Python Basics and Problem Solving Strategies for Python Developers

Introduction

Welcome to the lesson on Python basic concepts and problem-solving strategies for Python developers. in this lesson, we are going to introduce some of the features and concepts of the Python language. In addition, we will go over the strategies that we use as developers to effectively produce solutions. Problem-solving is a fundamental skill. Whether you're building web applications, data analysis tools, or machine learning models, you'll encounter complex problems that require creative and effective solutions. This foundational lesson will be a framework designed to change the way to think about a problem.

Table of Contents

Python Basics

Python Vocabulary

- **Syntax** - Syntax refers to how code is written. That means the spacing, use of punctuation or characters, use of curly braces, and/or indentation. Python's syntax, while easily readable compared to other languages, is still strict. A single syntax error in code will cause a Python program to execute completely differently or trigger an error, which halts code execution.

- **Variable** - A key component to any programming language, a variable is a stand-in for a certain piece of data, whether it is a static data type or the result of a function. Like you learned in Algebra with "x" as the usual stand-in for a value, variables in Python work the same way but are more powerful. A variable name in Python will typically be a lowercase word or series of words separated by the underscore character (`_`). We don't want to use single letters or cryptic words for variables as it is considered bad practice (modern code can contain thousands of variables); it is best practice to be *descriptive*.

  Variables are created or *assigned* with the assignment operator or `=`. Unlike you see in math where equations use `=` to illustrate equality, in python a single `=` is used to assign variables while `==` is used to compare things or check for equality.

In [ ]:
```
my_variable = 25
```

- **Comment** - Typically preceded in a code block with `#`, a comment is text ignored by the computer that we can use to provide additional information to ourselves or other programmers about a certain section of code.

In [ ]:
```
# This line will not run
my_list = [1,2,3,4,5] # <--- Everything left of the '#' will be executed
print(my_list)
```

```
[1, 2, 3, 4, 5]
```

- **Function** or **Method** - Functions and methods can be thought of as little programs or apps. they take specified input (nothing can be an input...), do some action, and then usually provide an output or "return" something.

  Functions are first *defined*, which means they are written out with all the included code inside, and then they are *called* with the specific inputs, if any, to be executed.

In [ ]:
```
# Function DEFINED here
def find_name(target_name, input_data):
    present = False
    name_location = input_data.find(target_name)
    if name_location != -1:
        present = True
    return present

long_string_of_data = "Jim Wallace, Bill Smith, Frank Scott, Deborah Randall, Willie Nelson, Kevin Zhang, Reah Nelson, Beth Doran, Aaron Wood, Chance Rooney, Jamal Free

# Function CALLED here
find_name("Aaron Wood", long_string_of_data)
```

Out[ ]: True

- **Argument** or **Parameter** - The technical term for the "input" to a function. in the example above, the parameters are `target_name` and `input_data`

- **Library** or **Module** - A library is a collection of code that defines certain methods and also could define the nature of special data types. For example, There is a `datetime` library that defines the behavior of `datetime` objects made to calculate or track dates or time passage. Libraries are imported at the top of a python file and once the import statement is read, the methods in that library are able to be accessed.

Reserved Keywords

- Python: Keywords

These are special words that Python already recognizes for special use cases and cannot be used, or re-assigned as a variable name. In an IDE such as a Jupyter Notebook, they usually have some sort of special formatting that gives a visual cue. See the example below for Python reserved keywords.

In [ ]:
```
# reserved Python keywords - run this cell to see all Python keywords
import keyword

print(keyword.kwlist)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

Built-in Methods

- Python: Built-in Functions

Some words have a special appearance when presented in an IDE such as a special color, or *italics*. They will also have `()` or `(some_parameter)` following the word(s). these are examples of built-in methods. Built-in methods are the exact same as methods (or functions) that you write yourself, but they are in the code behind the scenes. You just have to call them. The most prominent built-in method that you have already seen (in the cell above) is the `print()` method. In Python source code, there is a function written called `print()` that takes a parameter and its purpose is to present whatever the parameter is as the output.

Just like you would call `find_name("name")` in the function defined above and use the output, you call `print(whatever_you_want)` to show whatever you want!

Problem Solving Strategies

Understanding the Problem

Problem Statement Analysis

- Understand the problem statement thoroughly.
- Identify input, output, and expected behavior.
- Clarify ambiguities and constraints.
- Create an example or use case to illustrate the problem.

Identifying Constraints and Requirements

- Identify constraints on time, memory, or other resources.
- Consider special requirements or edge cases.
- Understand the problem's context and potential real-world impact.

Decomposition

Breaking Down Complex Problems

- Divide complex problems into smaller, more manageable parts.
- Define the main components of the problem.
- Prioritize subproblems and plan your approach.

Identifying Relevant Data and Processes

- Determine what data and operations are required to solve each subproblem.
- Sketch a high-level plan of how subproblems interact.
- Visualize the problem structure.

Algorithm Design

Selecting Appropriate Data Structures

- Choose the right data structures to represent problem components.
- Evaluate the trade-offs of different data structures (lists, dictionaries, sets, etc.).
- Optimize data access and manipulation.

Creating Algorithms to Solve Subproblems

- Design algorithms for each subproblem.
- Use pseudocode or flowcharts to outline your approach.
- Optimize the efficiency of your algorithms.

Coding and Debugging

Translating Your Algorithm into Python Code

- Write clean, modular, and readable code.
- Follow Python coding conventions (PEP 8).
- Use meaningful variable names and comments.

Strategies for Debugging and Troubleshooting

- Apply debugging techniques (print statements, debugging tools).
- Handle exceptions and errors gracefully.
- Fix issues systematically.

Documentation and Collaboration

Writing Clear and Concise Code Comments

- Document code to make it understandable to others.
- Describe the purpose of functions and complex logic.
- Include usage examples when relevant.

Collaborating Effectively with Team Members

- Communicate changes and updates with your team.
- Use version control systems (e.g., Git) to manage code.
- Foster a collaborative development environment.

Optimization

Analyzing Code for Performance Bottlenecks

- Profile your code to identify performance bottlenecks.
- Apply algorithmic and code-level optimizations.
- Balance between time and memory efficiency.

Iteration and Refinement

Continuously Improving Your Solution

- Seek feedback from peers and mentors.
- Iterate on your code to make it more elegant and efficient.
- Update documentation to reflect changes.

Example Problem Solving Exercise

In the paragraph below, write a script to identify the longest sentence by word count.

```
'''
Nestled in the heart of the Pacific Northwest, Washington State is a mesmerizing blend of natural wonders and urban sophistication. Its geography is a
tapestry of contrasts, stretching from the rugged Pacific coastlines to the dense, verdant rainforests of the Olympic Peninsula, culminating in the
majestic Cascade Mountains. Among these peaks, Mt. Rainier stands as a sentinel, an active volcano and the pinnacle of the state's natural beauty. This
diverse terrain provides a playground for adventurers and nature lovers alike, offering an array of activities from serene hiking trails and challenging
ski slopes to tranquil kayaking routes. Beyond its scenic landscapes, Washington's ecological diversity is a testament to its environmental richness,
hosting an array of wildlife that includes the majestic orcas navigating the Puget Sound and elusive elk roaming the shadowy rainforests. This harmonious
blend of natural beauty and wildlife creates a unique backdrop for the state, inviting both residents and visitors to immerse themselves in its outdoor
wonders. Washington State, through its stunning geography and vibrant ecosystems, stands as a beacon of the Pacific Northwest's splendor, embodying the
essence of adventure and the tranquility of nature in one cohesive landscape.
'''
```

In [ ]:
```
paragraph = """Nestled in the heart of the Pacific Northwest, Washington State is a mesmerizing blend of natural wonders and urban sophistication. Its geography is a ta

# Split the paragraph on "."
sentences = paragraph.split(".")

longest_sentence = ""

# For each sentence...
for sentence in sentences:

    # Count words and compare to the longest sentence we've seen so far
    if len(sentence.split(" ")) > len(longest_sentence.split(" ")):

        # If it's the longest, hold on to it
        longest_sentence = sentence + "."

# Finally, print the longest sentence and its word count
print(longest_sentence)
print(len(longest_sentence.split(" ")))
```

```
 Beyond its scenic landscapes, Washington's ecological diversity is a testament to its environmental richness, hosting an array of wildlife that includes the majestic o
rcas navigating the Puget Sound and elusive elk roaming the shadowy rainforests.
36
```

Hard-Coding vs. Programmatic Coding

Hard-coding is when you type explicit values in your code. *Programmatic coding*, is using variables and methods wherever possible. By writing programmatic code, you make your scripts useful to others as well as yourself. For example, let's say you want to plan the cost of a dinner for a large group. You could write code like this:

In [ ]:
```
total_dinner_cost = 50 * 8.00 + 50 * 25.00 + 50 * 4.00 + 20 * 10.00

print('Number of diners:', 50)
print('Cost of dinner:', total_dinner_cost)
```

```
Number of diners: 50
Cost of dinner: 2050.0
```

Now, we just received some late additions to our list of attendees. This requires that we adjust the code in several places, and remember what each number's significance is for the calculation.

What would happen if the price of one of the dinner options changed? What if the calculations were more complex, like more dinner options? What if another number in the calculation was the same as the number of attendees? How could we organize the data in a more freindly way? We could start by defining certain components as variables, like the number of diners.

In [ ]:
```
num_diners = 61
num_alc_option = 20

appetizer = num_diners * 8.00
main_course = num_diners * 25.00
dessert = num_diners * 4.00
alc_drink = num_alc_option * 10.00


total_dinner_cost = appetizer + main_course + dessert + alc_drink
print('Number of diners:', num_diners)
print('Cost of dinner:', total_dinner_cost)
```

```
Number of diners: 61
Cost of dinner: 2457.0
```

What if this code was just a small portion of a larger code base that calculated dinner costs per week over several venues? Programmatic coding allows for faster changes, reduces errors, and allows for quicker integration into more complex code bases. Back to the dinner, what if we scaled it up to feed 650 people? If we're hard-coding, we'd have to go in a swap in `650` everywhere we see the value `61` in our code above. With a script this small, it is a fairly easy task to make the changes and be confident they were all addressed. What if this number was used in 200 other places in a larger code base? Will you get them all? Variables eliminate that problem.

Conclusion

In this lesson, you learned some basic Python concepts that are important to know as you start to learn the foundational pieces of programming with the language. You also learned essential problem-solving strategies for Python developers. These skills are invaluable for tackling complex coding scenarios in real-world projects. Remember that practice, collaboration, and continuous learning are key to becoming a proficient problem solver and developer. Keep coding, keep solving, and keep building amazing things with Python!

Practice

- Data Type Practice
- First and Last Name