princeton' },  
          'mcurie'     : { 'first' : 'marie', 'last' : 'curie',  
                          'location' : 'paris' },  
          'nbohr'      : { 'first' : 'niels', 'last' : 'bohr',  
                          'location' : 'copenhagen' }  
}
```

```
for user in users:  
    print(user, end=':\n')  
    sub = users.get(user)  
    for info in sub:  
        print('\t', info, ':', sub.get(info))
```

```
aeinstein:  
    first : albert  
    last  : einstein  
    location : princeton  
mcurie:  
    first : marie  
    last  : curie  
    location : paris  
nbohr:  
    first : niels  
    last  : bohr  
    location : copenhagen
```

[196]