“Programming in Python - A Summary”

written by: John Clement J. Ibrahim, Cybersecurity (Group 1)

Programming is the *process of writing human legible words into something a computer understands*. Think of it like trying to teach a blind person to understand a written book. They wouldn’t understand one if you gave it the same way you always read it, so you give them one in braille, something they’ll understand. In a sense, this is the way we program. We use human readable code (in an IDE) so that it can be converted (with a compiler) to something a computer can read (to binary 1s and 0s). Then it does that process in reverse, so that we see the output of what we created.

There are three mainly used paradigms, or types of programming, commonly used: **procedural, event-driven, and object-oriented (OOP)**. Procedural programming, as the name suggests, is programming structured in a way so that the program proceeds on a line-by-line basis. This is also known as *single threading*. This makes the program extremely easy to write and debug, as the process is a singular line of thought, but it tends to make larger programs slower as there is more to process in one block of code. This is the way beginners typically begin coding, as it is straightforward with minimal risk of something going wrong in unexpected ways. An error in line 45 will always be an error at line 45.

Event-driven programming is where you create inputs for the user to interact with your program, it can be something as simple as a key stroke or something as complex as needing a password and email to login. This type of paradigm is typically applied to the front-end development cycle, as the things the users will see, such as graphical user interfaces (GUIs) or user experience (UX) for games or real-time applications. It is also used in game development to dictate the movement inputs and events that must happen in sync with player inputs.

Object-oriented programming (OOP) is using containers, called classes, to combine data and attributes into one neat block of code. Using OOP, we can manage large-scale projects where lots of different things need to interact with each other in multiple different ways, and promote code reusability and recycling. You can mix and match these different paradigms to suit what is needed, as there is never one way to do things.

To build a program, we need a way to store and manipulate information. This is where **variables** come in. Think of a variable as a labeled box. You put a value inside the box, and you give the box a name, or an **identifier**. Whenever you want to use that value, you just refer to the box's name. For example, in a game, you might have a box named *player_health* that holds the value *100*. When the player gets hurt, you can simply update the value inside that box to *75*.

In many programming languages, there's a complex concept called **pointers**, which are like ID tags for boxes that can lead to other boxes. Fortunately for beginners, Python largely shields you from this complexity. In our analogy, when you work with variables in Python, you're just writing the name on the box itself; the language handles the behind-the-scenes ID tagging for you, making it much more straightforward.

Now, what can we put inside these boxes? The most common things are the basic data types:

- **Integers** (`*int*`): Whole numbers, like `*-5*`, `*0*`, or `*42*`. Perfect for counting things, like the number of lives in a game (`*lives = 3*`).
- **Floating-Point Numbers** (`*float*`): Numbers with decimal points, like `3.14` or `-0.001`. You'd use this for precise values, such as a player's speed (`*speed = 5.7*`) or the calculation of a bill total including tax.
- **Strings** (`*str*`): Sequences of characters enclosed in quotes. They represent text, like a player's name (`*player_name = "John*"`) or a dialogue line in an adventure game.
- **Booleans** (`*bool*`): Represents only one of two values: `True` or `False`. This is the fundamental data type for decision-making. Is the door `locked`? `*True*`. Is the power-up `active`? `*False*`.

These basic types are built-in data types, meaning Python understands them from the get-go. As you advance, you can create user-defined data types using classes (from OOP). If the basic types are like single tools (a hammer, a screwdriver), a class is a custom-made toolkit designed for a specific job, like a "Player" toolkit that contains a health integer, a name string, and an inventory list.

Sometimes, you have a value that should never change, like the value of Pi (`π`) or the gravitational constant in a physics simulation. For this, we use **constants**. A constant is like a variable whose box you superglue shut. By convention in Python, we write their names in all capital letters (e.g., `*MAX_SPEED = 100*`) to signal to other programmers, "This value is not meant to be changed."

Data on its own is inert. To make our programs dynamic, we need tools to manipulate that data and control the program's flow. This is done with operators and conditional statements. Arithmetic operators are the ones you know from math: `+` for addition, `-` for subtraction, `*` for multiplication, and `/` for division. They work exactly as you'd expect on numbers. You can use the mathematical rule of PEMDAS (parenthesis, exponent, multiplication, division, addition, subtraction) to work out logic for your computations. The line "*result = 5 * 5 ** 2*" would look much different than the line "*result = (5 * 5) ** 2*", for instance.

Comparison operators compare two values and give you a Boolean (`True` or `False`) answer. Is `*10 > 5*`? `True`. Is `*player_score == 100*`? (The double equals `==` checks for equality). This becomes powerful when combined with conditional statements.

An `if` statement is the program's way of asking a question. It's like a fork in the road.

```
if weather == "raining":
    print("Take an umbrella!")
```

Here, `weather == "raining"` is the condition. If it's `True`, the code indented underneath runs. You can extend this with `else` to provide an alternative.

For more nuanced decisions, you use `elif` (short for "else if").

```
if weather == "raining":
    print("Take an umbrella!")
else:
    print("Wear sunglasses!")
```

You can even put `if` statements inside other `if` statements, creating *nested ifs*. For example, before printing "It's a hot day," you might check another condition: `if temperature > 30 and humidity > 80: print("It's also humid!")`.

That word `and` is crucial. It's a *logical operator* used to create **compound conditions**. They allow you to check multiple things at once. `and` requires both conditions to be true. `or` requires only one to be true. `not` flips a Boolean value. They are the logic gates of your program.

Finally, what if you need to do something over and over? You use **loops**. A `for` loop is used when you know how many times you want to repeat something, like iterating over every item in a shopping list.

```
shopping_list = ["bread", "milk", "eggs"]
for item in shopping_list:
    print(f"Don't forget to buy {item}!")
```

A `while` loop, on the other hand, repeats as long as a condition is true. Think of it as "while this is happening, keep doing that." It's like telling a friend, "While I'm away, keep feeding my cat." The loop only stops when you return (or the condition becomes `False`).

```
hungry = True
while hungry:
    # Code to eat a snack
    if snacks_eaten == 5:
        hungry = False
```

By combining these fundamental building blocks—variables, conditionals, and loops—we can construct the intricate logic that brings our software to life, from a simple calculator to the complex worlds of a video game. Programming fundamentals like these, whether it be in Python, Java, C++, or whatever language you choose, is the basis of forming a good habit when it comes to programming. Making code easily legible prevents confusion, making code compact and concise makes debugging easy, making code correctly formatted makes passing on code easier, so on and so forth. Choosing the correct types of code and when to use different techniques is important when making large projects.