

The background features abstract, overlapping green geometric shapes in various shades of green, creating a modern and dynamic look. The shapes are primarily triangles and polygons, some with thin white outlines, set against a white background.

# CSE1100: Intro to Programming Concepts with Python Midterm Review

(Offline reading version with expanded notes)

Dr. Stefan Joe-Yen  
sjoe yen@fit.edu

# Agenda

- ▶ Administrative Items
  - ▶ Announcements
  - ▶ Q&A Responses
- ▶ Lecture Topic: Course Overview
  - ▶ Contact Information
  - ▶ Course Mechanics
  - ▶ Course Outline
  - ▶ Grading Rubrics & Artifacts
- ▶ Demo & Hands-on Exercises
- ▶ Q&A

# Course Goals & Outcomes

- ▶ This course provides students with a general understanding of the concepts and methods required to successfully develop and apply computer software.
- ▶ Upon completion of the course, students will have:
  - ▶ Knowledge of the basic principles of Software Development and Programming Languages
  - ▶ Hands-on program design and testing experience through exercises and projects.
  - ▶ Introduction to real-world applications of the concepts presented.
  - ▶ Practical knowledge of software-driven solutions which can be implemented by students in their respective field of study.

# Major Topics Explored

- ▶ Programming in Python:
- ▶ Why Python?
  - ▶ Easy to learn but powerful enough to be applied in production systems.
  - ▶ Dynamic, High-Level features simplify program structures. This lets programmers express complicated concepts in fewer lines of code
  - ▶ Community support in a wide variety of domains. Freely available libraries help you not have to write things from scratch.
  - ▶ Multi-Paradigm support for various program organization styles such as Functional & Object-Oriented styles.
- ▶ In addition to specific Python programming techniques we will also study some general software development concepts such as:
  - ▶ Reliable Design Principles
  - ▶ Testing, Verification, & Validation
  - ▶ Applications (Real World Examples)
    - ▶ Games
    - ▶ Science
    - ▶ Internet

# Canvas Module Summary

- ▶ M01 - Development Tools & Programming Essentials
- ▶ M02 - Variables & Types
- ▶ M03 - Input & Output (Includes File Processing)
- ▶ M04 - Flow Control & Logic
- ▶ M05 - Lists & Dictionaries
- ▶ M06 - Functions
- ▶ M07 - Testing & Debugging
- ▶ M08 - Real World Examples
- ▶ Extras - Lecture Slides & Notes
- ▶ Extras - Additional Resources

# Key Notes from Demo & Lab Session 1

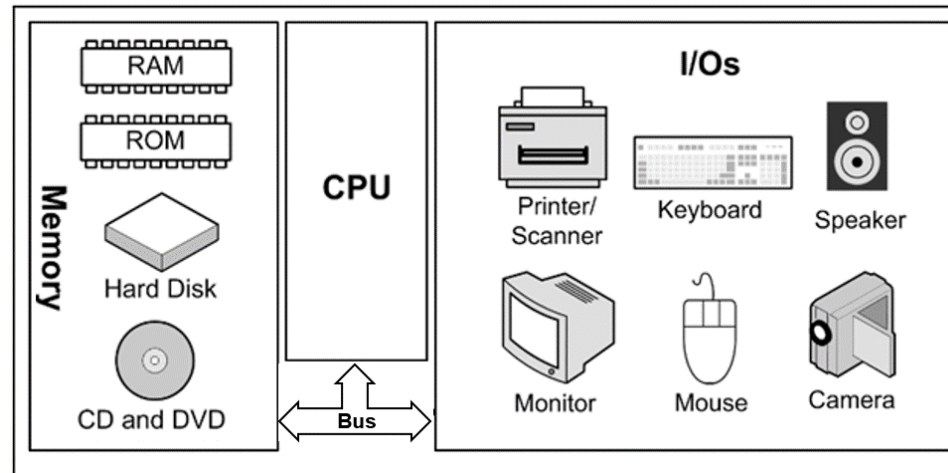
- ▶ We took a tour of the Canvas site for the course highlighting the features and links discusses in this presentation.
- ▶ We verified workstation access & availability of development tools.
  - ▶ Anaconda Python Distribution is available in the lab.
  - ▶ For installation on your own device, you should be able to get a copy of the community edition here: <https://www.anaconda.com/products/individual>
- ▶ If you are new to programming, I recommend using the Anaconda tools since your menus will match the slides and demonstrations shown in class.
- ▶ If you have programmed before and already have another Python environment, feel free to use that.

# Computers & Programs

Hardware, Software & Abstraction

# Computer Hardware

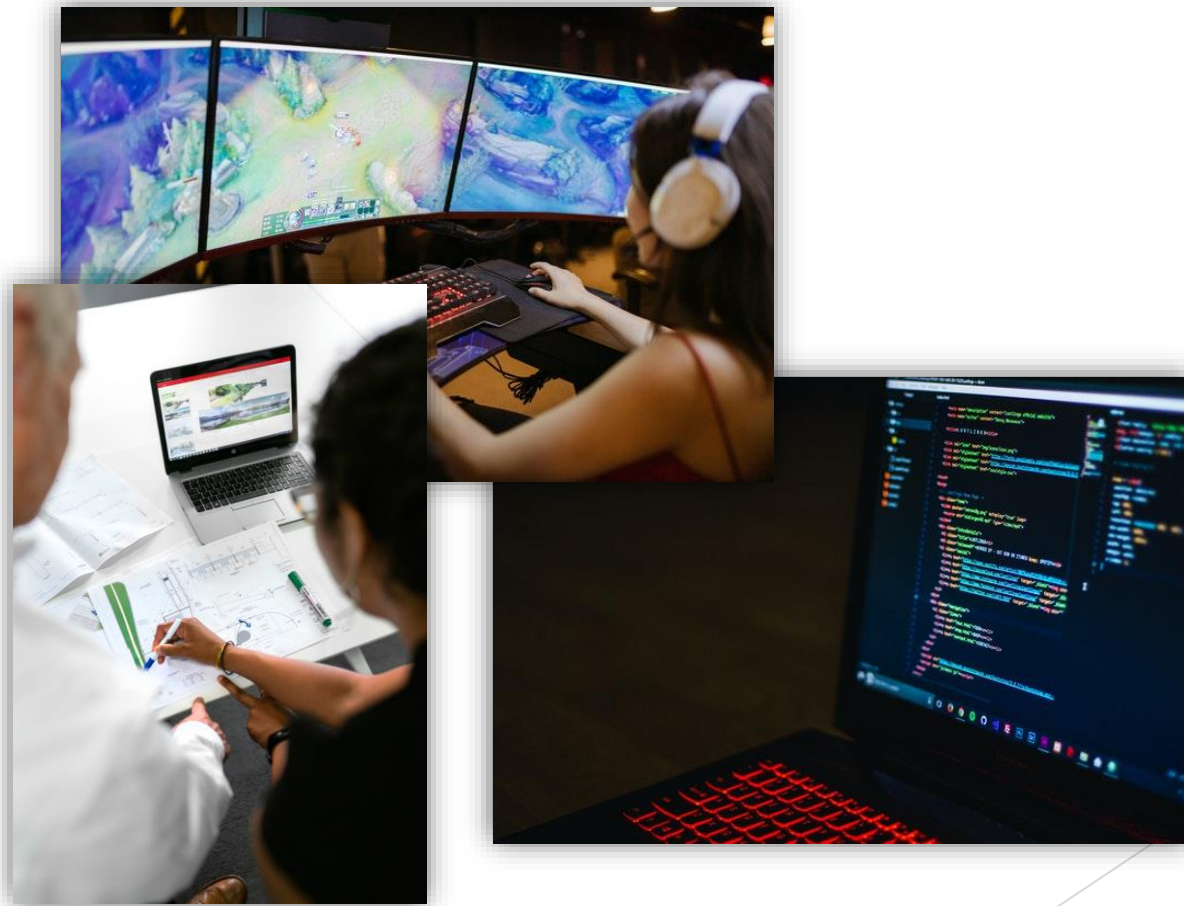
- ▶ Memory & Data Storage
- ▶ CPU
- ▶ I/O Devices, "Peripherals"
- ▶ Bus & Controllers, ASICs
- ▶ Sensors & Actuators



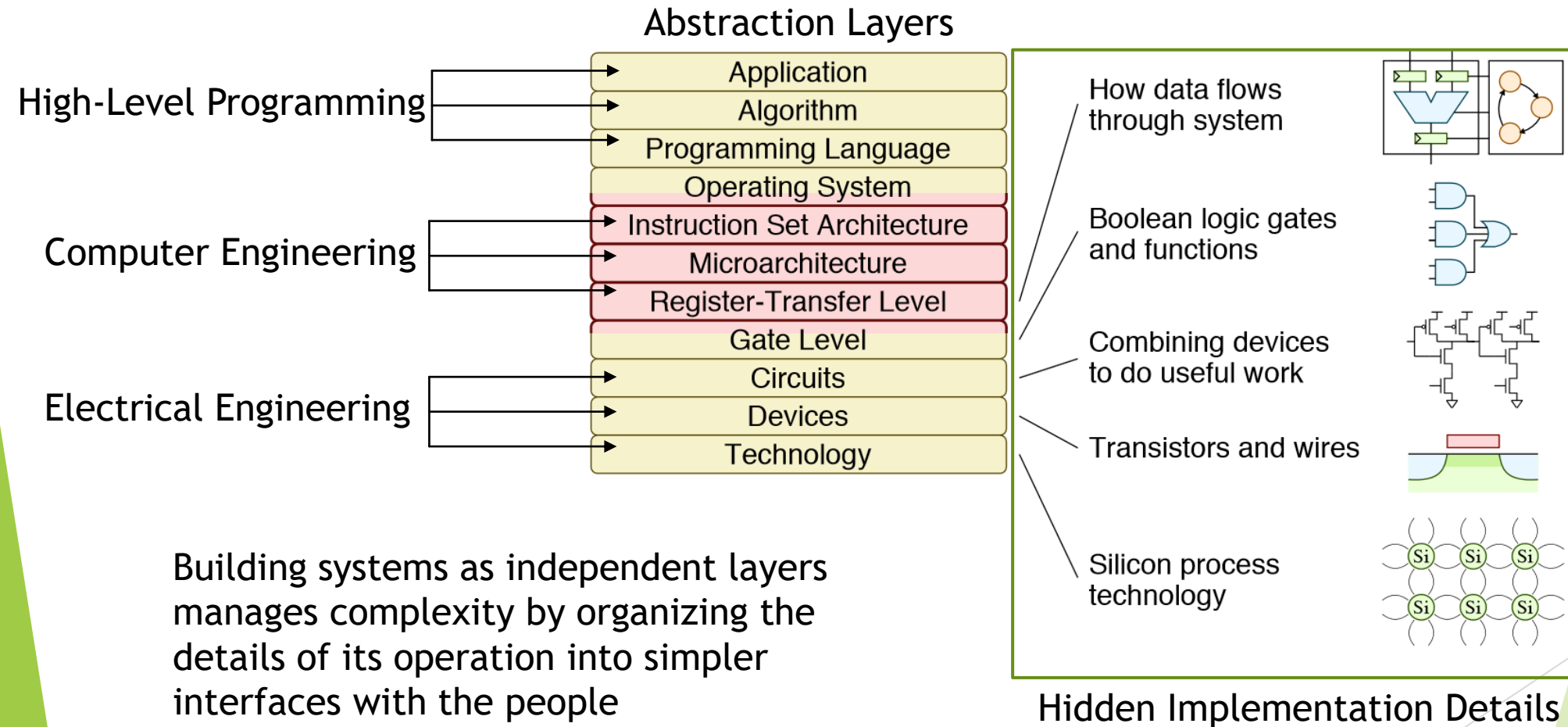


# Computer Software

- ▶ Applications
  - ▶ Games
  - ▶ Internet Browser
  - ▶ Office Tools
- ▶ Operating System
- ▶ I/O Device Drivers
- ▶ System Utilities
  - ▶ Antivirus
  - ▶ Disk Defrag
- ▶ Development Tools



# The Power of Abstraction

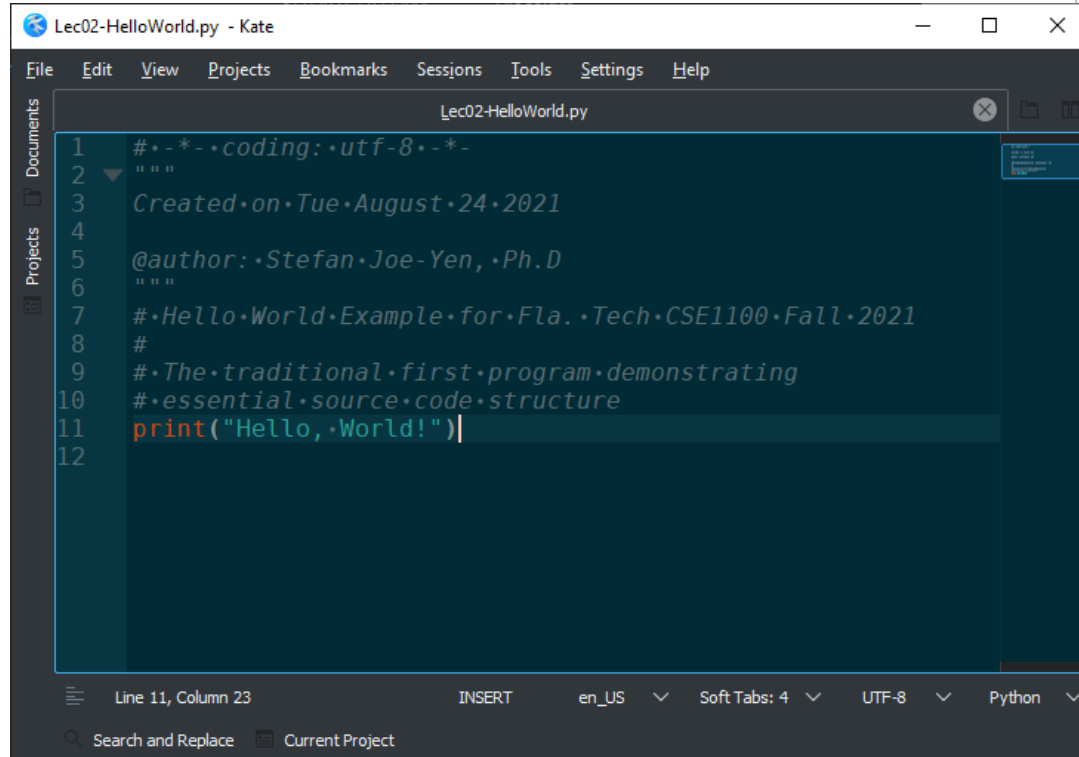


Building systems as independent layers manages complexity by organizing the details of its operation into simpler interfaces with the people (and other layers) that wish to use functionality assigned to that component.

# From Source Code to Executable

# Source Code

- ▶ Plain text file (no formatting)
- ▶ Use ASCII/Unicode Text Editor
- ▶ Syntax Highlighting
- ▶ IDE adds additional features
  - ▶ Code Completion
  - ▶ Syntax/Semantic Suggestions
  - ▶ Integrated Execution
  - ▶ Step-by-Step Debugger
  - ▶ Packaging & Deployment



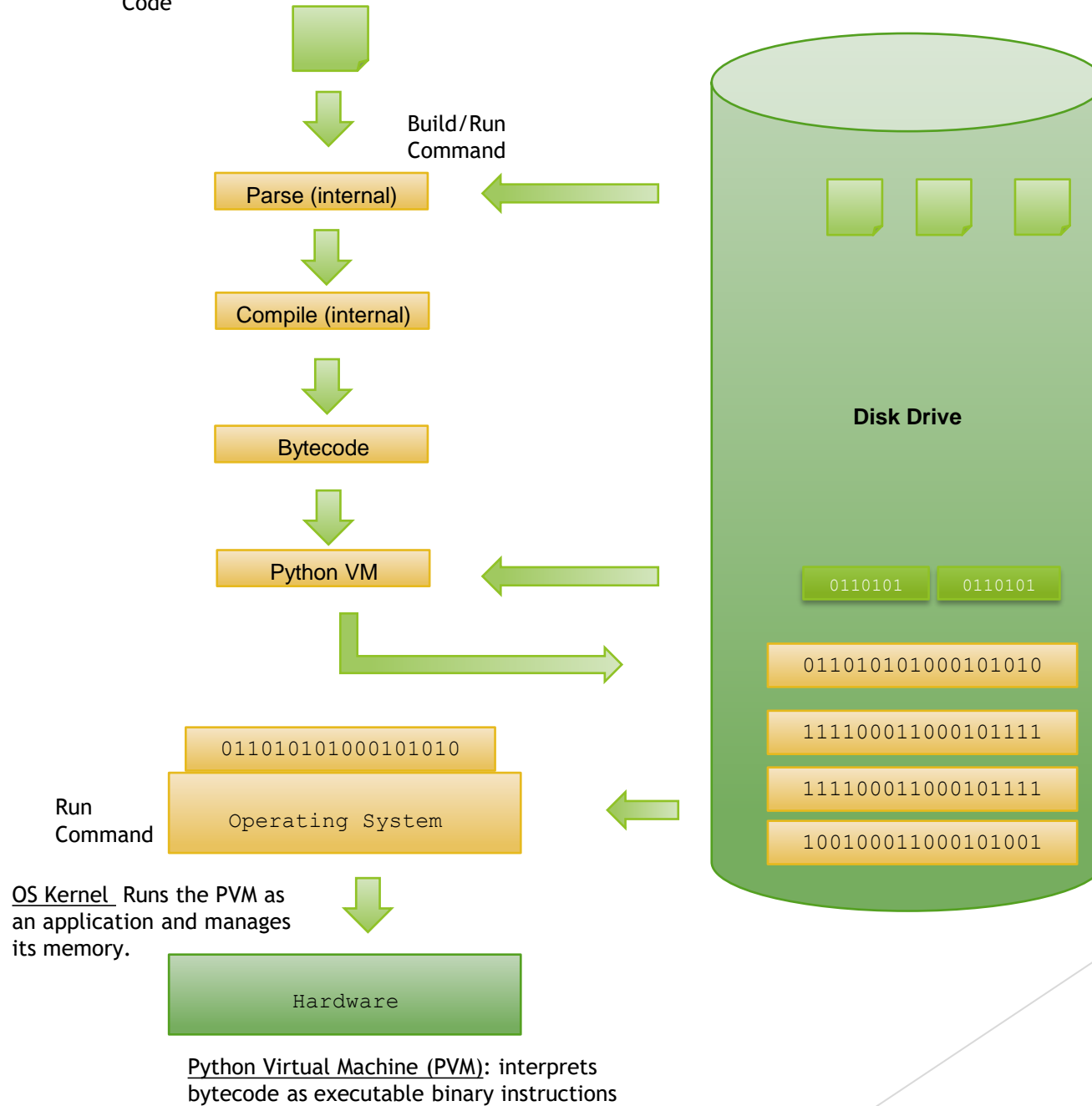
```
1  -*-*.coding: utf-8 -*-
2  """
3  Created on Tue August 24 2021
4
5  @author: Stefan Joe Yen, Ph.D
6  """
7  # Hello World Example for Fla. Tech CSE1100 Fall 2021
8  #
9  # The traditional first program demonstrating
10 # essential source code structure
11 print("Hello, World!")
12
```

# Creating a Runnable Program

- ▶ Creating an executable program from source code depends on the programming language. The process falls into 3 main Categories
  - ▶ Compiled
    - ▶ All the instructions in the source file are read, processed and turned into machine specific instructions (e.g. FORTRAN, C++).
  - ▶ Interpreted
    - ▶ The lines of code are translated into machine code and run one at a time by an program called the interpreter (e.g. LISP, PERL).
  - ▶ Hybrid/VM
    - ▶ The program is first translated into an intermediate language that is then executed by a program called the Virtual Machine that is customized to run on different hardware. (e.g. Python, Java)
    - ▶ VMs are typically optimized to run much faster than traditional interpreters.
- ▶ Program execution details are platform specific but a program is usually launched by:
    - ▶ Typing the name of the executable.
    - ▶ Clicking on the Desktop Icon using a mouse pointer on a Monitor
    - ▶ Tapping or Touching the Icon as displayed on a touch-screen
  - ▶ Any one of these recognized actions causes the OS to load the executable file from storage and proceed to execute the instructions (as scheduled)

- Information
- Software
- Hardware

Create & Edit Python Source Code

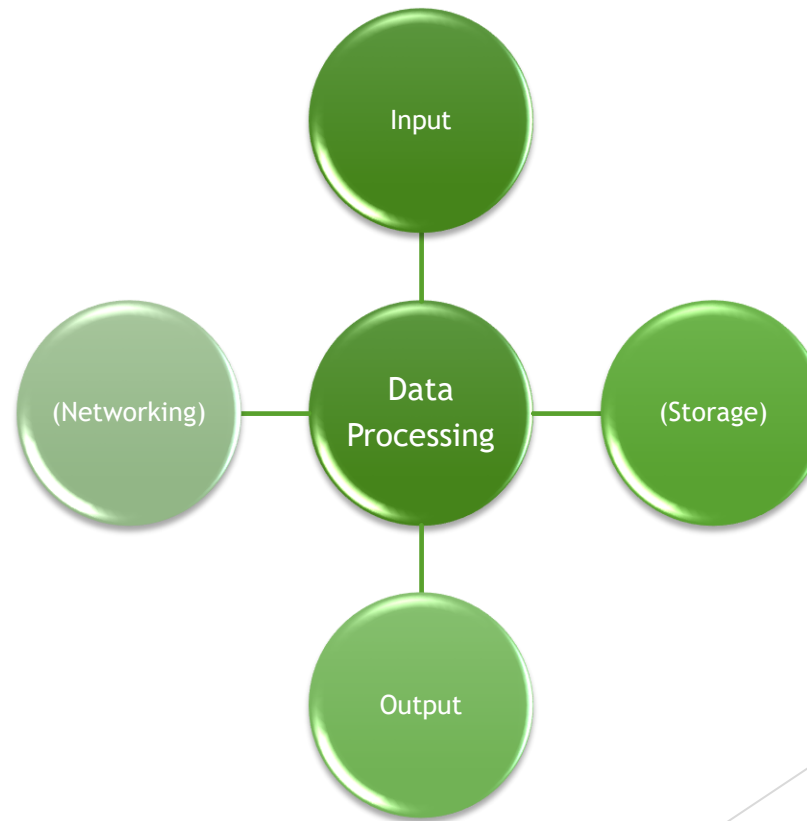


# The Bones of Software

General Software Responsibilities

# Common Software Elements

- ▶ While the specific structure of programs and software systems vary widely, most software performs a set similar functions.
- ▶ The functions can be viewed as connected sub-components that exchange data.
- ▶ Depending on the application:
  - ▶ The format of data is transformed from human-centric to machine-centric representations
  - ▶ Data exchanges between the sub-processes occur either sequentially or on-demand.
- ▶ Programming languages consist of text-based expressions that instruct the hardware to perform these tasks when the program is run.





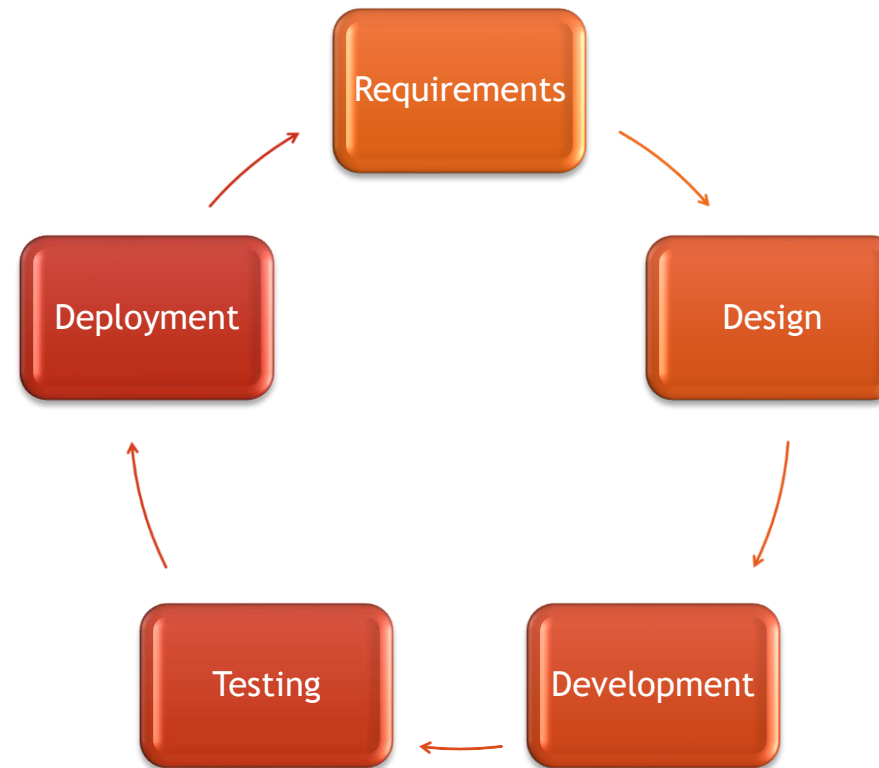
# The Software Development Lifecycle

(SDLC) - The lifespan of a computer software/system

# SDLC in a Nutshell

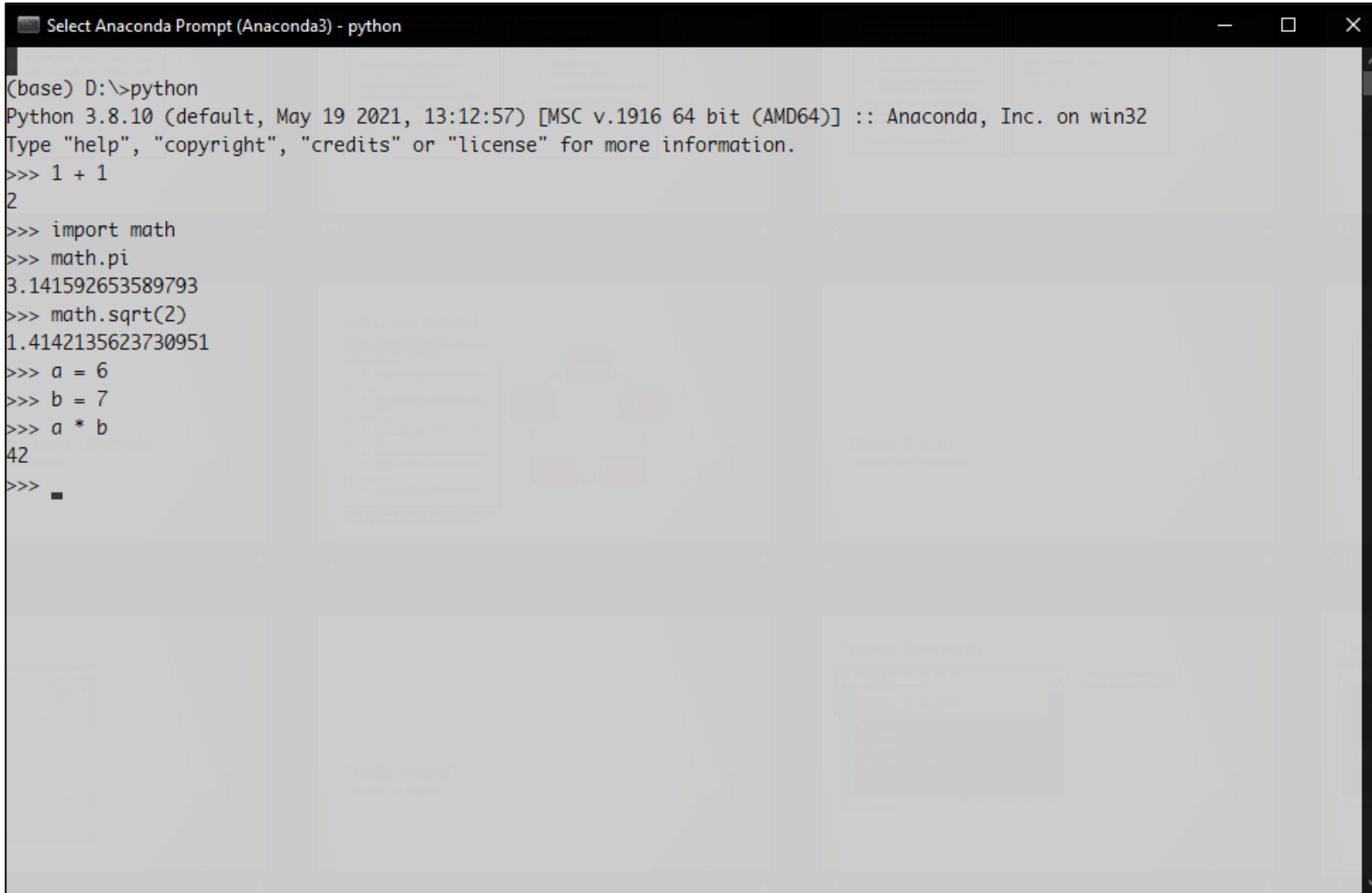
The SDLC has been studied for decades and the subtleties could form the basis of an entire course. These are the major components.

- ▶ **Requirements**
  - ▶ Determine the tasks the software-based system needs to accomplish.
- ▶ **Design**
  - ▶ Determine the “best” way for the software to accomplish its tasks and capture it into a design (plan).
- ▶ **Development**
  - ▶ Translate the design into an executable using a programming language.
- ▶ **Testing**
  - ▶ Verifying that the design requirements were met.
  - ▶ Validating that the design requirements were sound.
- ▶ **Deployment**
  - ▶ Making the completed software-based system available to the end user



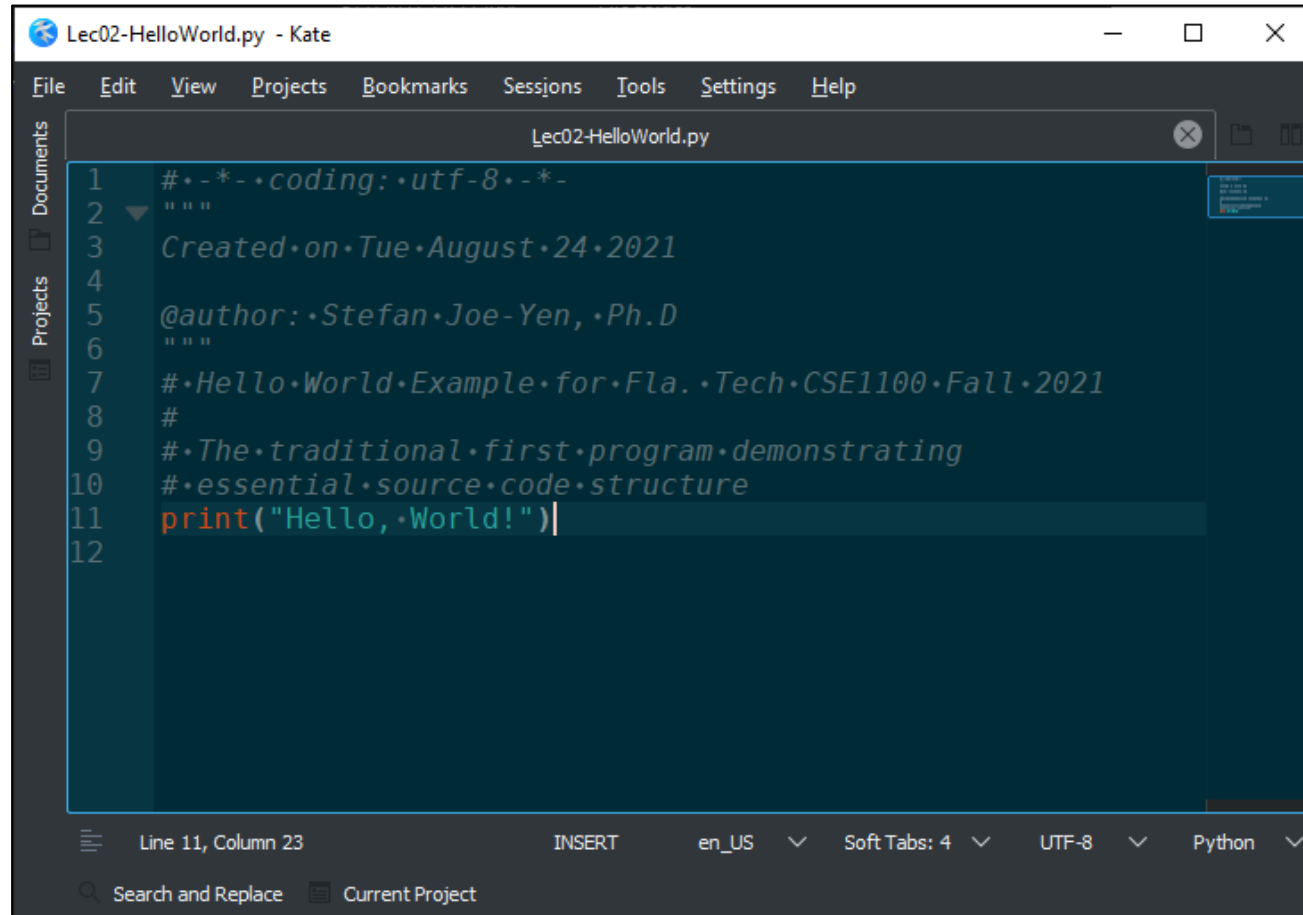
Issues discovered in Testing and Deployment lead to new Requirements and the cycle repeats.

# REPL (Read Evaluate Print Loop)

A screenshot of an Anaconda Prompt window titled "Select Anaconda Prompt (Anaconda3) - python". The window shows a Python 3.8.10 REPL session. The prompt is "(base) D:\>python". The output shows the Python version and architecture: "Python 3.8.10 (default, May 19 2021, 13:12:57) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32". The user enters several commands: "1 + 1" (output: 2), "import math", "math.pi" (output: 3.141592653589793), "math.sqrt(2)" (output: 1.4142135623730951), "a = 6", "b = 7", "a \* b" (output: 42), and finally a blank line with a cursor. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

```
(base) D:\>python
Python 3.8.10 (default, May 19 2021, 13:12:57) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 1 + 1
2
>>> import math
>>> math.pi
3.141592653589793
>>> math.sqrt(2)
1.4142135623730951
>>> a = 6
>>> b = 7
>>> a * b
42
>>> 
```

# Text Editor vs. IDE

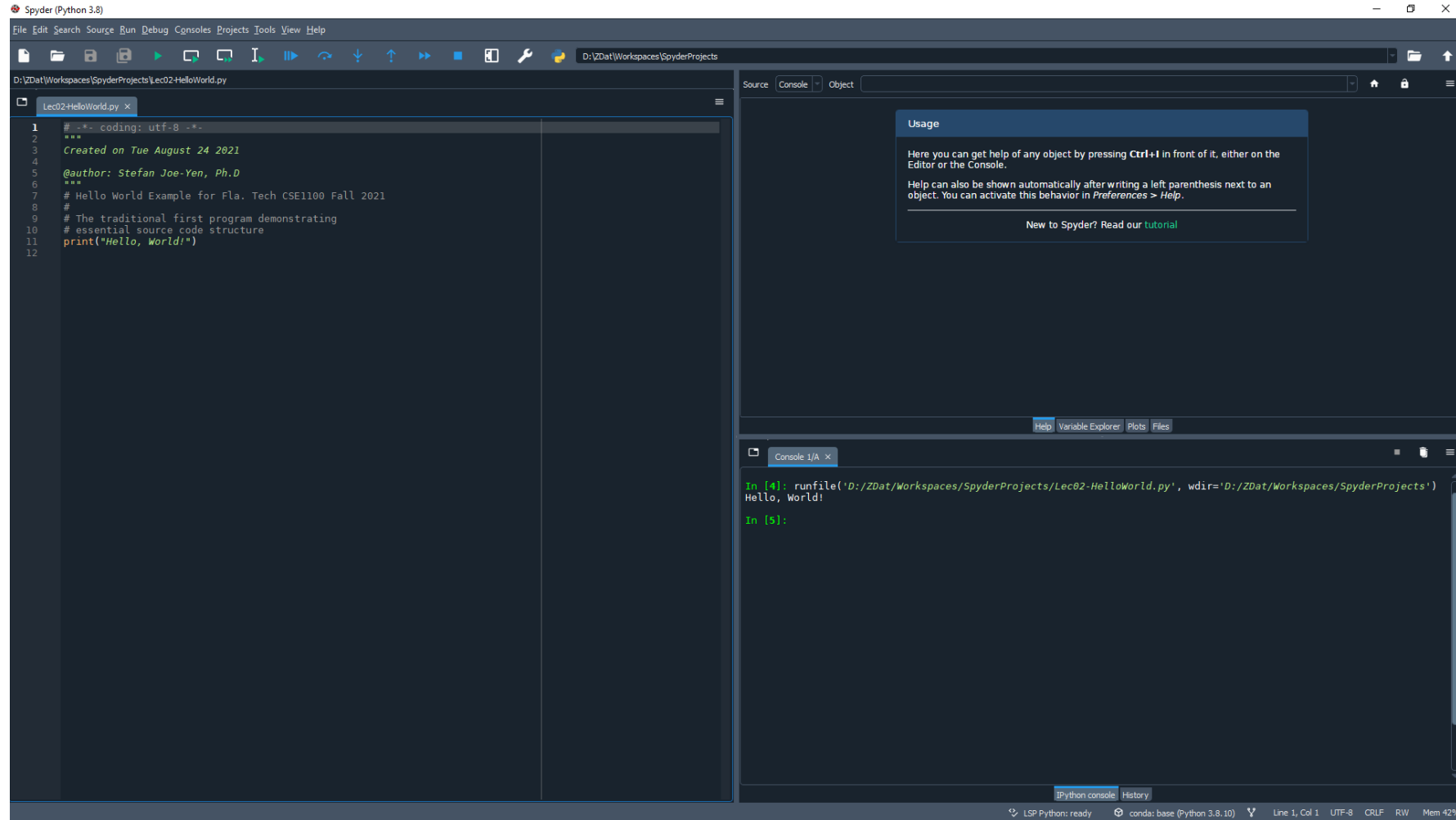


The screenshot shows the Kate text editor interface. The title bar reads 'Lec02-HelloWorld.py - Kate'. The menu bar includes 'File', 'Edit', 'View', 'Projects', 'Bookmarks', 'Sessions', 'Tools', 'Settings', and 'Help'. The left sidebar has 'Documents' and 'Projects' sections. The main editor area displays the following Python code:

```
1  #-*-coding:utf-8-*-
2  """
3  Created on Tue August 24 2021
4
5  @author: Stefan Joe-Yen, Ph.D
6  """
7  #Hello World Example for Fla. Tech CSE1100 Fall 2021
8  #
9  #The traditional first program demonstrating
10 #essential source code structure
11 print("Hello, World!")
12
```

The status bar at the bottom shows 'Line 11, Column 23', 'INSERT' mode, 'en\_US' locale, 'Soft Tabs: 4', 'UTF-8' encoding, and 'Python' language. A search bar at the bottom left contains 'Search and Replace' and 'Current Project'.

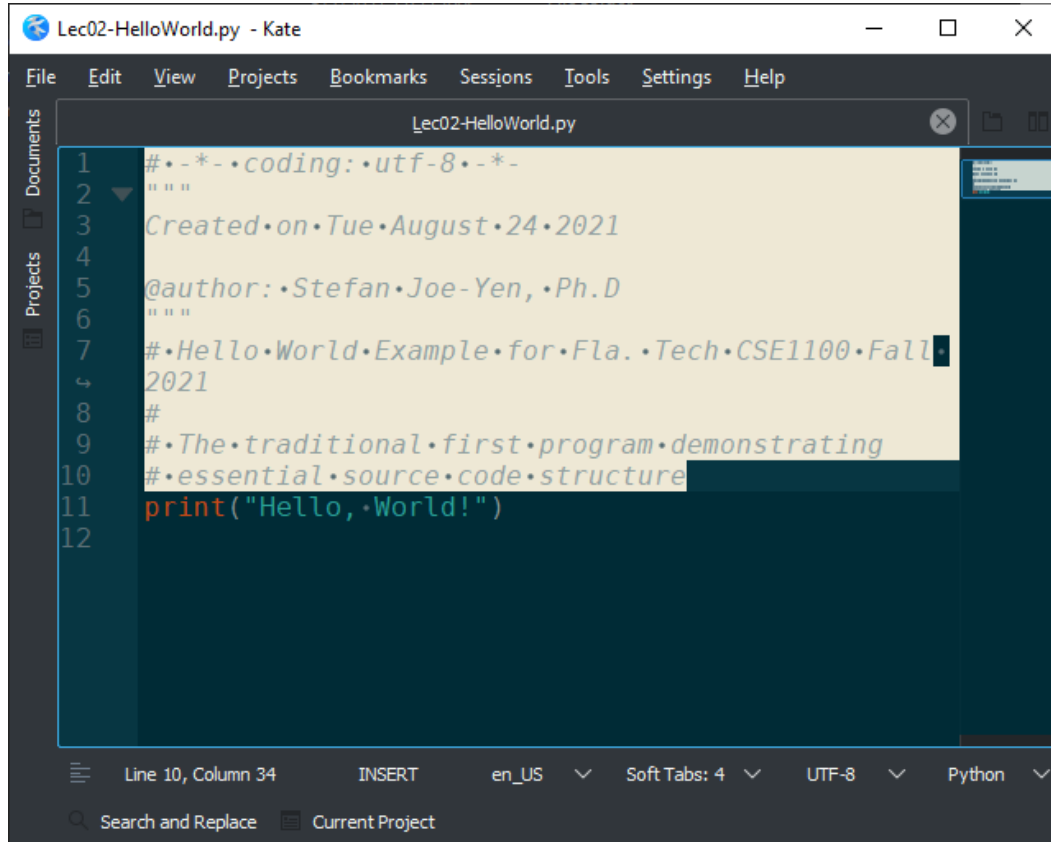
# Text Editor vs. IDE



# "Hello World"

The Traditional First Program

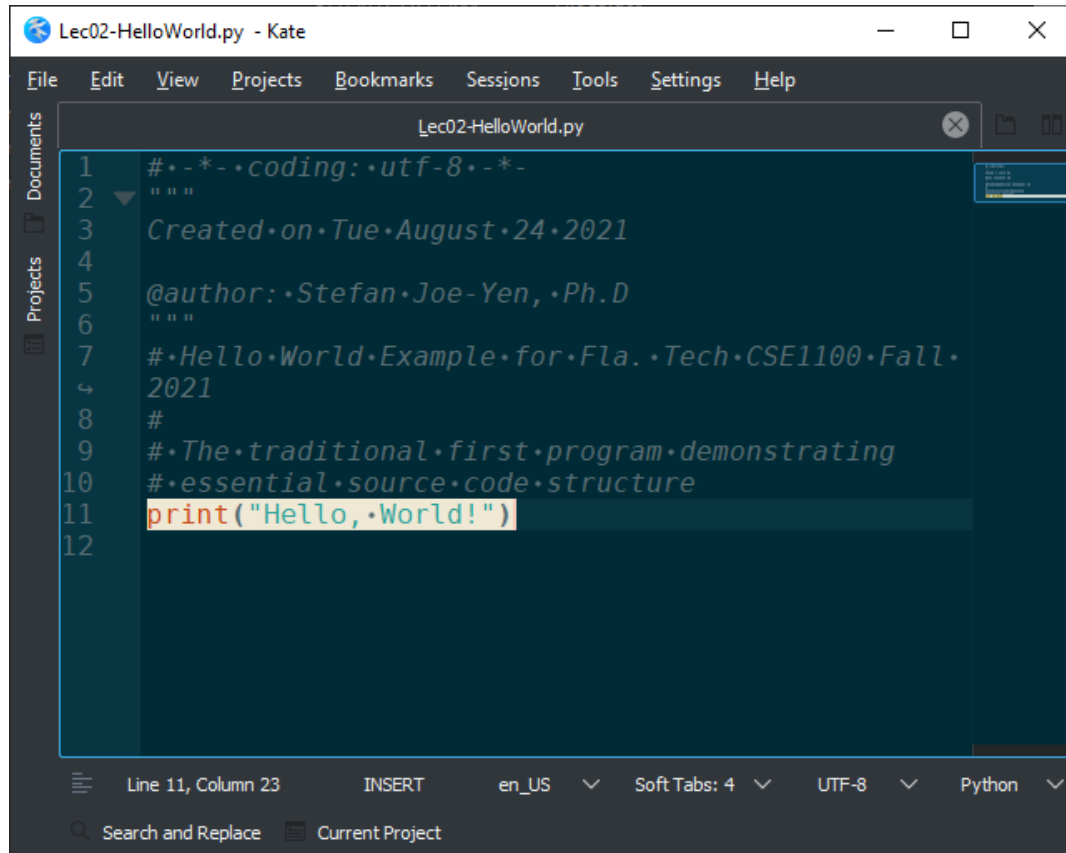
# Program Comments



```
1  -*- coding: utf-8 -*-
2  """
3  Created on Tue August 24 2021
4
5  @author: Stefan Joe-Yen, Ph.D
6  """
7  # Hello World Example for Fla. Tech CSE1100 Fall
8  2021
9  #
10 # The traditional first program demonstrating
11 # essential source code structure
12 print("Hello, World!")
```

- # Single Line Comments
- """ Multi-Line Comment """

# The program body (block structure)



The screenshot shows a code editor window titled "Lec02-HelloWorld.py - Kate". The editor contains the following code:

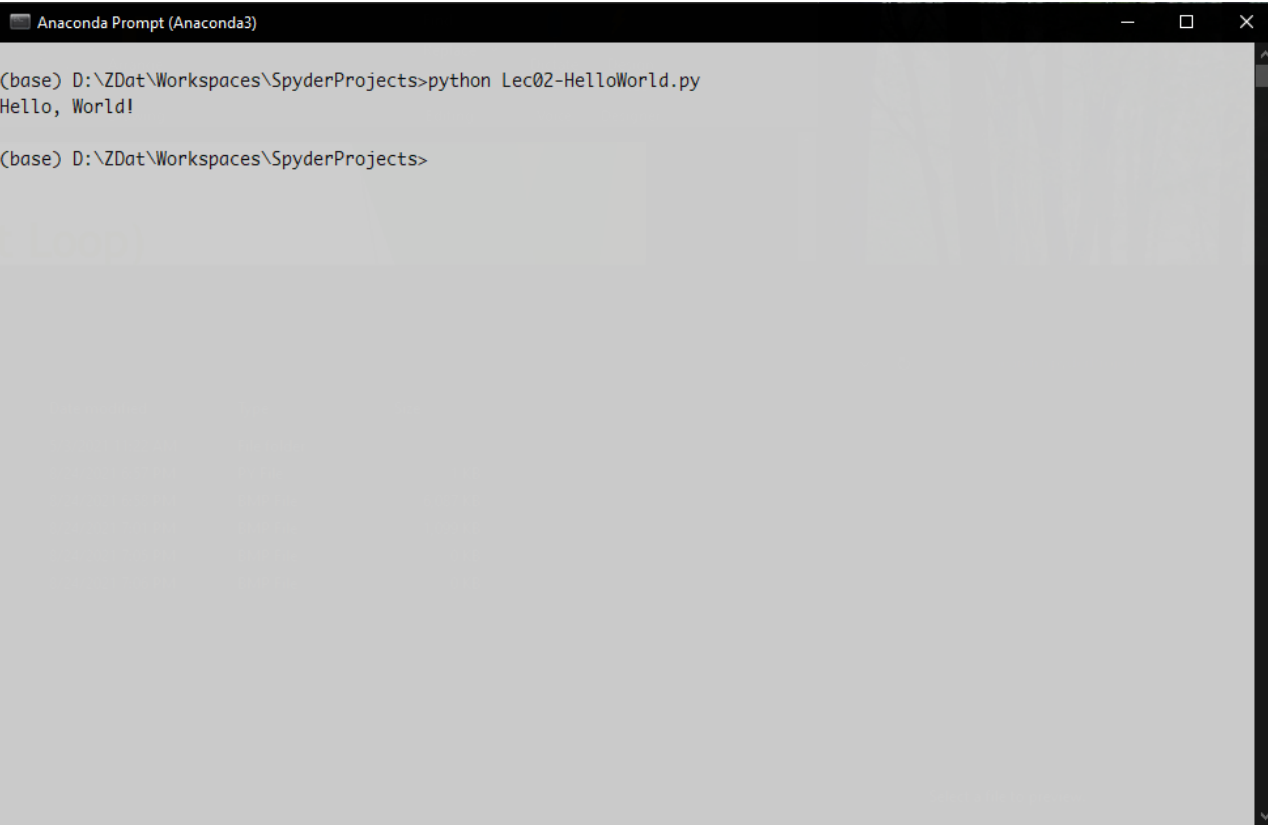
```
1  -*- coding: utf-8 -*-
2  """
3  Created on Tue August 24 2021
4
5  @author: Stefan Joe-Yen, Ph.D
6  """
7  # Hello World Example for Fla. Tech CSE1100 Fall
8  # 2021
9  # The traditional first program demonstrating
10 # essential source code structure
11 print("Hello, World!")
12
```

The code is written in a dark-themed editor. The line numbers are on the left. The code is a Python script with a docstring and a single line of code forming the body. The status bar at the bottom shows "Line 11, Column 23", "INSERT", "en\_US", "Soft Tabs: 4", "UTF-8", and "Python".

- Blocks of code in Python are not denoted by any special punctuation (e.g. curly braces { })
- This single line of code forms the entire body of the program
- Lines are not terminated by any special punctuation (e.g. ;)
- The program is executed in the IDE console as shown previously or passed to the Python interpreter as shown next.



# Console output

A screenshot of an Anaconda Prompt window. The title bar reads "Anaconda Prompt (Anaconda3)". The command prompt shows the command `python Lec02-HelloWorld.py` being executed, followed by the output `Hello, World!`. The prompt then returns to `(base) D:\ZDat\Workspaces\SpyderProjects>`.

```
(base) D:\ZDat\Workspaces\SpyderProjects>python Lec02-HelloWorld.py
Hello, World!

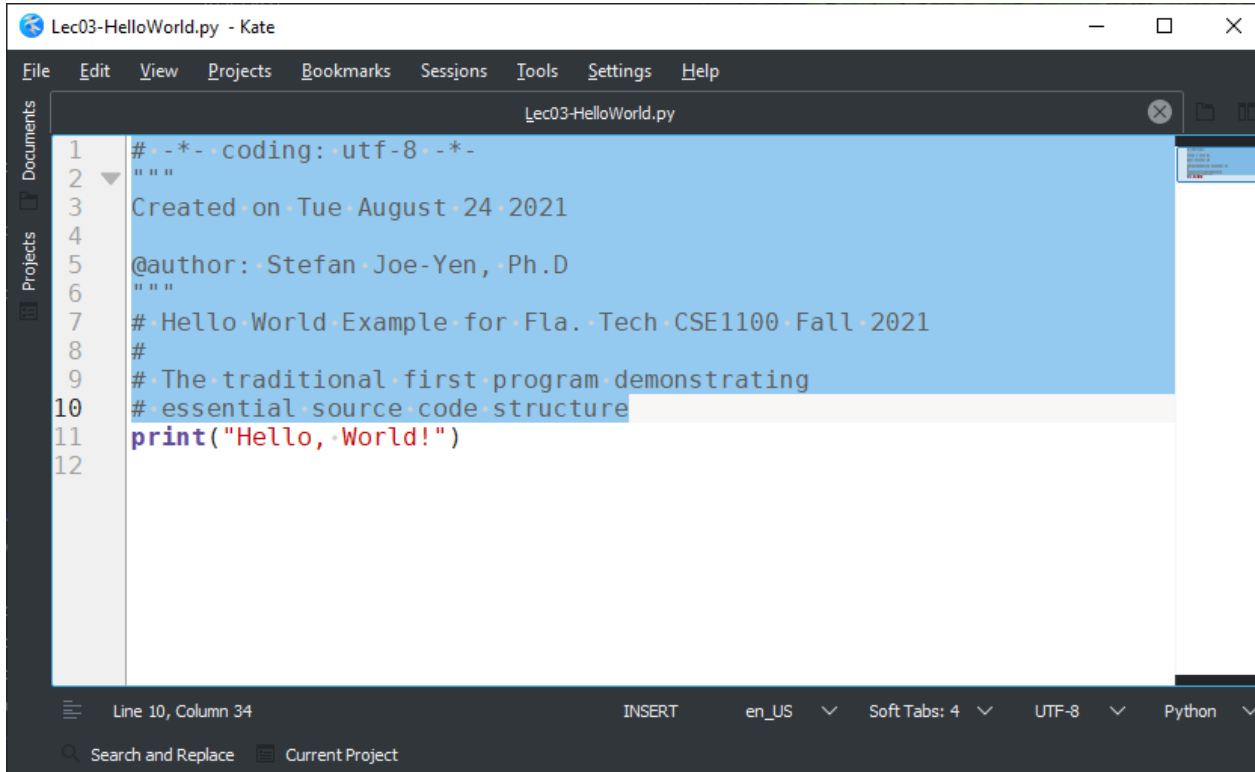
(base) D:\ZDat\Workspaces\SpyderProjects>
```

To execute your program from the command line, type *python myfilename.py* at the console prompt - replacing *myfilename.py* with the actual name of your script.

# "Hello World"

The Traditional First Program

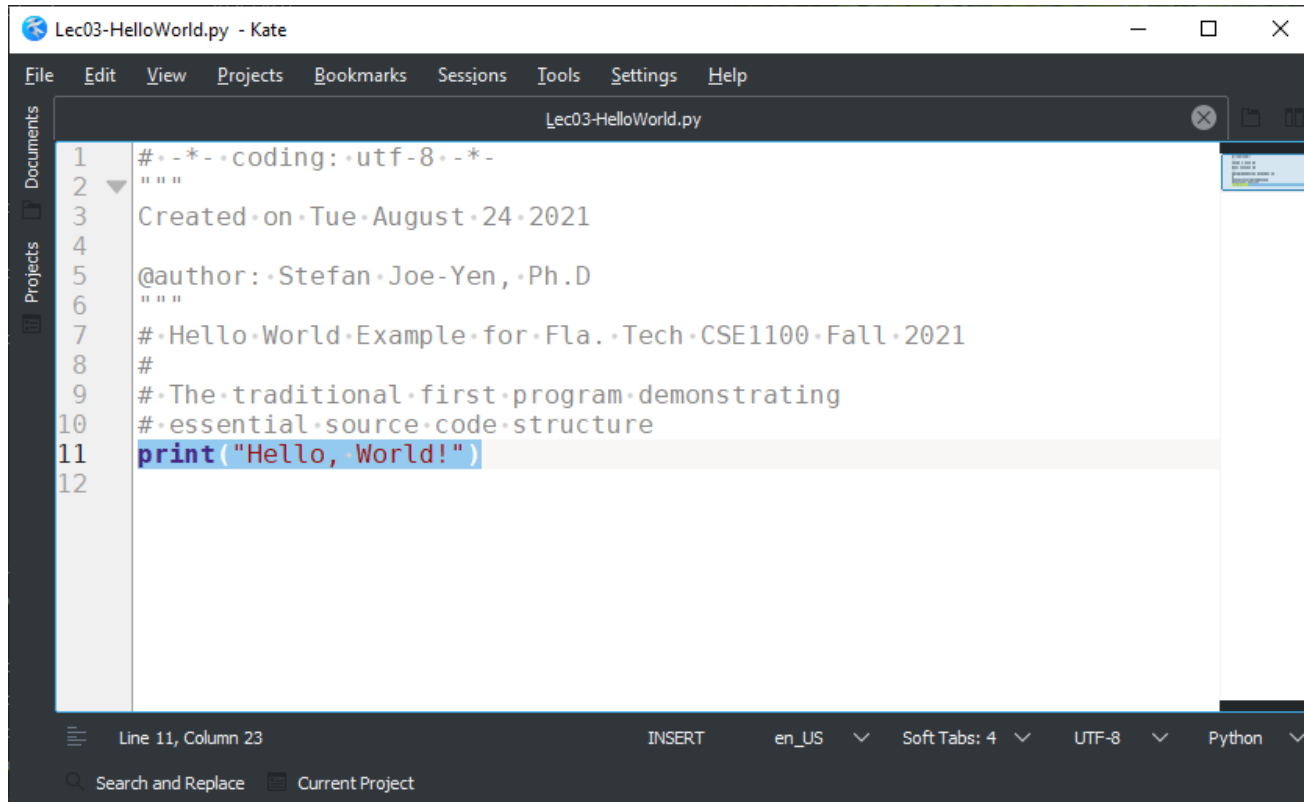
# Program Comments



```
1  -*- coding: utf-8 -*-
2  """
3  Created on Tue August 24 2021
4
5  @author: Stefan Joe Yen, Ph.D
6  """
7  # Hello World Example for Fla. Tech CSE1100 Fall 2021
8  #
9  # The traditional first program demonstrating
10 # essential source code structure
11 print("Hello, World!")
12
```

- ▶ # Single Line Comments
- ▶ """ Multi-Line Comment """

# The program body (block structure)

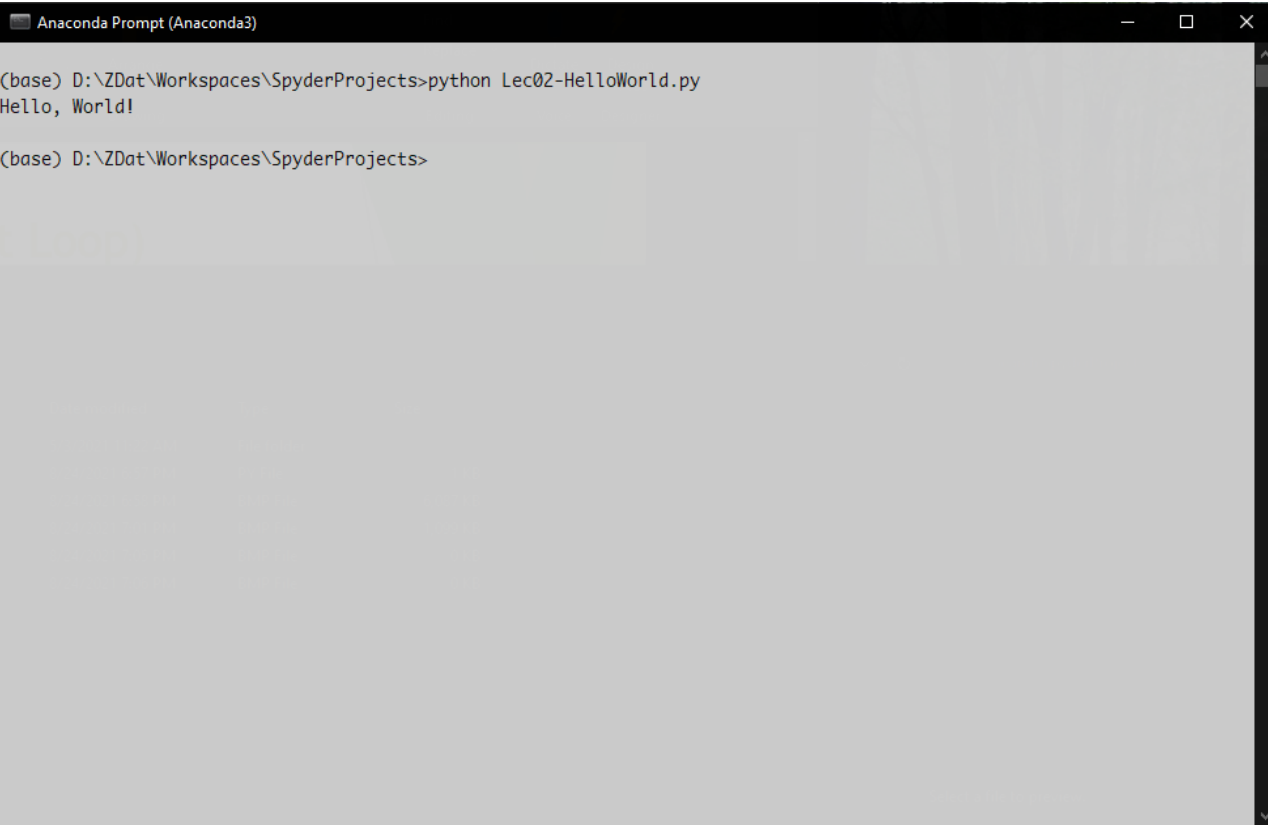
A screenshot of a code editor window titled 'Lec03-HelloWorld.py - Kate'. The editor shows a Python file with the following content:

```
1  -*- coding: utf-8 -*-
2  """
3  Created on Tue August 24 2021
4
5  @author: Stefan Joe-Yen, Ph.D
6  """
7  # Hello World Example for Fla. Tech CSE1100 Fall 2021
8  #
9  # The traditional first program demonstrating
10 # essential source code structure
11 print("Hello, World!")
12
```

The line number indicator on the left shows lines 1 through 12. The status bar at the bottom indicates 'Line 11, Column 23', 'INSERT' mode, 'en\_US' locale, 'Soft Tabs: 4', 'UTF-8' encoding, and 'Python' interpreter.

- ▶ Blocks of code in Python are not denoted by any special punctuation (e.g. curly braces { })
- ▶ This single line of code forms the entire body of the program
- ▶ Lines are not terminated by any special punctuation (e.g. ;)
- ▶ The program is executed in the IDE console as shown previously or passed to the Python interpreter as shown next.

# Console output

A screenshot of an Anaconda Prompt window. The title bar reads "Anaconda Prompt (Anaconda3)". The command prompt shows the command `python Lec02-HelloWorld.py` being executed, followed by the output `Hello, World!`. The prompt then returns to `(base) D:\ZDat\Workspaces\SpyderProjects>`.

```
(base) D:\ZDat\Workspaces\SpyderProjects>python Lec02-HelloWorld.py
Hello, World!

(base) D:\ZDat\Workspaces\SpyderProjects>
```

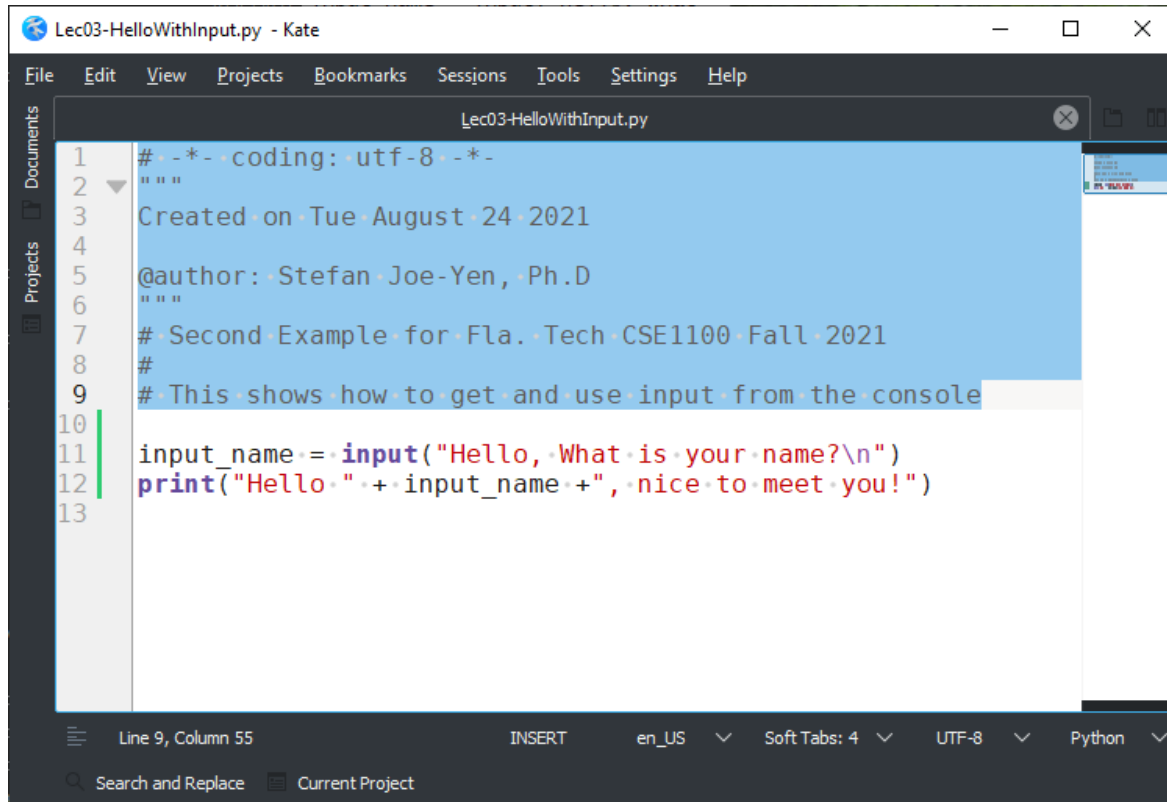
To execute your program from the command line, type *python myfilename.py* at the console prompt - replacing *myfilename.py* with the actual name of your script.

The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern, layered effect on the right side of the slide.

# "Hello <Your Name Here>"

The Second Python Program

# Always start with comments...



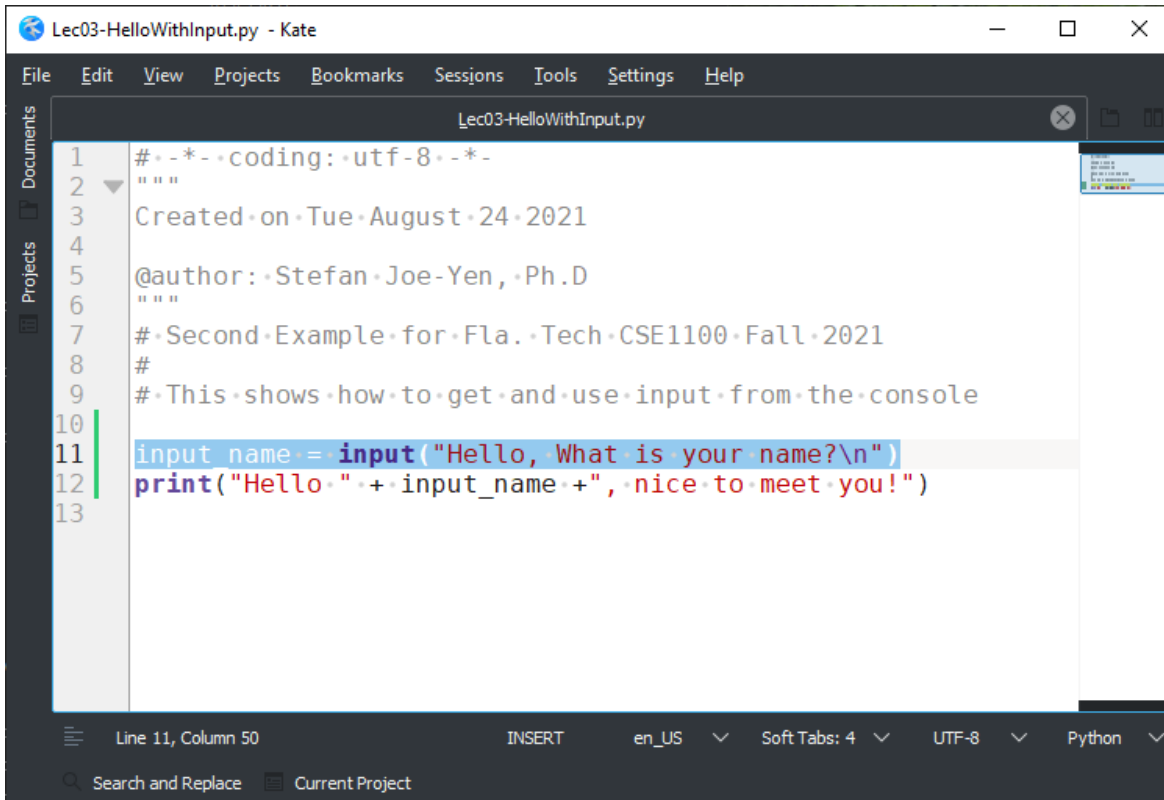
The screenshot shows a code editor window titled "Lec03-HelloWithInput.py - Kate". The editor displays a Python script with a docstring template. The docstring is a multi-line string enclosed in triple quotes, containing the following text:

```
1  #-*-coding:utf-8-*-
2  """
3  Created on Tue August 24 2021
4
5  @author: Stefan Joe-Yen, Ph.D
6  """
7  # Second Example for Fla. Tech CSE1100 Fall 2021
8  #
9  # This shows how to get and use input from the console
10
11  input_name = input("Hello, What is your name?\n")
12  print("Hello " + input_name + ", nice to meet you!")
13
```

The status bar at the bottom of the editor shows "Line 9, Column 55", "INSERT", "en\_US", "Soft Tabs: 4", "UTF-8", and "Python".

- ▶ Name
- ▶ Date created
- ▶ Purpose
- ▶ Modifications
- ▶ Some tools automatically generate this information based on a template

# Obtaining and storing input



The screenshot shows a code editor window titled "Lec03-HelloWithInput.py - Kate". The editor contains a Python script with the following code:

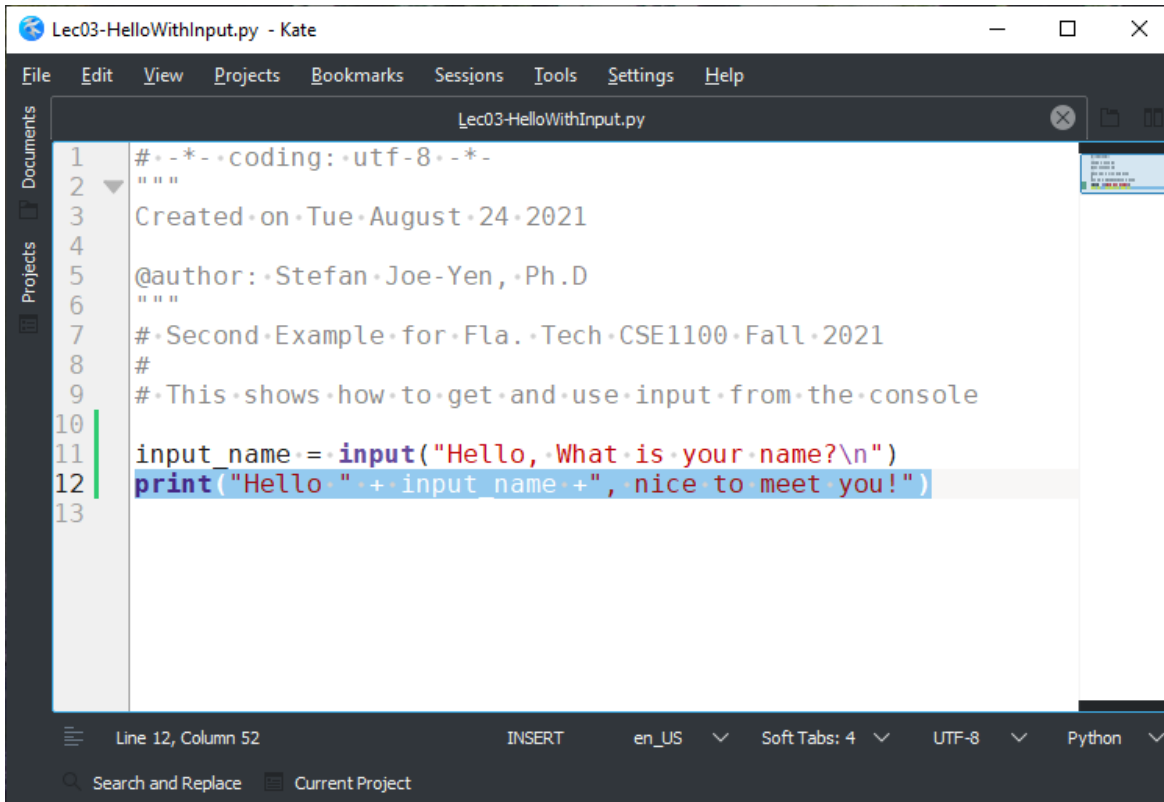
```
1  #-*-coding:utf-8-*-
2  """
3  Created on Tue August 24 2021
4
5  @author: Stefan Joe-Yen, Ph.D
6  """
7  # Second Example for Fla. Tech CSE1100 Fall 2021
8  #
9  # This shows how to get and use input from the console
10
11  input_name = input("Hello, What is your name?\n")
12  print("Hello " + input_name + ", nice to meet you!")
13
```

The code is written in a dark-themed editor. Line 11 is highlighted in blue. The status bar at the bottom indicates "Line 11, Column 50", "INSERT", "en\_US", "Soft Tabs: 4", "UTF-8", and "Python".

- ▶ The *input* function prints a message of your choice and then reads whatever is typed at the keyboard until the user hits "enter".
- ▶ This data is assigned to a named storage area called a variable using the = operator (read "gets").
- ▶ In the next lecture we will study variables in detail. Think of them as labeled boxes to store data. In this case the variable is named *input\_name*



# Using the stored data

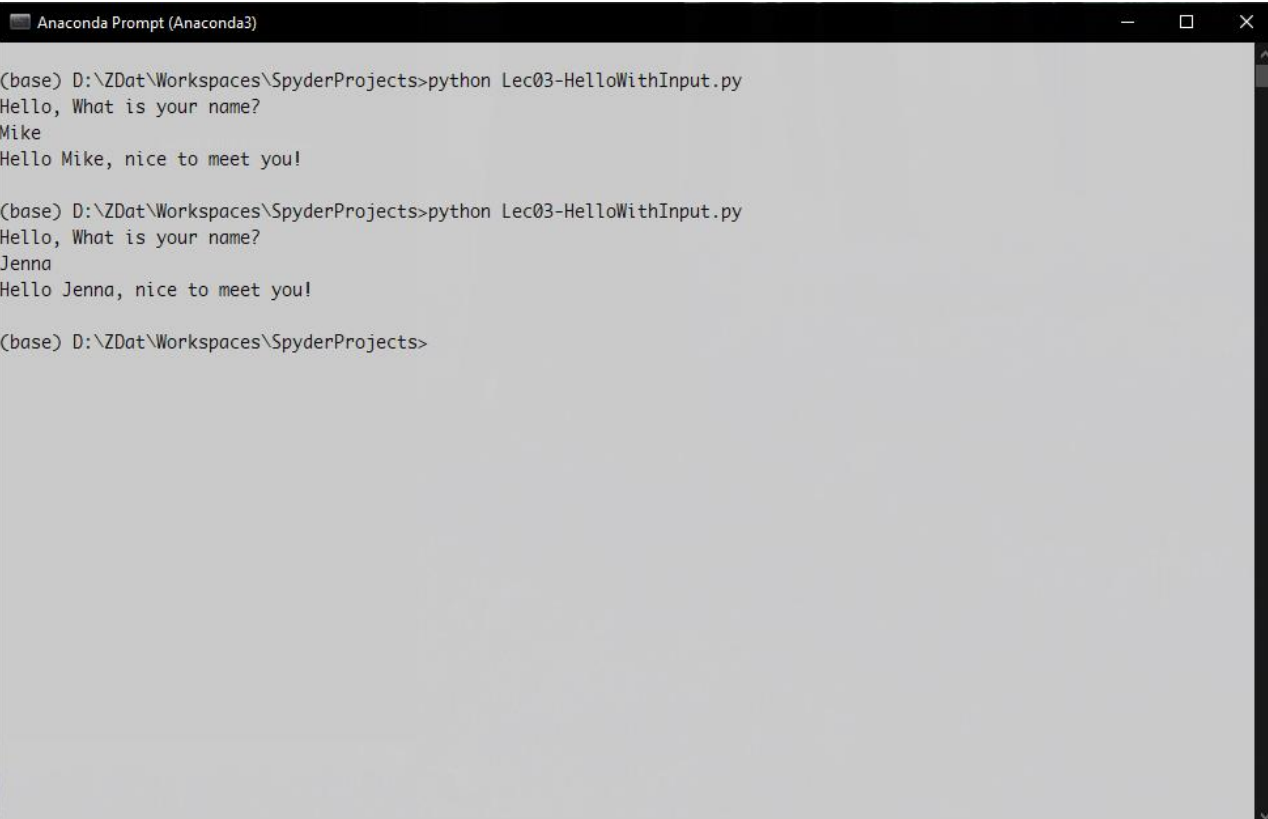


```
Lec03-HelloWithInput.py - Kate
File Edit View Projects Bookmarks Sessions Tools Settings Help
Lec03-HelloWithInput.py
1 #-*-coding:utf-8-*-
2 """
3 Created on Tue Aug 24 2021
4
5 @author: Stefan Joe-Yen, Ph.D
6 """
7 # Second Example for Fla. Tech CSE1100 Fall 2021
8 #
9 # This shows how to get and use input from the console
10
11 input_name = input("Hello, What is your name?\n")
12 print("Hello " + input_name + ", nice to meet you!")
13

Line 12, Column 52
INSERT en_US Soft Tabs: 4 UTF-8 Python
Search and Replace Current Project
```

- ▶ The last line produces the output using the data stored in *input\_name*.
- ▶ This data is integrated with fixed text by using `+` as a concatenation operator
- ▶ In the next few lectures, we will study operators in detail.

# Console output

A screenshot of an Anaconda Prompt window titled "Anaconda Prompt (Anaconda3)". The window shows the execution of a Python script named "Lec03-HelloWithInput.py". The prompt is "(base) D:\ZDat\Workspaces\SpyderProjects>". The script prompts the user for their name: "Hello, What is your name?". The user enters "Mike", and the script outputs "Hello Mike, nice to meet you!". The user then enters "Jenna", and the script outputs "Hello Jenna, nice to meet you!". The prompt returns to "(base) D:\ZDat\Workspaces\SpyderProjects>".

```
(base) D:\ZDat\Workspaces\SpyderProjects>python Lec03-HelloWithInput.py
Hello, What is your name?
Mike
Hello Mike, nice to meet you!

(base) D:\ZDat\Workspaces\SpyderProjects>python Lec03-HelloWithInput.py
Hello, What is your name?
Jenna
Hello Jenna, nice to meet you!

(base) D:\ZDat\Workspaces\SpyderProjects>
```

As before, execute your program from the command line by typing *python myfilename.py* at the console prompt - replacing *myfilename.py* with the actual name of your script.

# Sounds great! What could go wrong?

- ▶ A Lot!
  - ▶ We did no checking to make sure the input was valid syntactically or semantically
  - ▶ Depending on the build/execution environment, the console:
    - ▶ Can be hard to find/Non-interactive
    - ▶ May appear only briefly on the screen
    - ▶ May need to be configured (e.g. in an IDE)
    - ▶ May need the console input specified separately in another GUI element.
  - ▶ Input buffering (which will be covered in detail later) & non-printing characters can cause unexpected behavior.

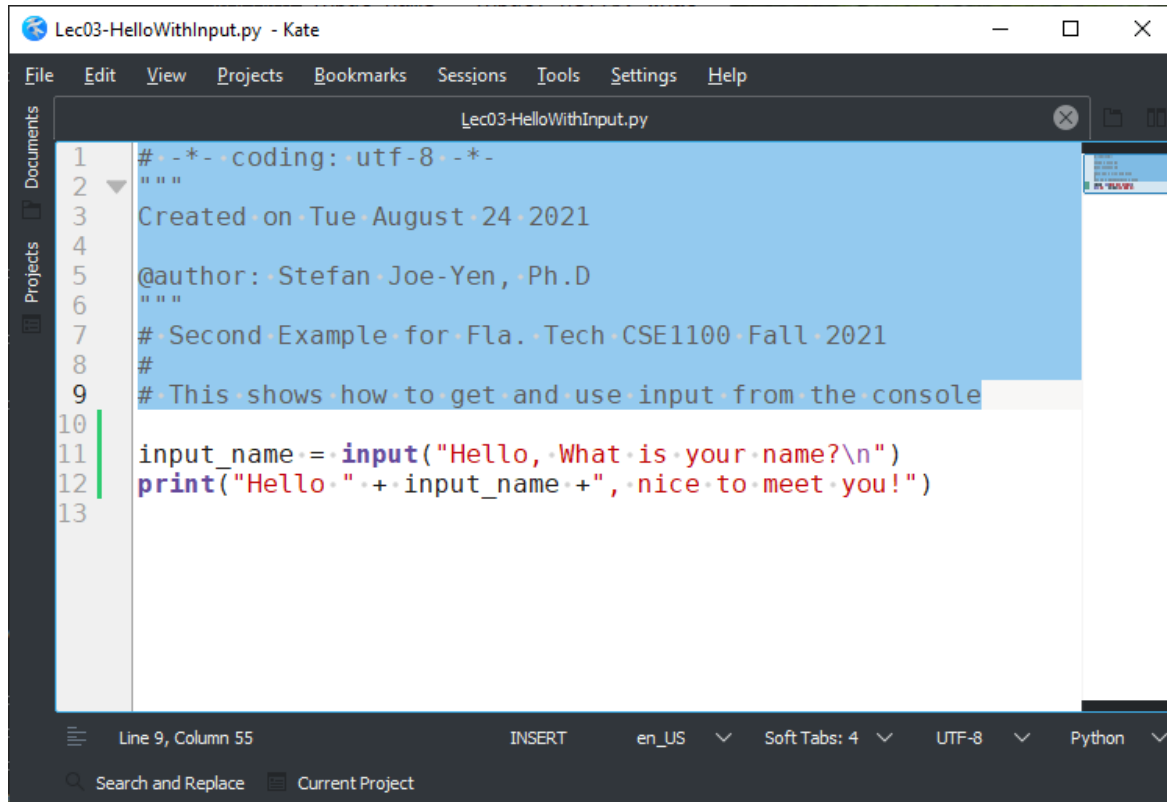
# Simple I/O Program (Review)

The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern and dynamic look. The shapes are concentrated on the right side and bottom of the slide, with some extending towards the left.

# "Hello <Your Name Here>"

The Second Python Program

# Always start with comments...

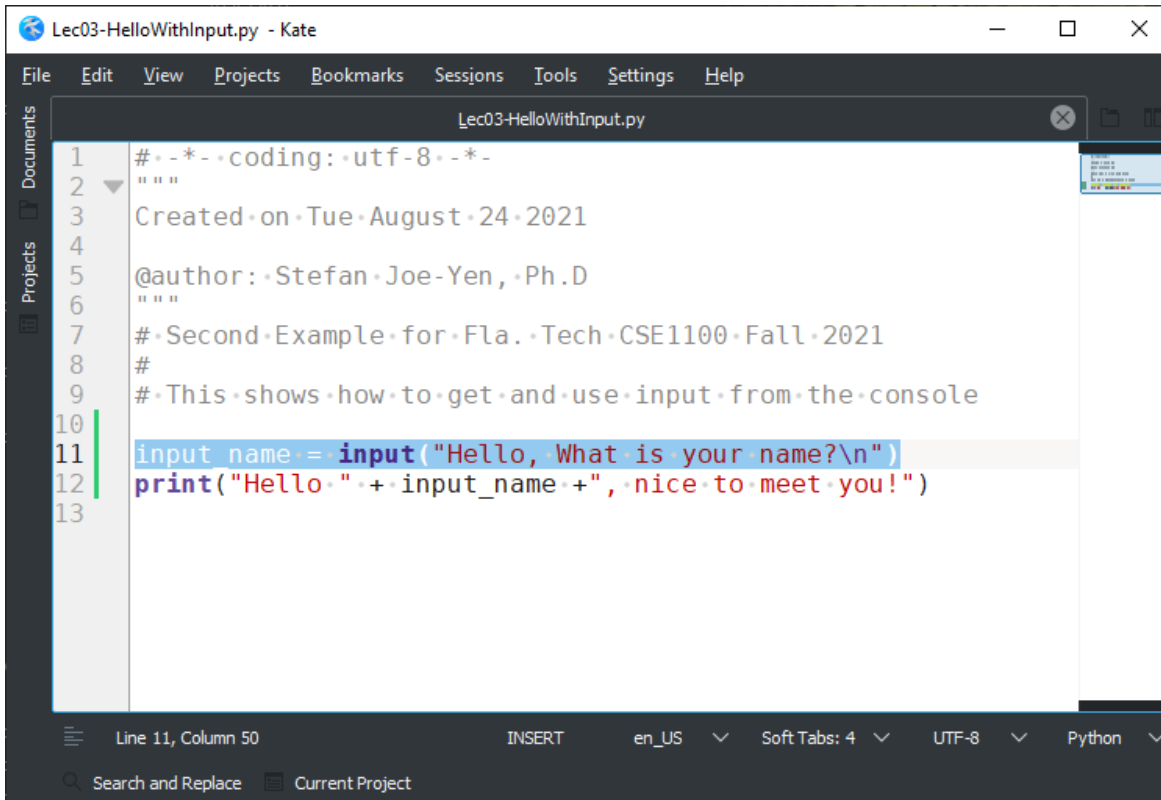


The screenshot shows a code editor window titled "Lec03-HelloWithInput.py - Kate". The editor displays a Python script with a multi-line comment block at the top. The comment block includes a docstring with the following text: "Created on Tue August 24 2021", "@author: Stefan Joe-Yen, Ph.D", "# Second Example for Fla. Tech CSE1100 Fall 2021", and "# This shows how to get and use input from the console". The code block starts on line 10 with "input\_name = input("Hello, What is your name?\n")" and ends on line 12 with "print("Hello " + input\_name + ", nice to meet you!")". The editor interface includes a menu bar (File, Edit, View, Projects, Bookmarks, Sessions, Tools, Settings, Help), a sidebar with "Documents" and "Projects" views, and a status bar at the bottom showing "Line 9, Column 55", "INSERT", "en\_US", "Soft Tabs: 4", "UTF-8", and "Python".

```
1 #-*- coding: utf-8 -*-
2 """
3 Created on Tue August 24 2021
4
5 @author: Stefan Joe-Yen, Ph.D
6 """
7 # Second Example for Fla. Tech CSE1100 Fall 2021
8 #
9 # This shows how to get and use input from the console
10
11 input_name = input("Hello, What is your name?\n")
12 print("Hello " + input_name + ", nice to meet you!")
13
```

- ▶ Name
- ▶ Date created
- ▶ Purpose
- ▶ Modifications
- ▶ Some tools automatically generate this information based on a template

# Obtaining and storing input



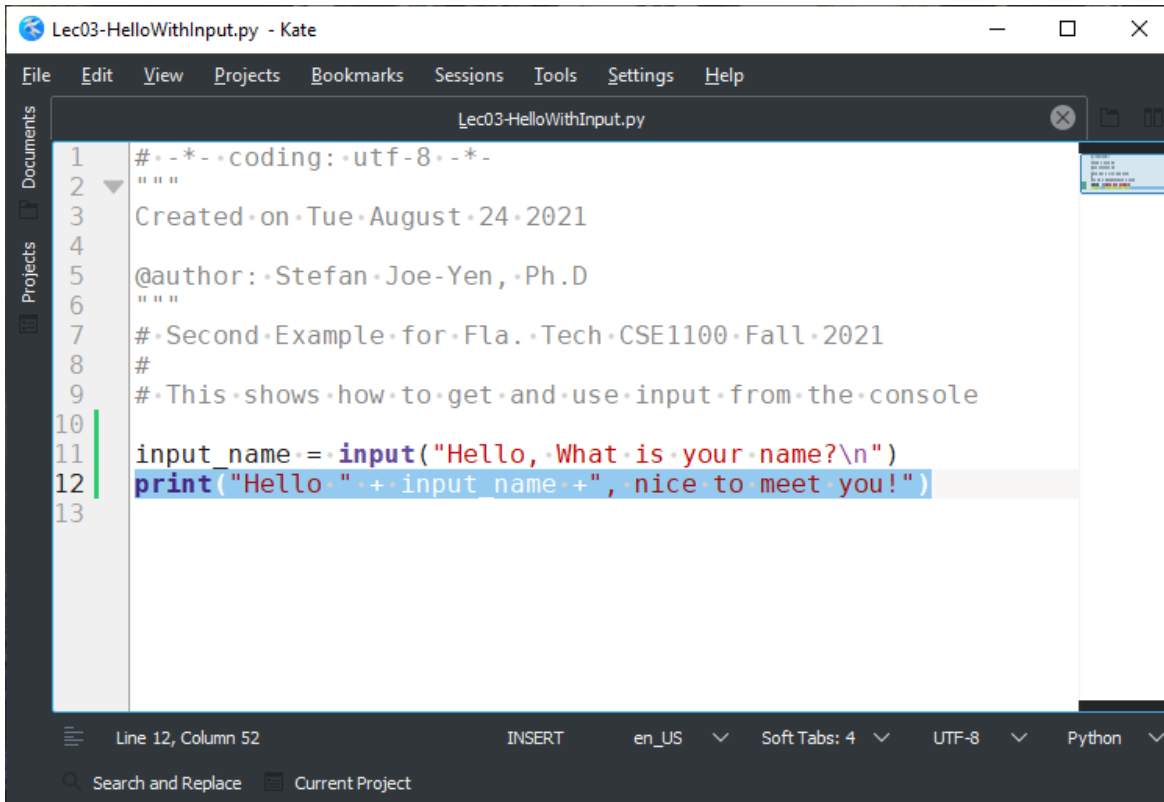
The screenshot shows a code editor window titled "Lec03-HelloWithInput.py - Kate". The editor contains a Python script with the following code:

```
1  #-*-coding:utf-8-*-
2  """
3  Created on Tue Aug 24 2021
4
5  @author: Stefan Joe-Yen, Ph.D
6  """
7  # Second Example for Fla. Tech CSE1100 Fall 2021
8  #
9  # This shows how to get and use input from the console
10
11  input_name = input("Hello, What is your name?\n")
12  print("Hello " + input_name + ", nice to meet you!")
13
```

The code is written in a dark-themed editor. Line 11 is highlighted in blue. The status bar at the bottom indicates "Line 11, Column 50", "INSERT", "en\_US", "Soft Tabs: 4", "UTF-8", and "Python".

- ▶ The *input* function prints a message of your choice and then reads whatever is typed at the keyboard until the user hits "enter".
- ▶ This data is assigned to a named storage area called a variable using the = operator (read "gets").
- ▶ In the next lecture we will study variables in detail. Think of them as labeled boxes to store data. In this case, the variable is named *input\_name*

# Using the stored data



```
Lec03-HelloWithInput.py - Kate
File Edit View Projects Bookmarks Sessions Tools Settings Help
Lec03-HelloWithInput.py
1 #-*-coding:utf-8-*-
2 """
3 Created on Tue Aug 24 2021
4
5 @author: Stefan Joe-Yen, Ph.D
6 """
7 # Second Example for Fla. Tech CSE1100 Fall 2021
8 #
9 # This shows how to get and use input from the console
10
11 input_name = input("Hello, What is your name?\n")
12 print("Hello " + input_name + ", nice to meet you!")
13

Line 12, Column 52 INSERT en_US Soft Tabs: 4 UTF-8 Python
Search and Replace Current Project
```

- ▶ The last line produces the output using the data stored in *input\_name*.
- ▶ This data is integrated with fixed text by using `+` as a concatenation operator
- ▶ In the next few lectures, we will study operators in detail.

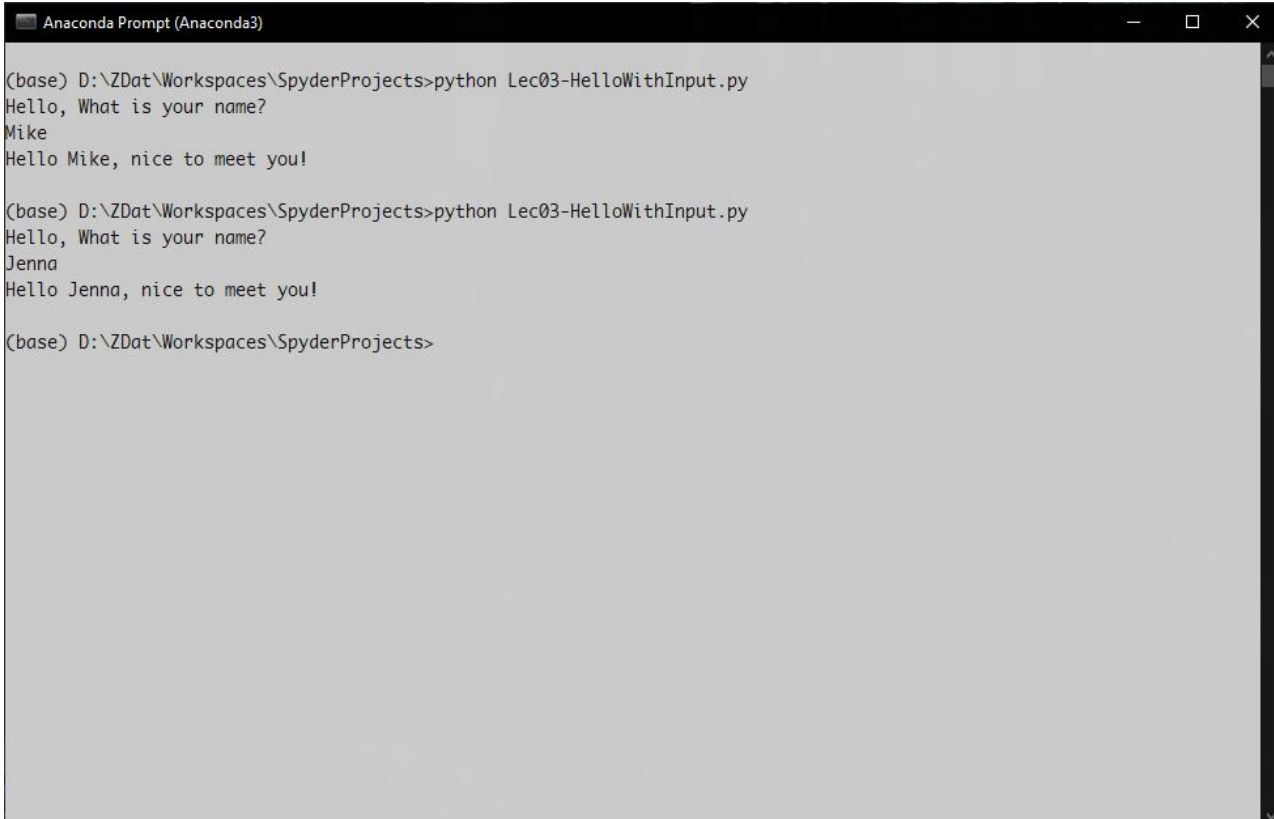


# REPL (Read Evaluate Print Loop)

```
Anaconda Prompt (Anaconda3) - python
(base) D:\ZDat\Workspaces\SpyderProjects>python
Python 3.8.10 (default, May 19 2021, 13:12:57) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> input_name = input("Hello, What is your name?\n")
Hello, What is your name?
Bill
>>> print("Hello " + input_name + ", nice to meet you!")
Hello Bill, nice to meet you!
>>> █
```

- The REPL should be used to try out short pieces of code.
- It is cumbersome to enter and run longer programs one line at a time.

# Console output from Scripts

A screenshot of an Anaconda Prompt window. The title bar reads "Anaconda Prompt (Anaconda3)". The window shows a command prompt session where a Python script is executed twice. The first execution prompts for a name, receives "Mike", and outputs "Hello Mike, nice to meet you!". The second execution prompts for a name, receives "Jenna", and outputs "Hello Jenna, nice to meet you!". The prompt is at the directory "D:\ZDat\Workspaces\SpyderProjects".

```
(base) D:\ZDat\Workspaces\SpyderProjects>python Lec03-HelloWithInput.py
Hello, What is your name?
Mike
Hello Mike, nice to meet you!

(base) D:\ZDat\Workspaces\SpyderProjects>python Lec03-HelloWithInput.py
Hello, What is your name?
Jenna
Hello Jenna, nice to meet you!

(base) D:\ZDat\Workspaces\SpyderProjects>
```

For longer programs, launch in batch mode from the command line by typing:

*python myfilename.py*

replacing *myfilename.py* with the actual name of your script.

# Data Representation & Types in Python

Core Concepts

# Variables and naming rules

- ▶ Variables are labels for storage space in memory.
- ▶ The value stored in a variables can be changed over time - hence the name.
- ▶ Variable names follow certain restrictions
  - ▶ Cannot start with a digit
  - ▶ Cannot contain spaces
  - ▶ Cannot use special characters other than the underscore (`_`)
    - ▶ Used for multiword labels and as a flag for special variables. Avoid starting variable names with `_` even though it will not produce an error.

- ▶ Variable names should:
  - ▶ Be descriptive (e.g. `account_balance` instead of `Q`)
  - ▶ Be consistent ([PEP 8](#) style guide discussed further in Lab 2. Click on the link for more info.)
  - ▶ Pick one and stick with it:
    - ▶ `multiWord`
    - ▶ `multi_word`
- ▶ The type of a variable is dynamic and is determined upon assignment.
  - ▶ `my_name = "Fred"`
  - ▶ `numEggs = 12`
  - ▶ `Phi = 1.61803`

# Built-in Types (1): Integers & Booleans

## ► Integers

- This is the core numeric type it can only hold whole number values. By assigning a value with a literal digit string with no decimals an integer type is inferred.

## ► Examples:

- `a = 6`
- `b = 9`
- `c = a + b`

## ► Boolean

- This type is used for logic and comparison where the allowed values can only be true or false
- The special values **True** and **False** can be assigned to and retrieved from Boolean variables.

## ► Examples:

- `doorIsOpen = False`
- `maryIsHappy = True`

# Built-in Types (2): Float and Complex

- ▶ Floating Point variables (type: float) are based on IEEE 754 standard for bit representation on most platforms. Specify by numeric literals with decimal points:

- ▶ `heightInches = 64.7`

- ▶ Complex variables are built-in types in python and use the engineering convention:

$$j = \sqrt{-1}$$

- ▶ Example:

- ▶ `z = 3 + 4j`

- ▶ Warnings:

- ▶ Real-number Representation & Rounding
  - ▶ Platform Specific Limitations on range and precision
    - ▶ Range: the absolute value
    - ▶ Precision: accuracy of decimals

# Built-in Types (3): Strings

- ▶ Character values are handled by the string type (str) in Python
- ▶ Strings can be specified 3 different ways
- ▶ Single Quotes
  - ▶ `myString1 = 'He said "Hello"!\n'`
- ▶ Double Quotes
  - ▶ `myString2 = "It's another string"`
- ▶ Three Sets of Double Quotes
  - ▶ `""" This string  
can span  
multiple lines """`

- ▶ Notes:
  - ▶ Escape sequences are preceded by a back-slash '\' to denote special and non-printing characters.
  - ▶ Common specials are:
    - ▶ `'\n'` - new line
    - ▶ `'\r'` - carriage return
    - ▶ `'\t'` - tab
    - ▶ `'\\'` - the backslash itself
    - ▶ `'\''` - the single quote
    - ▶ `'\"'` - the double quote
    - ▶ `\` (continuation of a line)

# Checking & Changing the Type of a Variable

```
Anaconda Prompt (Anaconda3) - python

(base) C:\Users\Draxwulf>python
Python 3.8.10 (default, May 19 2021, 13:12:57) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
>>>
>>> a = 6
>>> b = 7.9
>>> z = 8 + 14j
>>>
>>> type(a)
<class 'int'>
>>> type(b)
<class 'float'>
>>> type(z)
<class 'complex'>
>>>
>>> myString1 = "This is cool!"
>>> type(myString)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'myString' is not defined
>>> type(myString1)
<class 'str'>
>>>
>>> a = '6'
>>> type(a)
<class 'str'>
>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "float") to str
>>> float(a) + b
13.9
>>>
```

- ▶ You can query the type of a variable using the `type()` function
- ▶ You can also change the type of a variable using a function named after the target type:
  - ▶ `float()`
  - ▶ `int()`
  - ▶ `str()`
- ▶ We'll see how this comes in handy for using console input in the next session
- ▶ Note: error messages usually start with the word "Traceback" and try to provide a description of what went wrong on which line of code.
- ▶ Here we see a `NameError` when the name is not exactly correct and a `TypeError` when we try to add a float to a string (not allowed).



# Variables and Types

- ▶ Variables are labels for storage space in memory.
- ▶ Variable names follow certain restrictions
  - ▶ Cannot start with a digit
  - ▶ Cannot contain spaces
  - ▶ Cannot use special characters other than the underscore (`_`)
    - ▶ Used for multiword labels and as a flag for special variables. Avoid starting variable names with `_` even though it will not produce an error.

- ▶ The type of a variable is dynamic and is determined upon assignment.
  - ▶ `my_name = "Fred"`
  - ▶ `numEggs = 12`
  - ▶ `Phi = 1.61803`
  - ▶ `z = 3 + 4j`
  - ▶ `thisIsCool = True`
  - ▶ `pythonIsTooHard = False`
- ▶ These are of type `str`, `int`, `float`, `complex`, and two instances of `bool`, respectively.

# Checking & Changing the Type of a Variable

```
Anaconda Prompt (Anaconda3) - python

(base) C:\Users\Draxwulf>python
Python 3.8.10 (default, May 19 2021, 13:12:57) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
>>>
>>> a = 6
>>> b = 7.9
>>> z = 8 + 14j
>>>
>>> type(a)
<class 'int'>
>>> type(b)
<class 'float'>
>>> type(z)
<class 'complex'>
>>>
>>> myString1 = "This is cool!"
>>> type(myString)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'myString' is not defined
>>> type(myString1)
<class 'str'>
>>>
>>> a = '6'
>>> type(a)
<class 'str'>
>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "float") to str
>>> float(a) + b
13.9
>>>
```

- ▶ You can query the type of a variable using the `type()` function
- ▶ You can also change the type of a variable using a function named after the target type:
  - ▶ `float()`
  - ▶ `int()`
  - ▶ `str()`
- ▶ We'll see how this comes in handy for using console input in the next session
- ▶ Note: error messages usually start with the word "Traceback" and try to provide a description of what went wrong on which line of code.
- ▶ Here we see a `NameError` when the name is not exactly correct and a `TypeError` when we try to add a float to a string (not allowed).

# Operators

Core Concepts

# Operators (Part 1) - Arithmetic

- ▶ Binary Operators
- ▶ Use infix notation (e.g. 6+7)
- ▶ + Addition
- ▶ - Subtraction
- ▶ \* Multiplication
- ▶ / Division - Forces floating point division
- ▶ % Modulus
  - ▶ (remainder after integer division)
- ▶ \*\* Exponentiation
- ▶ // floor division - Forces integer division

- ▶ (Compound) Assignments
  - ▶ =
  - ▶ += increments and assigns
    - ▶  $a += 1$  same as  $a = a + 1$
  - ▶ -= decrements and assigns
    - ▶  $b -= 3$  same as  $b = b - 3$
  - ▶ \*= multiplies and assigns
  - ▶ /= divides and assigns
  - ▶ ...

# Operators (Part 2) - Logical

## ► Binary Operators

- Returns **True** or **False** based on the logical value of operands

- **and** Logical and

isRaining = **True**

haveUmbrella = **False**

a = (isRaining) **and** (haveUmbrella) **#False**

- **or** Logical or

isCloudy = **True**

isCold = **False**

b = (isCloudy) **or** (isCold) **#True**

## ► Unary Operators

- **not** Logical not - negates value

isPresent = **True**

c = **not**(isPresent) **#False**

## ► Comparison

- Returns **True** or **False** based on the result of the comparison

### ► **==** Equal

- a = 7

- b = 7

- c = a==b **#True**

### ► **!=** Not Equal

- d = a != b **#False**

### ► **<** Less than

### ► **>** Greater than

### ► **<=** Less than or equal to

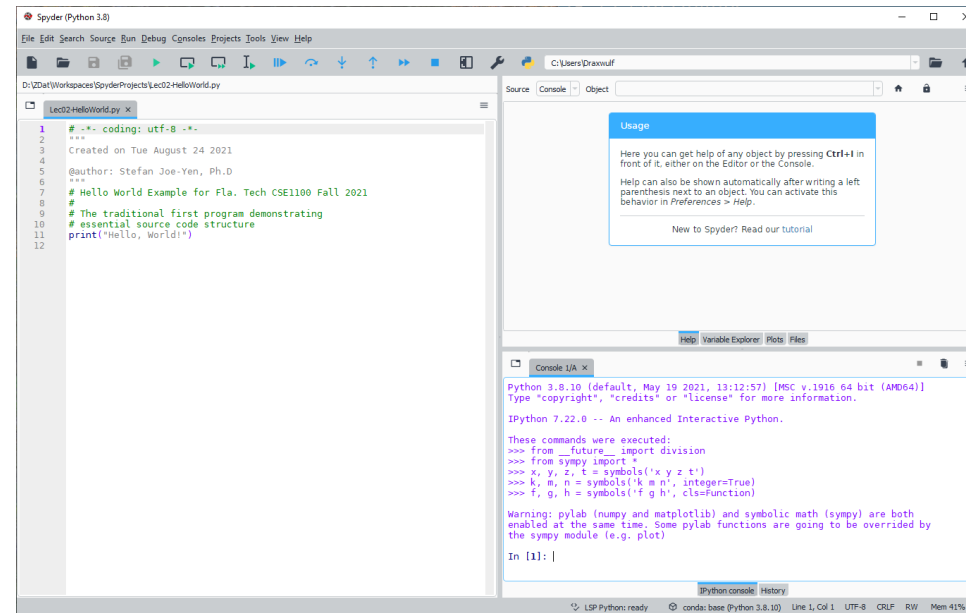
### ► **>=** Greater than or equal to

# Demo - Spyder IDE

Core Concepts

# Shown in Lecture 5 - class session.

- ▶ Spyder console included with Anaconda python distribution
- ▶ Contains
  - ▶ Editor (left panel) to create, load, and save scripts.
  - ▶ Console (lower right panel) for output and interactive execution
  - ▶ Multipurpose panel (upper right). Tabbed interface with Help, Variable Explorer, File Explorer, and Plots.



# Administrative Items



# Operators

- ▶ Arithmetic Operators

- ▶ (+, -, \*, /, %, \*\*, //)

- ▶ (Compound) Assignments

- ▶ (=, +=, -=, \*=, ...)

- ▶ Result is numeric:

- ▶ int
  - ▶ float
  - ▶ complex

- ▶ Logic

- ▶ (and, or, not)

- ▶ Comparison

- ▶ (<, >, ==, !=, <=, >=)

- ▶ Result is logical:

- ▶ True
  - ▶ False

# Program Control Structures

Conditional Execution

# If...then structures

- Statements can be set up to execute only if a logical condition is true.

```
if (a < 10):  
    print("a is less than 10\n")  
print("This prints regardless.\n")
```

- Multiple statements belonging to the same condition are enclosed in a block (at same indent level).

```
if (a < 10):  
    print("a is less than 10\n")  
    print("This prints conditionally\n")
```

# If...then...else structures

- Statements can be set up to execute an alternative path based on the logical condition.

```
if (grade > 60):  
    print("You pass the class.")  
else:  
    print("You fail the class.")  
print("Your grade details:")
```

- Blocks work the same as before.

```
if (Temperature > 70):  
    print("It feels OK.")  
    print("I don't need a jacket")  
else:  
    print("It's kind of chilly.")  
    print("Wear warmer clothes.")  
print("This prints regardless.")
```

# Multi-Branch Structures

Conditional Execution

# If...then...elif structures

- Multiple conditions can be checked in a row.

```
if (grade >= 90):  
    print("You get an A")  
elif (grade >= 80):  
    print("You get a B")  
elif (grade >= 70):  
    print("You get a C")  
elif (grade >= 60):  
    print("You get a D")  
else:  
    print("You get an F")  
    print("This is bad.")  
print("End of grade report.")
```

## Notes:

- If you want multiple statements to execute on a particular checked condition, make sure it is at the same indentation level.
- "elif" clauses only execute if the previous "if" clause does not. The first true elif condition is executed.

# Loops

Repeated Execution

# While Loops

- ▶ While loops are similar to the `if` statement.
- ▶ A Boolean condition is tested
- ▶ The subordinate statement is repeated as long as the condition is true.

```
x = 0
```

```
while (x < 19):  
    print("Are we there yet?")  
    print("\n")  
    x += 1
```



# While Loops - Infinite Loops

- This could theoretically go on forever if the condition being monitored is never reached.

```
guess = 0
answer = 200
while (guess != answer):
    inpString = input("Guess a number between 0 and 100?")
    guess = int(inpString)
```

Can you spot the problem in the code above?

# While Loops - Infinite Loops

- This could theoretically go on forever if the condition being monitored is never reached.

```
guess = 0
answer = 200
while (guess != answer):
    inpString = input("Guess a number between 0 and 100?")
    guess = int(inpString)
```

Can you spot the problem in the code above?

The on-screen instructions are wrong. A user would never guess the correct answer. This is a logical/design error and would be missed by syntax checks it is only noticeable when run-tested.

# While Loops - Break and Continue

- ▶ A **break** statement can be used to exit the loop .
- ▶ A **continue** statement skips to the next loop iteration.

```
while (guess != answer):  
    inpString = input("Guess a number between 0 and 100?")  
    guess = int(inpString)  
    count += 1  
    if (count > 5):  
        break  
    else:  
        continue
```

# While Loops

- ▶ You can also deliberately set up an infinite loop assuming some internal logic will execute a `break` at some time.

```
# Keep going until a function returns true, then break.  
while (True):  
    # do some stuff  
    if (quit()):  
        break
```

# For Loops

- ▶ For loops allow you to perform an action on each item in a sequence.
- ▶ A common sequence is simply the integers in a certain range

```
for i in range(10):  
    # do some stuff  
    # i goes from 0 to 9
```

- ▶ The range function defaults to starting on 0 but can be given an alternate start point.

```
for i in range(2, 9):  
    # do some stuff  
    # i goes from 2 to 8
```

- ▶ Since the default for indexed collections is to start at index 0 the range function is using strictly-less-than (<) as the comparator against the stated maximum value.
- ▶ This means the sequence does not include the max value. (e.g. range(7) only goes from 0 to 6.
- ▶ The range function can also take an increment argument. The default increment is 1. Here we step by 2.

```
for i in range(2, 10, 2):  
    # do some stuff  
    # i goes from 2 to 8 by 2
```

# Conditional Execution (summary)

## Branching with `if`, `else`, and `elif`

```
if (cond):  
    statement block
```

```
if (cond):  
    true branch  
else:  
    false branch
```

```
if (cond1):  
elif (cond2):  
elif (cond3):  
    ...  
else:
```

# Repeat Execution (summary)

## Looping with `while` and `for`

```
while(cond):  
    statement block
```

```
for var in list:  
    # do something with var
```

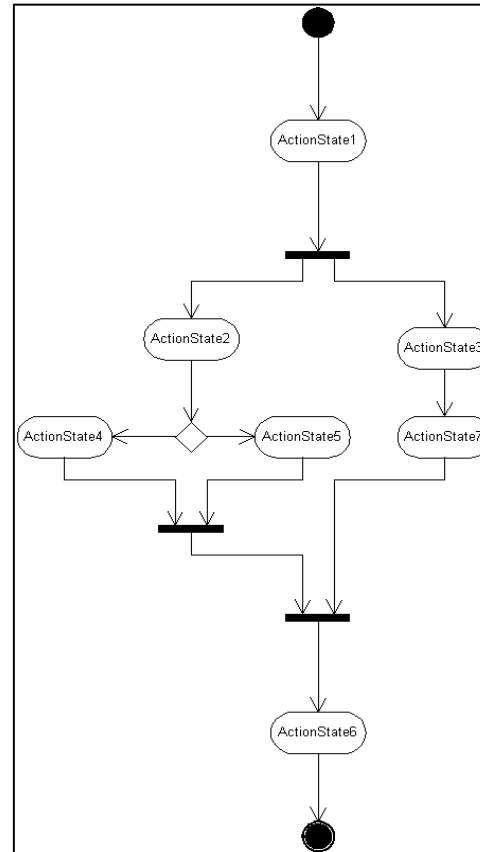
# Continue & Break

- ▶ Two keywords are commonly used in branch and loop structures. They are unconditional branches that jump to another point in the program.
- ▶ `break` exits a structure and continues program execution after the last statement in the block of the containing structure.
- ▶ `continue` exits the current loop iteration and continues program execution with the next iteration.



# Program Control - Closing thoughts

- ▶ Conditions can be any Boolean expression.
- ▶ All these decision structures can be composed and nested to create complex programs that branch and loop arbitrarily.
- ▶ Diagramming with [flowcharts](#) and/or UML can help during analysis and design. ([UML Official Site](#))
- ▶ Using loops unnecessarily can vastly increase the memory required and slow down the execution speed of a program.



UML State Diagram Example

# Program Design & Style Notes

Clean Code

# Magic Numbers & Named Constants

Avoid Magic Numbers:

```
while (guess != answer):  
    inpString = input("Guess a number between 0 and 100: ")  
    guess = int(inpString)  
    count += 1  
    if (count > 5):  
        break  
    else:  
        continue
```

# Magic Numbers & Named Constants

Use Named Constants instead:

```
MAX_TRIES = 5
while (guess != answer):
    inpString = input("Guess a number between 0 and 100: ")
    guess = int(inpString)
    count += 1
    if (count > MAX_TRIES):
        break
    else:
        continue
```

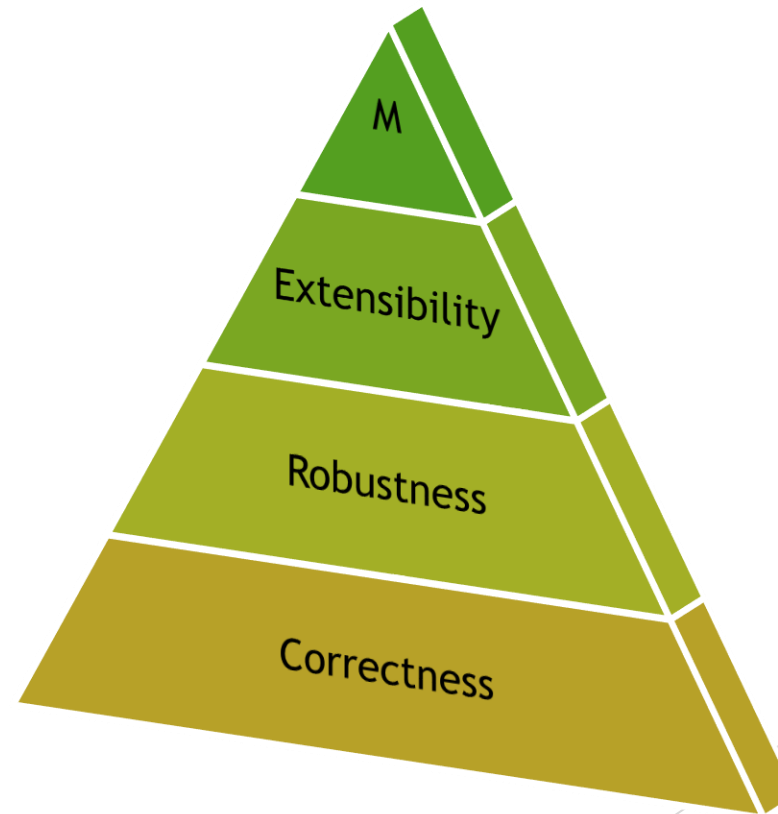
- Note: Python does not enforce named constants ... it's up to the programmer to make sure the value is never updated after initializing it.

# PEP8 - The Style Guide for Python

- ▶ Python.org maintains a set of PEPs (Python Enhancement Proposals)
  - ▶ Repository of suggestions and community standards
- ▶ Refer to [PEP8](#) for proper style
- ▶ The Spyder IDE and other IDEs provide hints and warnings based on PEP8

# Evaluating Software Design/Implementation

- ▶ Correctness
  - ▶ Design By Contract
- ▶ Robustness
  - ▶ Test-First/Unit Tests
- ▶ Extensibility
  - ▶ Modular/Encapsulation
- ▶ M: Mastery + Elegance
  - ▶ Clarity & Organization
  - ▶ Artistic Originality



# Demo & Lab Setup

- ▶ Try out concepts from today's lecture

# Python Lists (Part 1)

Creating and Using Lists



# Python Containers & Collections

- ▶ So far we have seen variables that refer to single values.
- ▶ Sometimes you want to operate on collections of values as a unit.
- ▶ Python has collections as built-in types along with associated functions for manipulating them.
- ▶ The built in containers are:
  - ▶ List (An ordered, indexed sequence)
  - ▶ Dictionaries (Associative containers; maps keys to values)
  - ▶ Sets (Acts like mathematical sets)
  - ▶ Tuples (Immutable lists w/ "special features")

# Creating Lists

- ▶ To create a list assign a comma-separated sequence of values in square brackets - []

```
>>> myList = [2, 4, 6, 8, 10]
```

- ▶ Lists can contain values of multiple types

```
>>> myList = [2, "Fred", 16.7]
```

- ▶ Empty lists can be specified

```
>>> myList = []
```

# Accessing Elements in Lists

- ▶ To access an individual element of a list, put the ordinal number of the element in square brackets [] after the variable name.
- ▶ Lists are numbered starting with 0.
- ▶ The `len()` function is helpful to track list sizes and prevent range errors.

```
>>> myList = [2, "Fred", 16.7]
```

```
>>> name = myList[1]
```

```
>>> myList[1]
```

```
'Fred'
```

```
>>> len(myList)
```

```
3
```

# Using Lists With Loops

- ▶ We saw how to use the `range()` function previously.
- ▶ To iterate through an entire list we can use a for loop on the index values

```
>>> for i in range(len(myList)):
...     print(i, "->" ,myList[i])
...
0 -> 2
1 -> Fred
2 -> 16.7
>>>
```

# Lists & Variable Assignment

- ▶ List assignment acts differently than the variable assignments discussed before.
- ▶ Assigning a list to another variable creates a *reference*.

```
>>> myList = [2, "Fred", 16.7]
```

```
>>> otherList = myList
```

```
>>> otherList[1] = "Wilma"
```

```
>>> myList[1]
```

```
'Wilma'
```

- ▶ To create an entirely new list from an existing list you use the `list()` function.

```
>>> newList = list(myList) # these now are independent lists
```

# Python Lists (Part 2)

List operations

# Appending to Lists

- ▶ To append more elements to a list you can use the append method.
- ▶ The element is added to the next index and length is adjusted accordingly.

```
>>> myList = [2, "Fred", 16.7]
```

```
>>> myList.append("Apples")
```

```
>>> len(myList)
```

```
4
```

```
>>> myList[3]
```

```
'Apples'
```

# Insert Elements

- ▶ The `insert()` method inserts an element before the given index.

```
>>> myList = [2, "Fred", 16.7]
```

```
>>> myList.insert(2, "Barney")
```

```
>>> myList
```

```
[2, 'Fred', 'Barney', 16.7]
```

- ▶ Inserting at an index one higher than the current last element is equivalent to append.

```
>>> myList.insert(4, 3.14159)
```

```
>>> [2, 'Fred', 'Barney', 16.7, 3.14159]
```



# Delete Elements

- ▶ The `pop()` method removes (and returns) an element at the given index.

```
>>> myList = [2, "Fred", 16.7]
```

```
>>> myList.pop(1)
```

```
'Fred'
```

```
>>> myList
```

```
[2, 16.7]
```

The `remove()` method erases an element with a specified value.

```
>>> myList = [2, 'Fred', 'Barney', 16.7, 3.14159]
```

```
>>> myList.remove("Barney")
```

```
>>> myList
```

```
[2, 'Fred', 16.7, 3.14159]
```

# List Operations (Review)

- ▶ Creating

```
>>> myList = [2, 4, 6, 8, 10]
```

- ▶ Accessing

```
>>> name = myList[1]
```

- ▶ Copying

```
>>> newList = list(myList) # These now are independent lists
```

- ▶ Appending

```
>>> myList.append("Apples")
```

- ▶ Inserting

```
>>> myList.insert(4, 3.14159)
```

- ▶ Removing/Popping

```
>>> myList.pop(1) # Returns value as well as removes item
```

```
>>> myList.remove(8) # The value 8 not the 8th index
```

# Python Lists (Part 3)

More List Operations & Concepts

# Search a List for Elements

- ▶ The `in` keyword returns a Boolean `True` value if the specified element is in the specified list, `False` otherwise.

```
>>> myList = [2, "Fred", 16.7]
```

```
>>> if "Fred" in myList:
    print("Found it!")
else:
    print("Not Found!")
```

- ▶ The `index()` method returns the position of the element found
- ▶ An optional start position can be specified to search for the term.

```
>>> pos1 = myList.index("Fred")
```

```
>>> pos2 = myList.index("Fred", pos1+1)
```

# Operators + and \*

- ▶ The `+` and `*` operators are defined for lists as concatenation and replication respectively

```
>>> myList1 = [2, 4, 6]
```

```
>>> myList2 = [8, 10, 12]
```

```
>>> myList1 + myList2
```

```
[2, 4, 6, 8, 10, 12]
```

```
>>> myList1 * 4
```

```
[2, 4, 6, 2, 4, 6, 2, 4, 6, 2, 4, 6]
```

# Sum, Min, Max

- ▶ The `sum()` function returns the sum of all the elements
- ▶ The `min()` and `max()` functions return the minimum and maximum elements
- ▶ `myList1 = [2, 4, 6]`

```
>>> sum(myList1)
```

```
12
```

```
>>> min(myList1)
```

```
2
```

```
>>> max(myList1)
```

```
6
```

# Sort

- ▶ The `sort()` method sorts the elements from min to max.
- ▶ Sorting happens in place (i.e. the list is altered).
- ▶ Text data is sorted in lexical order.

```
>>> myList1 = [6, 20, 3]
>>> myList2 = ["Fred", "Barney", "Wilma"]
>>> myList1.sort()
>>> myList2.sort()
>>> myList1
[3, 6, 20]
>>> myList2
['Barney', 'Fred', 'Wilma']
>>>
```

# Slicing and Negative Indexes

- ▶ Using a `:` in the index allows you to specify a range  $[n:(m - 1)]$ 
  - ▶ Empty range values represent the start and end of the list
  - ▶ The first range value is included the last range value is excluded.
- ▶ Negative indexes start from the back of the list ( $-1 \rightarrow$  last element)
- ▶ Here are some examples:

```
>>> myList1 = [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
>>> myList1[4:7]
[15, 18, 21]
>>> myList1[3:]
[12, 15, 18, 21, 24, 27, 30]
>>> myList1[:5]
[3, 6, 9, 12, 15]
>>> myList1[-2]
27
```



# Python Lists (Part 4)

More List Algorithms & Idioms (Patterns)

# Initialize and Fill a List

► Objective: Fill a list with values based on a formula.

► Example:

```
>>> n = 8 # an integer
>>> myList = [] # start with an empty list
>>> for i in range(n): # loop over the index values
    myList.append(2 * i) # append current value
>>> myList
[0, 2, 4, 6, 8, 10, 12, 14]
```

# Linear Search

- ▶ The `in` keyword returns a Boolean `True` value if the specified element is in the specified list, `False` otherwise and
- ▶ The `index()` method returns the position of the element found.
- ▶ These can be used to create a linear search algorithm.

```
>>> values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> pos = 0
>>> found = False
>>> while (pos < len(values) and not found):
...     if (values[pos] % 3 == 0):
...         found = True
...     else:
...         pos += 1
... if found:
...     print("Found match at position:", pos)
... else:
...     print("No list item matches.")
```

# Build or Count a List of Matches

- For loops make this easy.

Example:

```
>>> values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> result = []
>>> for value in values:
...     if (value % 3 == 0):
...         result.append(value)
```

- A count is just as simple.

Example:

```
>>> values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> count = 0
>>> for value in values:
...     if (value % 3 == 0):
...         count += 1
```

# Item Swap

- ▶ A common idiom that's good to know in any language.

Example:

```
>>> values = [1, 2, 3, 4]
>>> i = 1
>>> j = 2
>>> temp = values[i]
>>> values[i] = values[j]
>>> values[j] = temp
>>> values
[1, 3, 2, 4]
```

# Python Dictionaries (Part 1)

Creating and Using Dictionaries

# Python Containers & Collections

- ▶ Python has collections as built-in types along with associated functions for manipulating them.
- ▶ (Review) The built in containers are:
  - ▶ List (An ordered, indexed sequence)
  - ▶ Dictionaries (Associative containers; maps keys to values)
  - ▶ Sets (Acts like mathematical sets)
  - ▶ Tuples (Immutable lists w/ "special features")
- ▶ Last week we explored lists.
- ▶ The next few lectures will explore dictionaries

# Creating Dictionaries

- ▶ To create a dictionary assign a sequence of associated key-value pairs in curly braces as follows:

```
>>> animalColors = {"frog":"green", "swan":"white", "horse":"brown"}
```

- ▶ The keys and values are separated by the : character.
- ▶ The pairs are separated by commas. Any types can be associated freely:

```
>>> randomDict = {"Fred":16.7, "Burger":8}
```

- ▶ Empty dictionaries can be specified:

```
>>> emptyDict = {}
```



# Accessing Values in Dictionaries

- ▶ To access an individual element of a dictionary, put the key of the element in square brackets [] after the variable name.

```
>>> animalColors = {"frog":"green", "swan":"white", "horse":"brown"}
>>> frogColor = animalColors["frog"]
>>> frogColor
'green'
```

- ▶ You can update the value at any key by assignment.

```
>>> animalColors["swan"] = "black"
>>> animalColors
{"frog":"green", "swan":"black", "horse":"brown"}
```

- ▶ Note that, unlike lists, dictionaries are unordered and there is no concept of indexing elements by their position number.

# Duplicating and Adding Entries

- ▶ To create a duplicate dictionary from an existing one you use the `dict()` function.

```
>>> newColors = dict(animalColors)
```

- ▶ An empty dictionary can be filled dynamically by assigning values to keys as follows.

```
>>> newColors = {}
```

```
>>> newColors["apple"] = "red"
```

```
>>> newColors["orange"] = "orange"
```

```
>>> newColors["banana"] = "yellow"
```

# Delete Entries

- ▶ The `pop()` method removes both the key and the value from the dictionary and returns the value for the given key that can be assigned.

```
>>> animalColors = {"frog":"green", "swan":"white", "horse":"brown"}
>>> frogColor = animalColors.pop("frog")
>>> frogColor
'green'
>>> animalColors
{"swan":"white", "horse":"brown"}
```

# Python Dictionaries (Part 2)

More Dictionary Functions

# Creating Dictionaries (Review)

- ▶ To create a dictionary assign a sequence of associated key-value pairs in curly braces as follows:

```
>>> animalColors = {"frog":"green", "swan":"white", "horse":"brown"}
```

- ▶ The keys and values are separated by the : character.
- ▶ The pairs are separated by commas. Any types can be associated freely:

```
>>> randomDict = {"Fred":16.7, "Burger":8}
```

- ▶ Access is similar to list indexing but uses keys not positions:

```
>>> randomDict[swan] = "black"
```

# Iterate over the Keys

- In the context of a for loop, the iteration is over the keys.

```
>>> animalColors = {"frog":"green", "swan":"white", "horse":"brown"}
>>> for key in animalColors:
...     print(key)
...
frog
swan
horse
```

# Iterate over the Keys

- ▶ You can print the related items by using the keys.

```
>>> animalColors = {"frog":"green", "swan":"white", "horse":"brown"}
>>> for key in animalColors:
...     print(key, "->", animalColors[key])
...
frog -> green
swan -> white
horse -> brown
```

# Iterate over Pairs

- ▶ The `items()` method returns all the pairs contained in the dictionary as tuples.
- ▶ Tuples are indexed exactly like lists.
- ▶ The underlying algorithm is more efficient than looking up the value for each key.

```
>>> animalColors = {"frog":"green", "swan":"white", "horse":"brown"}
>>> for pair in animalColors.items():
...     print(pair[0], "->", pair[1]) # each pair is a two-element object
...
frog -> green
swan -> white
horse -> brown
```



# Return all the values

- ▶ The `values()` method returns a sequence of all the values contained in the dictionary.

```
>>> animalColors = {"frog":"green", "swan":"white", "horse":"brown"}
```

```
>>> for value in animalColors.values():
```

```
...     print(value)
```

```
...
```

```
green
```

```
white
```

```
brown
```

- ▶ Note that dictionaries are unordered. For larger collections the previous outputs could have been returned in another order.

# Python Sets & Tuples

More Collection Structures

# Creating Sets

- ▶ To create a set assign a sequence of associated values in curly braces as follows:

```
>>> foods = {"Hamburger", "Pizza", "Sushi"}
```

- ▶ The elements are separated by commas.
- ▶ You cannot make an empty set with empty braces because that is treated as a dictionary. Instead there is a function that constructs a set.

```
>>> emptyDict = {}
```

```
>>> emptySet = set()
```

- ▶ The `in` and `not in` operator keywords can check set membership:

```
>>> "Pie" in foods #False
```

```
>>> "Sushi" in foods #True
```

```
>>> "Bagel" not in foods #True
```

# Set Operations (1)

► Given example sets

```
>>> odds = {1, 3, 5, 7, 9, 11}
```

```
>>> primes = {3, 5, 7, 11}
```

► Add elements

```
>>> odds.add(13)
```

```
>>> primes.add(13)
```

► Removing

```
>>> odds.discard(11) # Removes 11
```

```
>>> odds.discard(17) # Fails silently
```

```
>>> odds.remove(8) # Fails with an exception
```

► Clear the set

```
>>> odds.clear() # removes all elements; sets length to 0
```

# Set Operations (2)

- ▶ Given example sets

```
>>> odds = {1, 3, 5, 7, 9, 11}
```

```
>>> primes = {3, 5, 7, 11}
```

- ▶ Subset Test

```
primes.issubset(odds) # true
```

```
odds.issubset(primes) # false
```

- ▶ Union

```
>>> odds.union(primes)
```

```
{1, 3, 5, 7, 9, 11}
```

- ▶ Intersection

```
>>> odds.intersection(primes)
```

```
{3, 5, 7, 11}
```

- ▶ Difference

```
>>> odds.difference(primes)
```

```
{1, 9}
```

# Compound Structures

- ▶ All the collection types we have seen so far can be used as elements of other collections
- ▶ Thus you can have structures like lists of lists:

```
>>> nestedList = [1, [3, 5], [7, 9, 11]]
```

- ▶ and dictionaries of lists:

```
>>> mealMenus = {"Breakfast":["sausage", "ham"], "Lunch":["sandwich", "soup"], \
                  "Dinner":["sushi", "steak"]}
```

- ▶ The usual access and update methods work on the composite structure:

```
>>> mealMenus["Breakfast"].append("bacon")
```

```
>>> print(mealMenus)
```

```
{'Breakfast': ['sausage', 'ham', 'bacon'], 'Lunch': ['sandwich', 'soup'], 'Dinner':  
['sushi', 'steak']}
```

# Tuples

- ▶ Creating

```
>>> myTuple = (2, 4, 6, 8, 10) # cannot be changed once created
```

- ▶ Accessing uses the same syntax as list

```
>>> num = myTuple[1] # num gets 4
```

- ▶ List operations that don't alter the elements are supported by tuples

- ▶ `in`, `not in`

- ▶ iterating with `for`

- ▶ finding the size with `len()`

- ▶ A common use of tuples is in function definition & usage:

- ▶ used for varying the number of inputs: tuples as parameters

- ▶ used for returning multiple values: tuples as return objects

- ▶ The next major unit of the course focuses on functions.

# Zip() function

► The `zip()` function combines elements from two collections into tuple pairs.

```
a = ("Mary", "Ben", "Ralph")
```

```
b = (18, 26, 52)
```

```
q = zip(a, b)
```

```
#print the result:
```

```
print(tuple(q)) #tuple function formats output
```

```
(( 'Mary', 18), ( 'Ben', 26), ( 'Ralph', 52))
```

► You can use zip on any collection type to create the pairs as tuples

```
>>> set1 = {30, 60, 90}
```

```
>>> set2 = {'r', 'g', 'b'}
```

```
>>> list(zip(set1, set2))
```

```
[(90, 'r'), (60, 'g'), (30, 'b')] #note that this is randomized
```

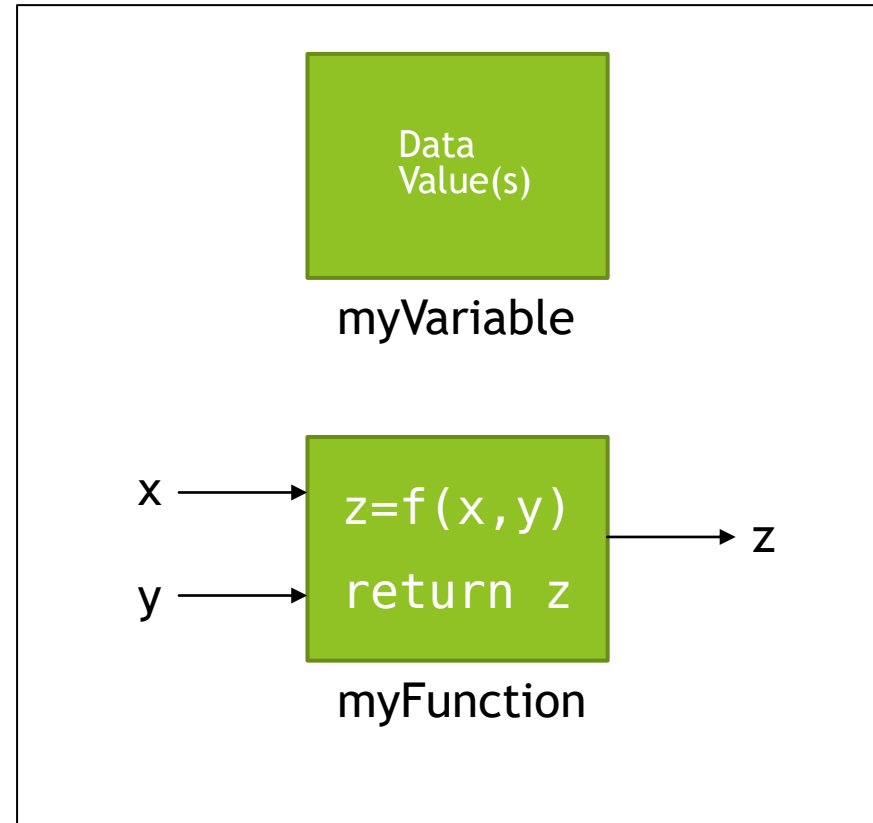


# Functions

Using and defining subprograms

# Functions Overview

- ▶ Functions act as sub-programs to perform a sequence of actions on-demand.
- ▶ A program that follows procedural (a.k.a. modular) design is a series of functions starting with the outer environment (REPL, Script File, `__name__ == __main__`).
- ▶ Functions optionally take input parameters and optionally return a value.
- ▶ Variables as labeled Boxes for Values.
- ▶ Functions as labeled Boxes for Actions.
- ▶ Functions Enable Hierarchical Abstraction since:
  - ▶ details are now encapsulated by the i/o interface
  - ▶ different sequences of actions (algorithms) produce the same results.



# Functions - Anatomy of a Function

- ▶ To define a function use `def`:

```
def identifier(params):
```

- ▶ followed by a block of instructions  
- the body of the function
- ▶ Identifier follows the same naming rules as variables:
  - ▶ No spaces, or initial numerals
  - ▶ The only special char allowed is '\_'
  - ▶ No reserved words.

- ▶ Definition (Body):

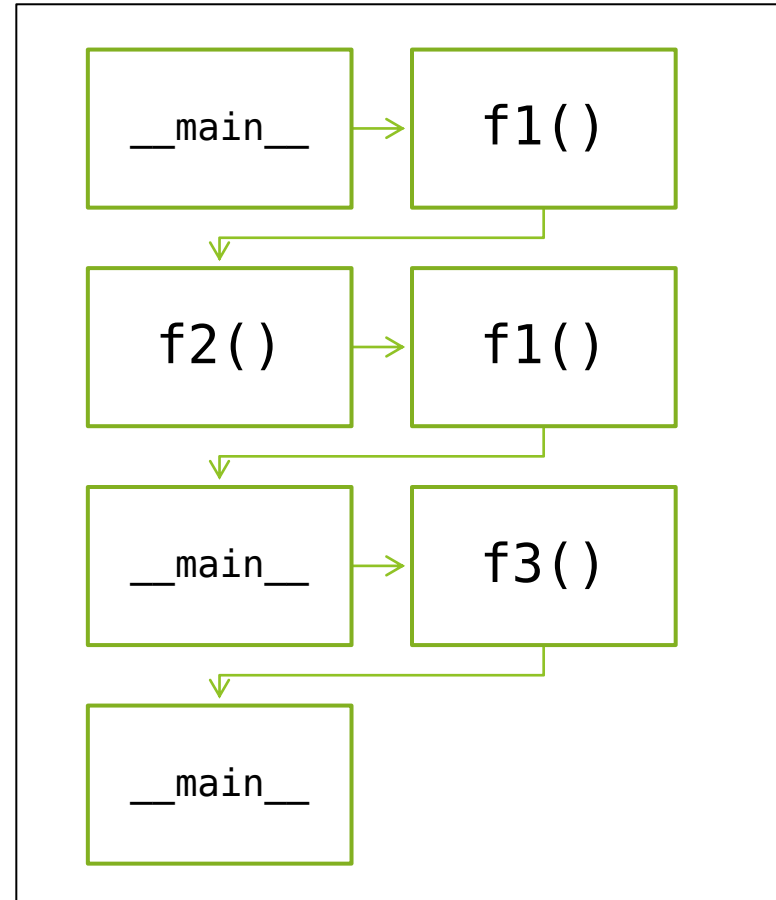
- ▶ Statements + Control Structures
- ▶ return

- ▶ Example:

```
def square(x):  
    return x*x
```

# Execution/Scope Pathways

```
def f1():  
    #do some f1 stuff  
    f2():  
    #do more f1 stuff  
    return  
  
def f2():  
    #do f2 stuff  
    return  
  
def f3():  
    #do f3 stuff  
    return  
# Function Call Sequence Example  
f1()  
f3()
```



# Parameter Variables

```
def myFunc1(x,y) :  
    # do stuff with x and y  
    # can also declare more local variables
```

- The parameter names (x and y in this example) are local placeholders and do not need to match the names of parameters passed in. The original variables are left unaffected for (non-collection) built-in types.

# Function and Variable scope

- ▶ Scope rules apply to the visibility of the identifier. Possible scopes include:
  - ▶ *file* (declared outside a block or function)
  - ▶ *control-block* (declared within a control structure)
  - ▶ *function* (declared within a function body)

# Parameters and Local Variables

- ▶ The body of the function is contained as a statement block
- ▶ Can contain any other loop or decision block as part of the function body
- ▶ Variables declared within the block are not visible outside the block.
- ▶ Example Output:

```
function 1 weighted sum: 66  
function 2 weighted sum: 84
```

```
def f1(x,y,z) :  
    a,b,c; #these are local to f1  
    a = 1  
    b = 3  
    c = 5 #set some values  
    print("function 1 weighted sum: " + str(a*x+b*y+c*z) + "\n")  
  
def f2(x,y,z) :  
    a,b,c; #these are local to f2  
    a = 2  
    b = 4  
    c = 6 #set some values  
    print("function 2 weighted sum: " + str(a*x+b*y+c*z) + "\n")  
  
p=3  
q=6  
r=9  
f1(p,q,r)  
f2(p,q,r)
```

# Parameters - Passing By Value

- ▶ Functions use pass by value as a default behavior for non-collection parameters.
- ▶ In the example below, copies of parameter values are stored in variables `x` and `y` that are local to the function.
- ▶ Leaves original values unaffected.

```
def myValFunction(x,y):  
    x+=1  
    y+=1  
    return x * y  
  
x = 6  
y = 7  
  
print(myValFunction(x,y))  
print(x,y)
```



# Quiz Answer Recap

Using and defining subprograms

Attempts: 17 out of 17

Registers, Arithmetic Logic Unit (ALU) and Control Unit are built-in directly into the CPU.

True	16 respondents	94 %	<div><div></div></div> ✓
False	1 respondent	6 %	<div><div></div></div>

-0

Discrimination Index



94% answered correctly

Attempts: 17 out of 17

The Python compilation process: [item1]

item1

Translates high level code to intermediate bytecode	16 respondents	94 %	<div></div> ✓	94% answered correctly
Translates assembly code to machine code.		0 %	<div></div>	
Executes bytecode instructions.		0 %	<div></div>	
Translates high-level code to an executable file.	1 respondent	6 %	<div></div>	

Attempts: 17 out of 17

+0.5

Which of the following is *not* a valid identifier?

Discrimination Index



my Value	16 respondents	94 %	<div><div></div></div> ✓
_AAA1		0 %	<div><div></div></div>
width		0 %	<div><div></div></div>
m_x	1 respondent	6 %	<div><div></div></div>

94% answered  
correctly

Attempts: 17 out of 17

Which of the following statements could potentially change the value of number2?

<code>number2 = int(input("Number?"))</code>	16 respondents	94 %	<div></div> ✓
<code>sum = number1 + number2</code>		0 %	<div></div>
<code>number1 = number2</code>	1 respondent	6 %	<div></div>
<code>print(number2)</code>		0 %	<div></div>

+0.44

Discrimination Index



94% answered correctly

Attempts: 17 out of 17

+0.5

Discrimination Index  
?

What will be the output after the following statements have been executed?

```
a = 4
b = 12
c = 37
d = 51

if ( a < b ) :
    print("a < b")
if ( a > b ) :
    print("a > b")
if ( d <= c ) :
    print("d <= c")
if ( c != d ) :
    print("c != d")
```

a b c != d	16 respondents	94 %	<div></div> ✓
a b d = c c != d		0 %	<div></div>
a b c != d		0 %	<div></div>
a b c d a != b	1 respondent	6 %	<div></div>

94% answered correctly

Attempts: 17 out of 17

Which of the following results in an error?

Adding a string to an integer with +		0 %	<div></div>
Using => instead of >= in the condition of an if	1 respondent	6 %	<div></div>
Omitting the colon ( : ) after an if condition		0 %	<div></div>
All of the options	16 respondents	94 %	<div></div> ✓

**+0.21**

Discrimination Index



94% answered correctly

Attempts: 17 out of 17

Each of the following is a relational or equality operator *except*:

<=		0 %	<div></div>
!=	16 respondents	94 %	<div></div> ✓
==		0 %	<div></div>
>	1 respondent	6 %	<div></div>

**-0.07**

Discrimination Index



94% answered  
correctly



Attempts: 17 out of 17

A variable with Python data type `int` can be used to represent which of the following values:

9.6	1 respondent	6 %	<div></div>
"Big red dog"		0 %	<div></div>
True		0 %	<div></div>
None of the options	16 respondents	94 %	<div></div> ✓

**+0.57**

Discrimination Index



94% answered correctly

Attempts: 17 out of 17

The following for loop:

```
for i in range(1,10,2):  
...    print(i)
```

Prints the even numbers between 1 and 10		0 %	<div></div>
<b>Prints the odd numbers between 1 and 10</b>	16 respondents	<b>94 %</b>	<div></div> ✓
Runs infinitely		0 %	<div></div>
Prints 10 decimal numbers evenly spaced between 1 and 2	1 respondent	6 %	<div></div>

**+0.43**

Discrimination Index



94% answered correctly

Attempts: 17 out of 17

Which of the following is *false*?

A for loop can replace the functionality of a while loop	2 respondents	12 %	<div></div>
A for loop is useful for operating on collections such as lists		0 %	<div></div>
A for loop can never be infinite	15 respondents	88 %	<div></div> ✓

+0.52

Discrimination Index



88% answered correctly

Attempts: 17 out of 17

Which of the following for headers is *not* valid?

for i in range(4,10,6):	1 respondent	6 %	<div></div>
for i in range(10)	16 respondents	94 %	<div></div> ✓
All of the options		0 %	<div></div>

**+0.21**

Discrimination Index



94% answered  
correctly

Attempts: 17 out of 17

Which of the following is *false*?

break and continue statements alter the flow of control		0 %	<div></div>
continue statements skip the remaining statements in current iteration of the body of the loop in which they are embedded	2 respondents	12 %	<div></div>
break statements exit from the loop in which they are embedded		0 %	<div></div>
<b>continue and break statements may be embedded only within repetition statements</b>	15 respondents	88 %	<div></div> ✓

**+0.35**

Discrimination Index



88% answered correctly

Attempts: 17 out of 17

What is the output of the following program:

```
q = 2
i = 2
for i in range(5):
    print(str(q * i) + " ")
```

0 2 4 6 8	15 respondents	88 %	<div></div> ✓
4 6 8		0 %	<div></div>
6 8 10 12		0 %	<div></div>
4 6 8 10	1 respondent	6 %	<div></div>
No output, compiler error.	1 respondent	6 %	<div></div>

+0.57

Discrimination Index



88% answered correctly

Attempts: 17 out of 17

What is the output of the following code?

```
q = 2
i = 2
while (i > 2):
    i+=1
    if (i >= 0):
        print(str(q * i) + " ")
        i-=1
    else:
        break
```

This code results in an infinite loop	2 respondents	12 %	<div></div>
2 0		0 %	<div></div>
Syntax error		0 %	<div></div>
None of the options	15 respondents	88 %	<div></div> ✓

+0.6

Discrimination Index



88% answered correctly

# Functions

## Part 2



# Functions and Lists (1)

- ▶ Lists can be passed as arguments to functions.
  - ▶ Lists are mutable since list variables technically store references
- ▶ A common pattern is to operate on each list element in a loop

```
>>> odds = [1, 3, 5, 7, 9, 11]
```

```
>>> def sumList(inpList):
```

```
...     sum = 0
...     for i in inpList:
...         sum += i
...     return sum
```

```
>>> sumList(odds)
```

```
36
```

# Functions and Lists (2)

- ▶ Lists can be returned by functions.
  - ▶ Lists are mutable since list variables technically store references
- ▶ A common pattern is to operate on each list element in a loop

```
>>> def generateEvens(seqLength):  
...     evens = []  
...     for i in range(seqLength):  
...         evens.append(2 * i)  
...     return evens  
>>> generateEvens(12)  
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22]
```

# Functions and Tuples (1)

- ▶ Tuples provide some very useful interactions with functions.
- ▶ A function can be defined that takes varying numbers of parameters

```
>>> def average(*values):  
...     sum = 0  
...     for i in values:  
...         sum += i  
...     return float(sum/len(values))  
  
>>> average(27, 9)  
18.0  
  
>>> average(12, 18, 26)  
18.666666666666668
```

# Functions and Tuples (2a)

- ▶ Tuples provide a way to return multiple values from a function.
- ▶ A function can be defined that takes varying numbers of parameters

```
>>> def getName():  
...     firstName = input("First Name: ")  
...     lastName = input("Last Name: ")  
...     return (firstName, lastName)
```

# Functions and Tuples (2b)

- ▶ This function can either be assigned to a tuple or two separate variables in the main program.

```
>>> fullname = getName()
```

```
First Name: Amy  
Last Name: Jackson
```

```
>>> fullname
```

```
('Amy', 'Jackson')
```

```
>>> (firstName, lastName) = getName()
```

```
First Name: Paul  
Last Name: Benson
```

```
>>> firstName
```

```
'Paul'
```

```
>>> lastName
```

```
'Benson'
```

# Function Arguments (1)

- ▶ Functions usually determine argument assignment by position.
- ▶ Providing default values in the function allows a function to be called with less than the full amount of arguments

```
>>> def greetName(name, greeting="How's it going?"):
...     print("Hi " + name)
...     print(greeting)
```

- ▶ This function can be called with one or two arguments.

```
>>> greetName("Bob", "Nice tie!")
```

Hi Bob

Nice tie!

```
>>> greetName("Jenny")
```

Hi Jenny

How's it going?

# Function Arguments (2)

- ▶ You can also use arguments in any order if you explicitly set them.

```
>>> greetName(greeting="It's cold out, huh?", name="Sally")
```

```
Hi Sally
```

```
It's cold out, huh?
```

# Functions

Import Modules



# Functions and Modules

- ▶ Collections of functions can be stored in separate files for reuse in other programs.
- ▶ To include the functions in a module use the import command.

```
import turtle
```

```
import math
```

- ▶ Module functions must be prefixed with the module name and a dot (.).

```
math.cos()
```

```
turtle.getscreen()
```

# Aliasing Functions & Modules

- ▶ Use the as keyword to create a shorter name for the module.

```
import turtle as tur
```

```
import math as m
```

- ▶ Module functions must be prefixed with the module name and a dot (.).

```
m.cos()
```

```
tur.getscreen()
```

# Selecting Functions from Modules

- ▶ Use the `from` keyword to import specific functions from the module.

```
from myModule import myFunction1
```

- ▶ To import every function in a module use `*`

```
from myModule import *
```

- ▶ Functions can also be aliased using the `as` keyword.

```
from myModule import myFunction1 as fn1
```

- ▶ Note: When using `from`, you don't need to prefix the function names with the module name.

# Making your own modules

mathfuncs.py:

```
def wtSum(x,y,z) :  
    a,b,c; #these are local to f1  
    a = 1  
    b = 3  
    c = 5 #set some values  
    print("function 1 weighted sum: " + \  
str(a*x+b*y+c*z) + "\n")
```

main.py:

```
import mathfuncs  
mathfuncs.wtsum(2,4,6)
```

- ▶ Making modules is really easy. Just store your functions in a separate .py file in the same directory as the main.py file.
- ▶ Use the import methods as shown in the previous slides.
- ▶ Multiple modules can be collected into packages by collecting them in a directory and creating an index file called `__init__.py`

# Pragmatics

(Meta)Tools & Matters of Style

# Software Quality Criteria - Definitions

- ▶ Correctness
  - ▶ Does it produce the expected/required outputs?
- ▶ Robustness
  - ▶ Can it handle incorrect or unexpected run-time situations?
- ▶ Extensibility
  - ▶ How hard is it to change or update functionality?
- ▶ Clarity & Organization
  - ▶ Can someone (including you) understand what you did?
- ▶ Elegance
  - ▶ Is it beautiful? (factors above play into it + simplicity; subjective)

# Software Quality Criteria - Supporting Approaches

- ▶ Correctness
  - ▶ Test-First Methodology/Unit Tests
- ▶ Robustness
  - ▶ Exception Handling/Input Checking
- ▶ Extensibility
  - ▶ Encapsulation/Modular Design
- ▶ Clarity & Organization
  - ▶ Programming Conventions, Comments, Documentation
- ▶ Elegance
  - ▶ Develops over time; serendipity.

# Programming Conventions

- ▶ Encourage Consistency & Readability
  - ▶ Don't sacrifice flexibility and creativity
- ▶ Commonly Accepted (underscores & CamelCase) :
  - ▶ `variableName`
  - ▶ `CONSTANT, kVal1, kVal2;`
  - ▶ `file_name.*`, `fileName.*`, `FileName.*`
  - ▶ `FolderName`, `fileName`
- ▶ PEP 8 (<https://pep8.org/>) guidelines are good to utilize, but consistency and readability are the most important factors.



# Verification vs. Validation

## Verification:

- ▶ Did I get what I asked for?
- ▶ Evaluated against specification
- ▶ Various levels of test and debugging.
- ▶ Identify HW/SW components to re-implement or repair.

## Validation:

- ▶ Did I want what I asked for?
- ▶ Evaluated during use in target environment
- ▶ End-user engagement
- ▶ Identify HW/SW components to re-design or remove.

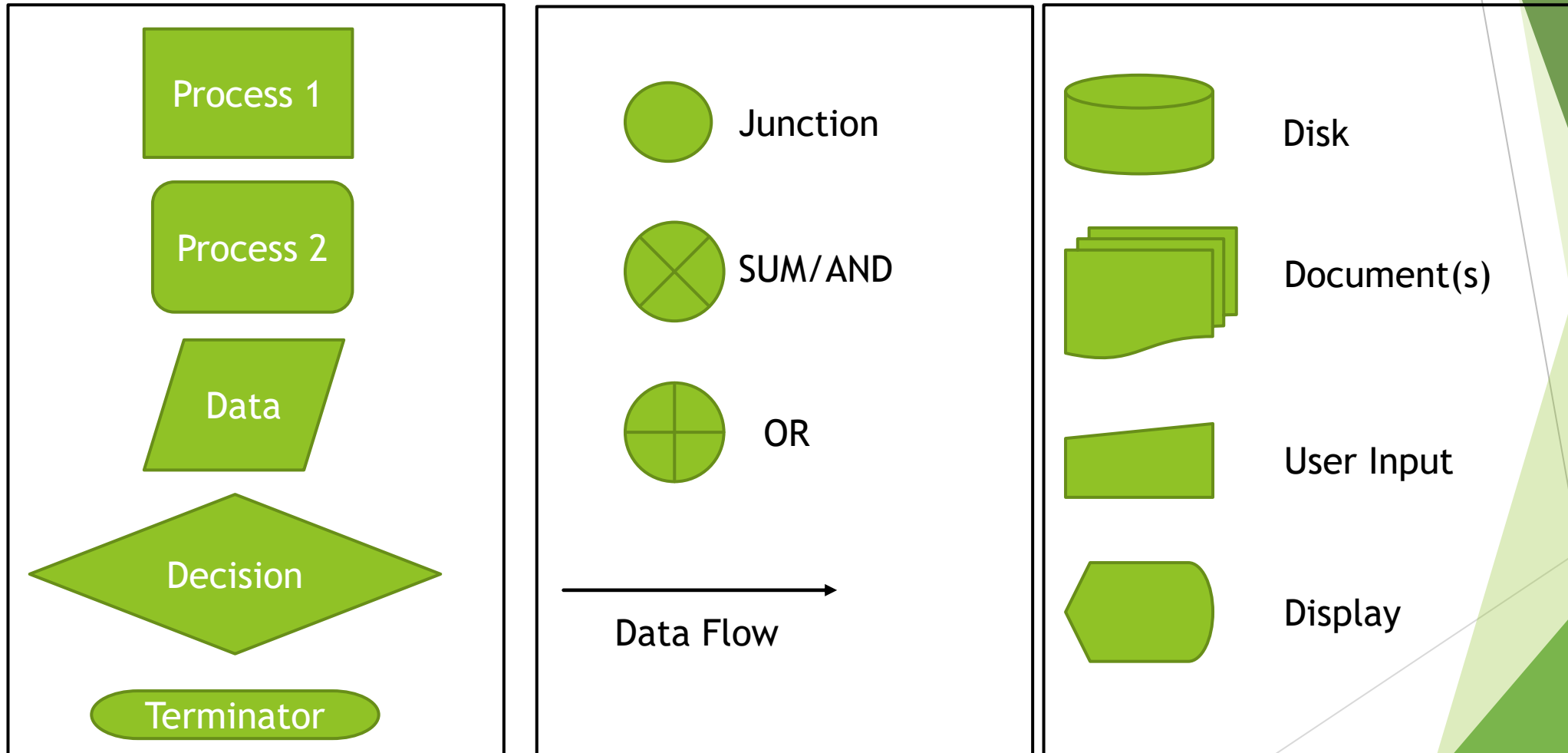
# Visualizing Processes

Flow Charts & UML

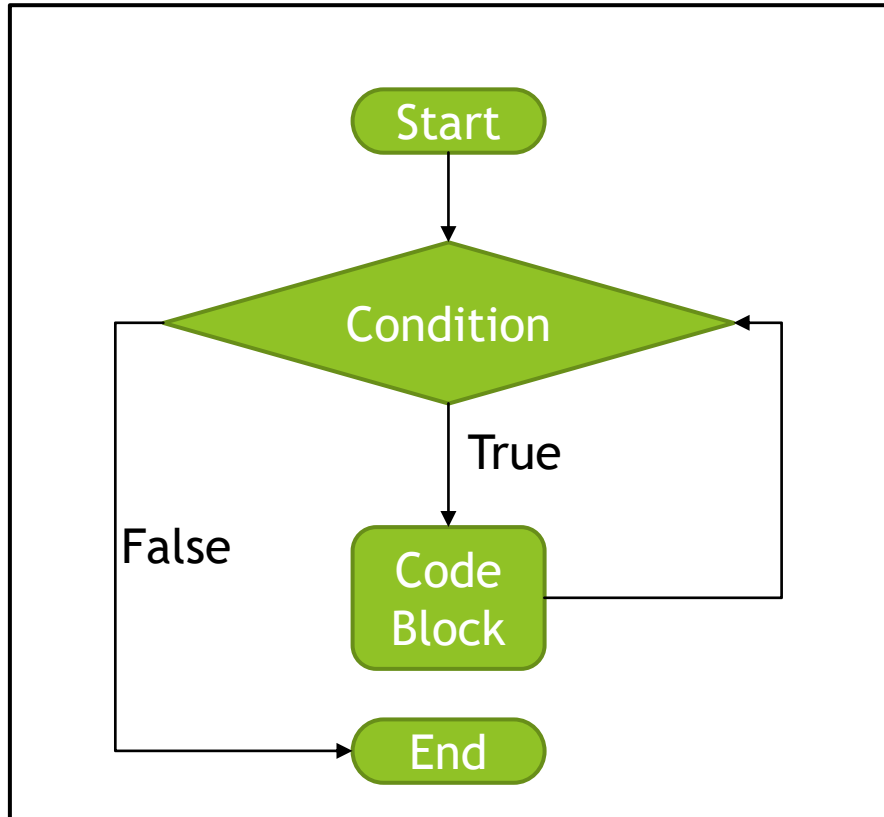
# Diagramming Languages - Overview & History

- ▶ Flow Charts (as flow process charts) date back ~100 years (Frank & Lillian Gilbreth, 1921)
- ▶ John von Neumann & Herman Goldstine applied flow-chart diagramming to software/algorithms in the 1940s.
- ▶ UML (Booch, Jacobson, Rumbaugh) started the design which was then defined and standardized by OMG (1997)
- ▶ ISO publishes and maintains the official language standard:
  - ▶ ISO/IEC 19505-1:2012 [ISO/IEC 19505-1:2012] (Reviewed 2017)

# Flowchart Symbols



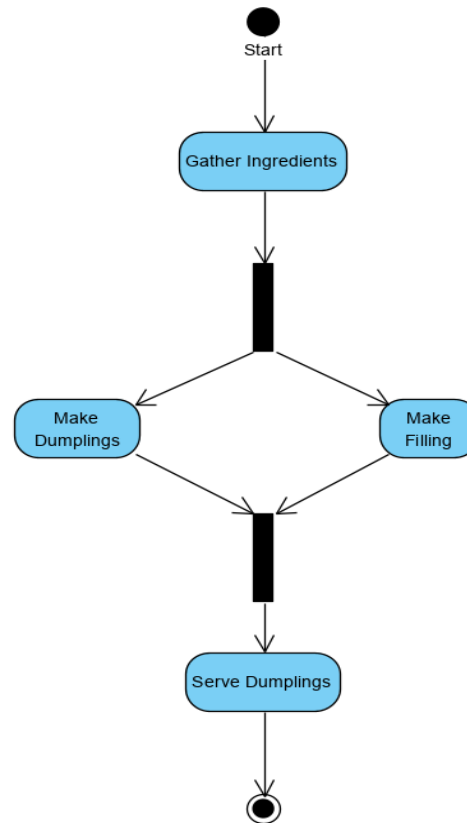
# While Loop



```
while (Condition):  
    #Code Block
```

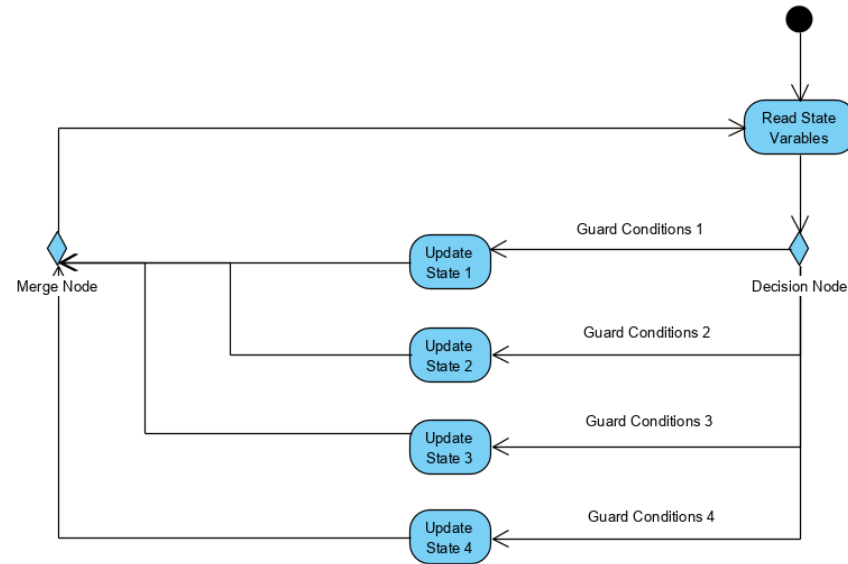
# UML Activity

- ▶ Adds a distinct symbol for start and stop
- ▶ Extends semantics with fork, and join symbols for parallel processes.



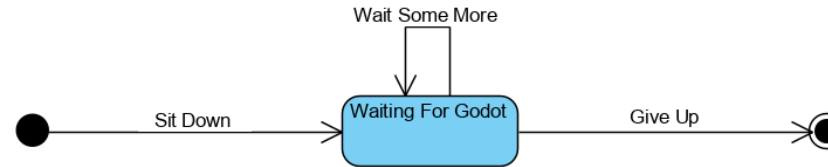
# UML Activity

- Adds merge and decision nodes to split and join control flows.
- Simplifies diagrams of switch statements and state machines.



# UML State Diagram

- ▶ Similar to activity diagram but provides more flexibility in creating state transitions, guards, and compound states.
- ▶ Nodes (rectangles) correspond to states not actions.
- ▶ Actions are indicated by transitions.



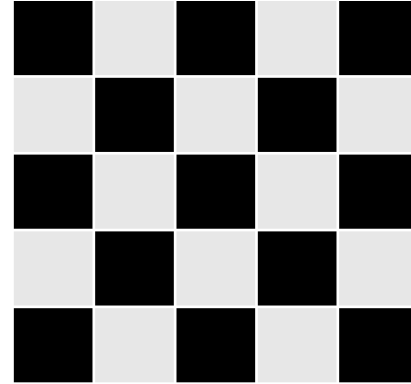


# Software Test Cases

Verification & Validation Concepts

# Designing Test Data

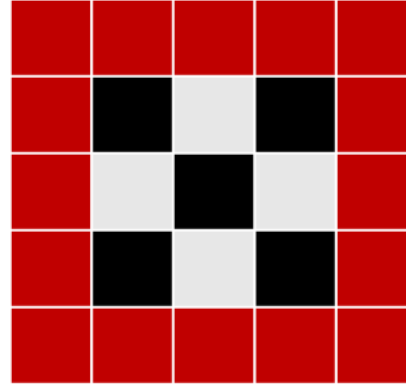
- ▶ Test cases that will cover the full range of the expected inputs
- ▶ Be sure to include partially exceptional cases
  - ▶ a.k.a. "Corner & Edge Cases"
- ▶ Try to test for common problems in input values BEFORE the program uses them for logic and arithmetic.
  - ▶ Types
  - ▶ Range
  - ▶ Filesystem & Memory
  - ▶ Format & Size
- ▶ If possible, provide a way to correct the input before fails.
- ▶ Python has language features specifically designed Exception Handling.



Grid based algorithms often compute cell values based on neighboring cells

# Designing Test Data

- ▶ Test cases that will cover the full range of the expected inputs
- ▶ Be sure to include partially exceptional cases
  - ▶ a.k.a. "Corner & Edge Cases"
- ▶ Try to test for common problems in input values BEFORE the program uses them for logic and arithmetic.
  - ▶ Types
  - ▶ Range
  - ▶ Filesystem & Memory
  - ▶ Format & Size
- ▶ If possible, provide a way to correct the input before fails.
- ▶ Python has language features specifically designed Exception Handling.



Edge cells are missing some neighbors so require special consideration

# Designing Test Data

- ▶ Test cases that will cover the full range of the expected inputs
- ▶ Be sure to include partially exceptional cases
  - ▶ a.k.a. "Corner & Edge Cases"
- ▶ Try to test for common problems in input values BEFORE the program uses them for logic and arithmetic.
  - ▶ Types
  - ▶ Range
  - ▶ Filesystem & Memory
  - ▶ Format & Size
- ▶ If possible, provide a way to correct the input before fails.
- ▶ Python has language features specifically designed Exception Handling.



Corner cells have even fewer valid neighbors and so need to be considered for additional special treatment

# Tests and Test Frameworks

- ▶ Manual Testing with Asserts
- ▶ Testing Frameworks/Libraries
  - ▶ Unittest (<https://docs.python.org/3/library/unittest.html>)
  - ▶ PyTest (<https://docs.pytest.org/en/6.2.x/>)

# Refactoring

Not All Improvements are Visible

# Debugging vs. Refactoring

## Debugging:

- ▶ Identify and correct errors in the source code
- ▶ Main Sources of Error:
  - ▶ Incorrect Syntax
  - ▶ Misunderstood Logic
  - ▶ Flawed Design
  - ▶ Unexpected Runtime Conditions
- ▶ Error Messages
  - ▶ Line number & Error Type
  - ▶ Errors cascade...Fix the problems on earlier lines of code first.

## Refactoring:

- ▶ Refactoring is a process of making small directed changes to improve software at the source code level (Fowler & Beck 2019)
- ▶ These changes may or may not have a visible impact on the behavior of the code from a user perspective but improve its maintainability and extensibility from the perspective of the creators
- ▶ One of the key proponents of this approach, Martin Fowler, maintains an online catalog of specific advice for refactoring techniques <https://refactoring.com/catalog/>
- ▶ IDEs and Plugins also produce automated suggestions for refactoring.

# Refactoring Quick Tips

## ▶ Avoid Duplications

- ▶ Loops
- ▶ Functions
- ▶ Parameterized Algorithms

## ▶ Break Up Large Functions

- ▶ Hierarchical Structure
- ▶ Assign Module Responsibilities
- ▶ Pre-conditions & Post-conditions

## ▶ Remove Dead Code

- ▶ Unreachable Conditions
- ▶ Old Attempts & Quick Debug Prints

## ▶ Remove Magic Numbers

- ▶ Extract Constants
- ▶ Header File
- ▶ Config File

## ▶ Avoid Nested Loops

## ▶ Use pre-built algorithms

- ▶ Standard Library
- ▶ Well-known Libraries
  - ▶ SciPy
  - ▶ NumPy



# References/Readings/Image credits

- ▶ Textbook Reading:
  - ▶ Horstmann, C. S., & Necaise, R. (2019). *Python for everyone*. John Wiley & Sons, Inc.
    - ▶ Related Chapters: 1 to 8
- ▶ Diagrams adapted from previous Florida Tech lectures by:
  - ▶ Dr. Carlos Otero
  - ▶ Dr. Stefan Joe-Yen
- ▶ Stock Photos from [Pexels](#)
  - ▶ Photo by [RODNAE Productions](#)
  - ▶ Photo by [ThisIsEngineering](#)
  - ▶ Photo by [Danny Meneses](#)
- ▶ Fowler, M. C., & Beck, K. (2019). *Refactoring: Improving the design of existing code*. Addison-Wesley.