

Python For Engineers III



Where we left off...

- Last workshop, we covered more advanced features of Python.
 - Extra data types.
 - Object-Oriented Programming.
 - Libraries.
- In this workshop, we're going to learn:
 - How to read syntax & runtime errors.
 - How to reference the Python Standard Library.
 - Using everything we've learned to solve real-life problems.
 - How to use the PIP program to add more libraries.
 - Basically, we're going to learn what real Software Engineers do often on the job.

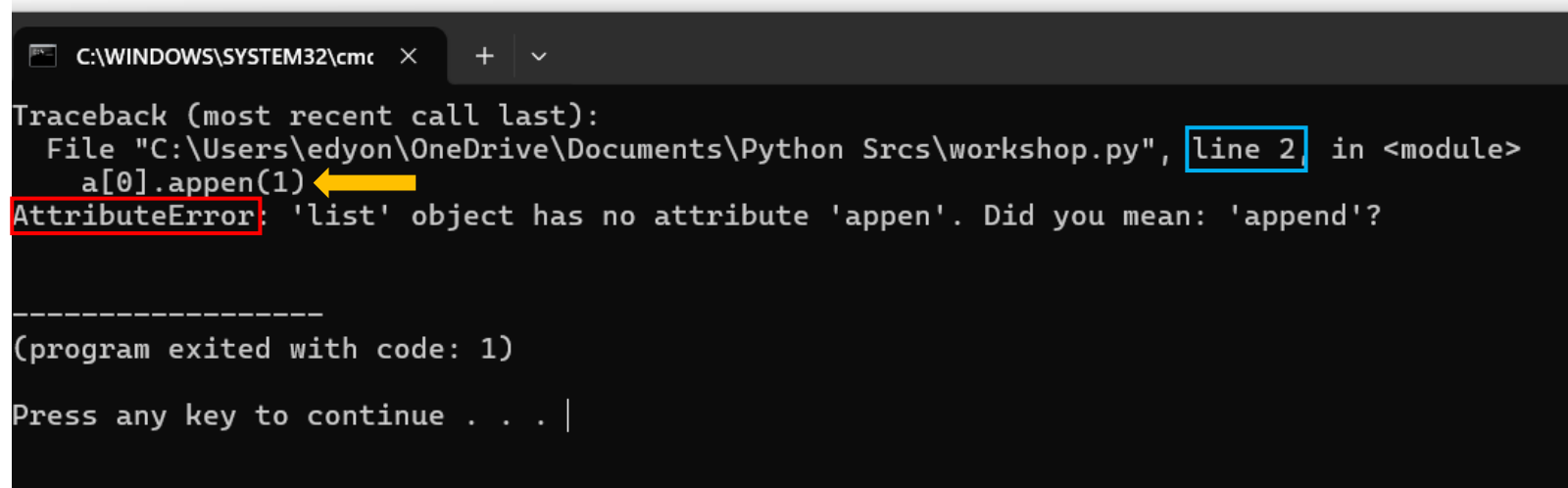
Exploration of the Python Standard Library

- Python's motto when it comes to libraries is: “batteries included”.
- Just don't ask what the voltage is.
- What is a library?
 - pre-existing code that others have written and ready to be used, simply requires importing.
- [Python 3 Standard Library Reference Link](#)

How to Read & Comprehend Errors I

- **Step 1:** Read type of error.
 - **NameError,**
AttributeError,
ValueError, etc
- **Step 2:** Check for the line number.
- **Step 3:** Investigate and try to resolve the error based on the error type.
- **Step 4:** practice making errors.
- **DON'T forget, these errors can be used in try-excepts!**

```
a = [[]] * 5
a[0].appen(1)
print(a)
```



A screenshot of a Windows command prompt window. The title bar shows 'C:\WINDOWS\SYSTEM32\cmd'. The command prompt displays a Python traceback error. The text is as follows: 'Traceback (most recent call last):', 'File "C:\Users\edyon\OneDrive\Documents\Python Srcs\workshop.py", line 2, in <module>', 'a[0].appen(1)', 'AttributeError: \'list\' object has no attribute \'appen\'. Did you mean: \'append\'?'. A yellow arrow points to the line 'a[0].appen(1)'. Below the error message, it says '(program exited with code: 1)' and 'Press any key to continue . . . |'. The error message 'AttributeError' is highlighted with a red box, and 'line 2' is highlighted with a blue box.

```
C:\WINDOWS\SYSTEM32\cmd x + v
Traceback (most recent call last):
  File "C:\Users\edyon\OneDrive\Documents\Python Srcs\workshop.py", line 2, in <module>
    a[0].appen(1)
AttributeError: 'list' object has no attribute 'appen'. Did you mean: 'append'?

-----
(program exited with code: 1)
Press any key to continue . . . |
```

How to Read & Comprehend Errors II

```
bad_num = int('abc')
```

```
C:\WINDOWS\SYSTEM32\cmd  x  +  v
Traceback (most recent call last):
  File "C:\Users\edyon\OneDrive\Documents\Python Srcs\workshop.py", line 1, in <module>
    bad_num = int('abc') ←
ValueError: invalid literal for int() with base 10: 'abc'
```

```
a = [[]] * 5
a[0].append(1)
print(b)
```

```
C:\WINDOWS\SYSTEM32\cmd  x  +  v
Traceback (most recent call last):
  File "C:\Users\edyon\OneDrive\Documents\Python Srcs\workshop.py", line 3, in <module>
    print(b) ←
NameError: name 'b' is not defined
```


Using PIP: Python Index Packager

- Go to a terminal whether Windows, MAC, or Linux.
- Type 'pip'.
- If this doesn't work, try: **python -m pip <command>**

```
PS C:\Users\edyon> pip
```

```
Usage:
```

```
pip <command> [options]
```

```
Commands:
```

```
install  
download  
uninstall  
freeze  
inspect  
list  
show  
check  
config  
search  
cache  
index  
wheel  
hash  
completion  
debug  
help
```

```
Install packages.
```

```
Download packages.
```

```
Uninstall packages.
```

```
Output installed packages in requirements format.
```

```
Inspect the python environment.
```

```
List installed packages.
```

```
Show information about installed packages.
```

```
Verify installed packages have compatible dependencies.
```

```
Manage local and global configuration.
```

```
Search PyPI for packages.
```

```
Inspect and manage pip's wheel cache.
```

```
Inspect information available from package indexes.
```

```
Build wheels from your requirements.
```

```
Compute hashes of package archives.
```

```
A helper command used for command completion.
```

```
Show information useful for debugging.
```

```
Show help for commands.
```

```
PS C:\Users\edyon> pip install numpy  
>> pip install scipy  
>> pip install sympy
```

Creating Larger Python programs

- Until now, we've made single-file Python programs.
- Single-file Python programs are alright but **unrealistic in the long run**.
- Sometimes programs are more complex in scale.
 - Requires breaking up into systems of independent actions.
 - This is where **software engineering** comes in.
 - Knowing how and what best practices to do when designing larger-scale software systems.
 - Systems Engineering practice also helps here!
 - Think: What's the purpose? And Who's the customer?
- Tip: **Isolate higher-level actions into systems.**
 - Higher Level System = does something unique that other systems do not do.
- **Break down each system into smaller, manageable pieces.**
- **The manageable pieces will have clearly defined inputs and outputs.**
- **At lower levels, each manageable piece will have defined behaviors that describes how to use the inputs and outputs.**
- Packages are folders of python scripts that you can import.
- Packages can also be custom-made libraries that you use as a system of itself or use it make parts of systems.
- Organize each system as *packages*.

Setting up structured Python programs

The image shows a code editor window with a file explorer on the left and a file explorer on the right. The code editor displays the following code:

```
1 from my_pkg import module1
2 module1.greet('kevin')
```

The file explorer on the left shows the directory structure of the project, including the `src` directory and the `my_pkg` package. The file explorer on the right shows the contents of the `my_pkg` package, including the `__init__.py` file and the `module1.py` file.

The code editor window has a tab bar at the top with the following tabs: `Documents`, `design.md`, `grammar-and-specification.md`, `go_parser.go`, `Fdr.h`, `Fdr.cpp`, `Iridium1.61.ino`, `KIRCHHOFF.py`, `main.c`, `node.h`, `workshop.py`, and `main.py`. The `main.py` tab is selected.

The file explorer on the left shows the following directory structure:

- `C:\Users\edyon\OneDrive\Doc`
- `Example Python Project`
- `src`
- `my_pkg`

The file explorer on the right shows the following directory structure:

- `C:\Users\edyon\OneDrive\Doc`
- `Example Python Project`
- `src`
- `my_pkg`
- `__init__.py`
- `module1.py`

Starter Python Scripts

- This code is in `main.py`
- `__name__` refers to the current python script file.

```
from my_pkg import module1
```

```
# this if-statement runs if  
# this py file is the starter script.
```

```
if __name__ == '__main__':  
    → module1.greet('kevin')
```

Word Problem Example & Practice I

- Make a function that prompts the user for an integer.
- If the given input can't be converted to int, ask the user to try again.
- First part isn't too bad:

```
def get_int_from_input(msg: str):  
    → str_input = input(msg + ': ')
```

Word Problem Example & Practice II

- But what happens if the user fat-fingers the input and gives a letter by accident?
 - We get a **ValueError** runtime exception!
- Well don't we just use an exception to handle this? **Yes**
- One problem though: Still doesn't ask our user to try again!
- **Function *prints* for the user to try again but doesn't restart!**

```
def get_int_from_input(msg: str):  
    → str_input = input(msg + ': ')  
    → try:  
        → i = int(str_input)  
    → except ValueError:  
        → print(f'"{str_input}" couldn\'t be  
            converted to int, try again.')  
    → else:  
        → return i
```

Word Problem Example & Practice III

- So how do we make sure the function restarts if an exception happens?
There's two options to do this:

- We could recursively call the function until the user gives valid input.
- Put the whole code inside an infinite loop.

- Doing things recursively sometimes is simpler and/or more efficient.
Here it's neither.

- Go with infinite loop.

- Why? Because simplicity.

```
def get_int_from_input(msg: str):  
    while True:  
        str_input = input(msg + ': ')  
        try:  
            i = int(str_input)  
        except ValueError:  
            print(f'"{str_input}" couldn\'t be  
                  converted to int, try again.')  
        else:  
            return i
```

Word Problem Tips

- Make sure you're understanding the problem.
- Determine whether a function itself should handle an error or if it should “bubble up” (have the function return an error and let higher-level systems deal with it).
- Determine the inputs and outputs.
 - Recall that Python's functions and methods can return more than one thing.
 - *If* you need to return anything that is.
- **Break down the problem**, start simple, into logical steps.
- Once you have each logical step, try to **convert the logical steps into code** [this is for any programming language].
- Remember to think of and handle edge cases.
- If there are duplicate sections of code, it might be worth putting that code into a function and replacing the duplicate section with a function call.

Abstract Word Problem

- Make a program that asks the user for 3 coefficients of the values in the quadratic formula: $Ax^2 + Bx + C$
- Best to make this as a function so we can return two x values.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Practice Project: Hangman

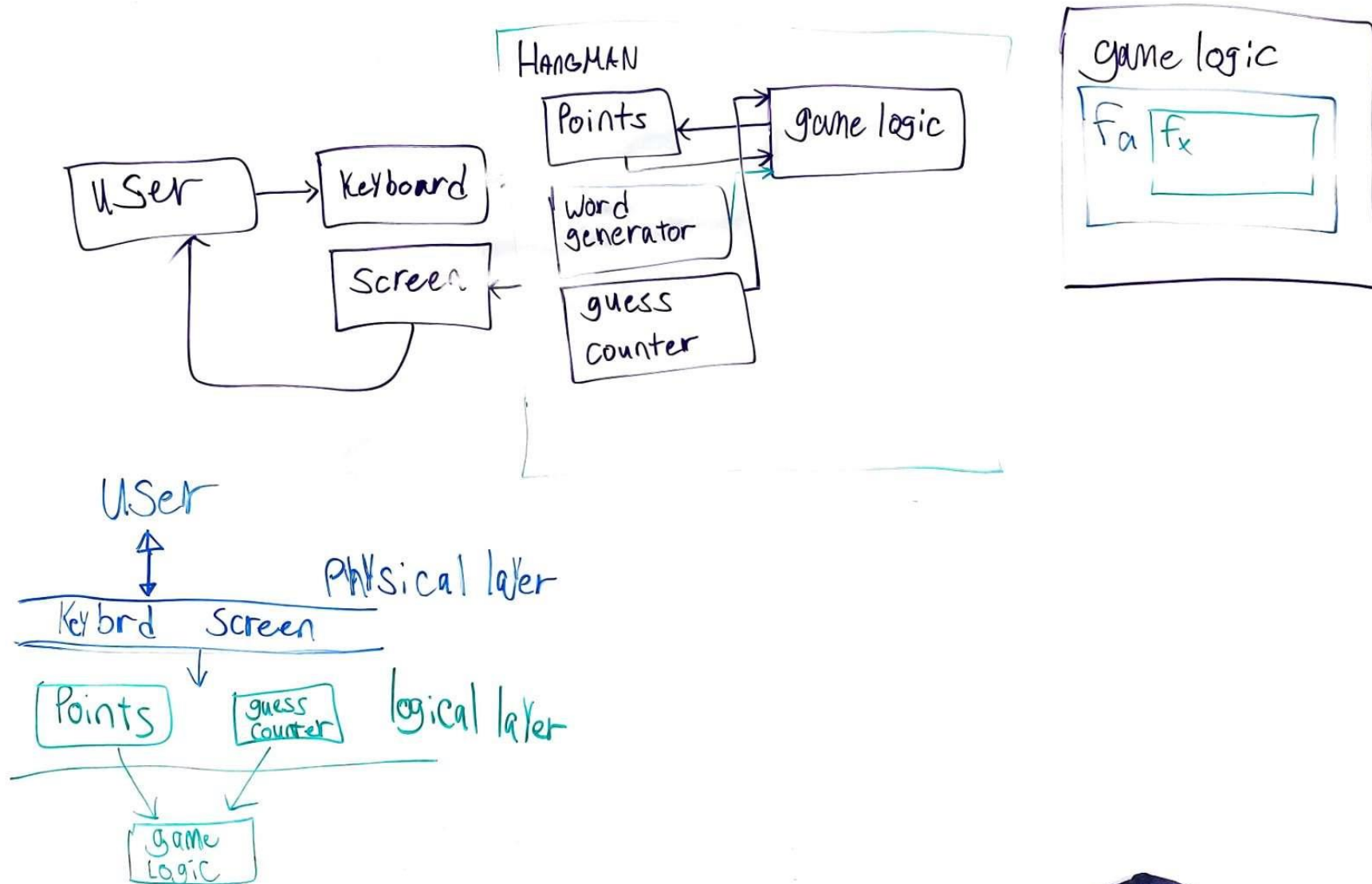
- Requirements:

- The game will prompt the player for a letter as a guess from input.
- The player is allowed 6 total wrong guesses.
- For each letter in the mystery word matching the player's letter, the player gains \$100.
- If the letter is a vowel, charge the player \$50.
- The player is not allowed to go into debt.
- The player cannot use the same letter they previously guessed.
- If the player got 6 wrong guesses in total or other situations that make the game stuck, the player then loses and prompted to either play again or quit.

- Hints:

- Try to break this up into one or more systems.
- Use dictionaries and/or classes if needed.
- Remember to use the Python Standard Library for anything you might need.
- What's the user going to see? What do you expect the user to do?
- Think user experience & think as a user not as a developer.

Hangman System Block Diagram



End of Python For Engineers III

- Thank you for attending.
- Next Time: **Python for Engineers IV.**
 - Implement Numerical Methods from Calculus in Python.
 - Learn about Numpy, SciPy, and SymPy.

