Module 6. Errors, Debugging, and Testing

Errors

Errors are an inherent part of the programming journey, and Python is no exception. These errors can range from simple syntax mistakes to more complex logical flaws.

Traceback

Whenever Python runs into an error, it prints this line first:

```
Traceback (most recent call last):
```

This announces what's coming next: *a traceback of what went wrong.*
This is also known as a **stack trace.**

Error Types

There is an error called a **TypeError.** Here, Python is telling us about what kind of error it encountered.

For example, if we tried to multiply two strings, which makes no sense to Python, this will result in a TypeError and will be displayed like this:

```
TypeError: can't multiply sequence by non-int of type 'str'
```

Other common errors:

- **SyntaxError =** You made a typo or indentation mistake
- **ImportError =** You are trying to import an error that does not exist.
- **IndexError =** You are trying to access a spot in a list that does not exist.

- **ValueError =** You tried to call a function with arguments of the wrong type.

How to handle errors:

There's *a way to handle errors* so that your program may *retry, log or report the error* to the user in another way and continue.

The ***try...except*** construct in Python allows you to deal with errors or exceptions in a controlled manner.

It works in the following way:

You enclose the code that might generate an error within the **try** block. *If an error occurs*, instead of crashing the program, it jumps to the **except block** that handles that specific type of error.

Debugging

To get rid of bugs:

- Identify the direct cause of the crash.
- Check your assumptions.
- Fix the bug.

Debugging at runtime

In Python, debugging at runtime refers to the process of identifying and resolving issues or errors in a running program.

Print statements can be used for debugging where we insert print statements throughout the code to display the values of variables or to track the execution flow.

API (Application Programming Interface) = is a way for two or more computer programs to communicate with each other.

Logical breakpoints in the application can also be used to debug the issue at runtime.

Logical breakpoints allow you to pause the execution of your code at specific points or conditions.

Testing

Python tests are a way to verify the correctness and functionality of your Python code. They help ensure that your code is as expected and meets the specified requirements.

Assert = In Python, the **assert statement** is used to perform a simple form of testing within the code. The purpose of using assert is to catch potential errors or mistakes in the code during development and debugging.

Pytest = Pytest is a popular tool that makes it easy to write tests using assertions. You can install it using pip.

In the following command, **-U** is an *option* that tells pip to upgrade pytest to the latest version if you happened to already have a version installed.

👉 **pip install -U pytest**

Test Cases

```
test_initials_common_name()
```

This test case verifies the behavior of initials() for a common name scenario where the input name such as "Daniel Radcliffe". It asserts that the output of initials('Daniel Radcliffe') should be equal to 'D. R.'

```
test_initials_double_barrelled()
```

This test case checks the behavior of initials() for a name with a double-barreled last name. It asserts that the output of initials, for example, ('Helena Bonham Carter') should be equal to 'H. B. C.'.

These test cases use the *assert statement* to *compare* the actual output of initials() with the expected output. If the assertion fails, indicating a mismatch between the actual and expected values, an **AssertionError** will be raised.