Explore debugging techniques

Previously, you examined three types of errors you may encounter while working in Python and explored strategies for debugging these errors. This reading further explores these concepts with additional strategies and examples for debugging Python code.

Types of errors

It's a normal part of developing code in Python to get error messages or find that the code you're running isn't working as you intended. The important thing is that you can figure out how to fix errors when they occur. Understanding the three main types of errors can help. These types include syntax errors, logic errors, and exceptions.

Syntax errors

A **syntax error** is an error that involves invalid usage of a programming language. Syntax errors occur when there is a mistake with the Python syntax itself. Common examples of syntax errors include forgetting a punctuation mark, such as a closing bracket for a list or a colon after a function header.

When you run code with syntax errors, the output will identify the location of the error with the line number and a portion of the affected code. It also describes the error. Syntax errors often begin with the label `"SyntaxError:"` . Then, this is followed by a description of the error. The description might simply be `"invalid syntax"` . Or if you forget a closing parentheses on a function, the description might be `"unexpected EOF while parsing"`. `"EOF"` stands for "end of file."

The following code contains a syntax error. Run it and examine its output:

```
1
2
message = "You are debugging a syntax error
print(message)
```

Reset

This outputs the message `"SyntaxError: EOL while scanning string literal"`. `"EOL"` stands for "end of line". The error message also indicates that the error happens on the first line. The error occurred because a quotation mark was missing at the end of the string on the first line. You can fix it by adding that quotation mark.

Note: You will sometimes encounter the error label `"IndentationError"` instead of `"SyntaxError"`. `"IndentationError"` is a subclass of `"SyntaxError"` that occurs when the indentation used with a line of code is not syntactically correct.

Logic errors

A **logic error** is an error that results when the logic used in code produces unintended results. Logic errors may not produce error messages. In other words, the code will not do what you expect it to do, but it is still valid to the interpreter.

For example, using the wrong logical operator, such as a greater than or equal to sign (**>=**) instead of greater than sign (**>**) can result in a logic error. Python will not evaluate a condition as you intended. However, the code is valid, so it will run without an error message.

The following example outputs a message related to whether or not a user has reached a maximum number of five login attempts. The condition in the **`if`** statement should be **`login_attempts < 5`**, but it is written as **`login_attempts >= 5`**. A value of **`5`** has been assigned to **`login_attempts`** so that you can explore what it outputs in that instance:

```
login_attempts = 5
if login_attempts >= 5:
    print("User has not reached maximum number of login attempts.")
else:
    print("User has reached maximum number of login attempts.")
```

Reset

The output displays the message **`"User has not reached maximum number of login attempts."`** However, this is not true since the maximum number of login attempts is five. This is a logic error.

Logic errors can also result when you assign the wrong value in a condition or when a mistake with indentation means that a line of code executes in a way that was not planned.

Exceptions

An **exception** is an error that involves code that cannot be executed even though it is syntactically correct. This happens for a variety of reasons.

One common cause of an exception is when the code includes a variable that hasn't been assigned or a function that hasn't been defined. In this case, your output will include **`"NameError"`** to indicate that this is a name error. After you run the following code, use the error message to determine which variable was not assigned:

```
1
2
3
```

```
username = "elarson"
month = "March"
total_logins = 75
failed_logins = 18
print("Login report for", username, "in", month)
print("Total logins:", total_logins)
print("Failed logins:", failed_logins)
print("Unusual logins:", unusual_logins)
```

Reset

The output indicates there is a **`"NameError"`** involving the **`unusual_logins`** variable. You can fix this by assigning this variable a value.

In addition to name errors, the following messages are output for other types of exceptions:

- **`"IndexError"`**: An index error occurs when you place an index in bracket notation that does not exist in the sequence being referenced. For example, in the list **`usernames = ["bmoreno", "tshah", "elarson"]`**, the indices are **`0`**, **`1`**, and **`2`**. If you referenced this list with the statement **`print(usernames[3])`**, this would result in an index error.
- **`"TypeError"`**: A type error results from using the wrong data type. For example, if you tried to perform a mathematical calculation by adding a string value to an integer, you would get a type error.
- **`"FileNotFound"`**: A file not found error occurs when you try to open a file that does not exist in the specified location.

Debugging strategies

Keep in mind that if you have multiple errors, the Python interpreter will output error messages one at a time, starting with the first error it encounters. After you fix that error and run the code again, the interpreter will output another message for the next syntax error or exception it encounters.

When dealing with syntax errors, the error messages you receive in the output will generally help you fix the error. However, with logic errors and exceptions, additional strategies may be needed.

Debuggers

In this course, you have been running code in a notebook environment. However, you may write Python code in an Integrated Development Environment (IDE). An **Integrated Development Environment (IDE)** is a software application for writing code that provides editing assistance and

error correction tools. Many IDEs offer error detection tools in the form of a debugger. A **debugger** is a software tool that helps to locate the source of an error and assess its causes.

In cases when you can't find the line of code that is causing the issue, debuggers help you narrow down the source of the error in your program. They do this by working with breakpoints. Breakpoints are markers placed on certain lines of executable code that indicate which sections of code should run when debugging.

Some debuggers also have a feature that allows you to check the values stored in variables as they change throughout your code. This is especially helpful for logic errors so that you can locate where variable values have unintentionally changed.

Use print statements

Another debugging strategy is to incorporate temporary print statements that are designed to identify the source of the error. You should strategically incorporate these print statements to print at various locations in the code. You can specify line numbers as well as descriptive text about the location.

For example, you may have code that is intended to add new users to an approved list and then display the approved list. The code should not add users that are already on the approved list. If you analyze the output of this code after you run it, you will realize that there is a logic error:

```
new_users = ["sgilmore", "bmoreno"]
approved_users = ["bmoreno", "tshah", "elarson"]
def add_users():
    for user in new_users:
        if user in approved_users:
            print(user,"already in list")
        approved_users.append(user)
add_users()
print(approved_users)
```

Reset

Even though you get the message **`"bmoreno already in list"`**, a second instance of **`"bmoreno"`** is added to the list. In the following code, print statements have been added to the code. When you run it, you can examine what prints:

```
new_users = ["sgilmore", "bmoreno"]
approved_users = ["bmoreno", "tshah", "elarson"]
def add_users():
    for user in new_users:
        print("line 5 - inside for loop")
        if user in approved_users:
            print("line 7 - inside if statement")
            print(user,"already in list")
        print("line 9 - before .append method")
        approved_users.append(user)
add_users()
print(approved_users)
```

Reset

The print statement **`"line 5 - inside for loop"`** outputs twice, indicating that Python has entered the **`for`** loop for each username in **`new_users`**. This is as expected. Additionally, the print statement **`"line 7 - inside if statement"`** only outputs once, and this is also as expected because only one of these usernames was already in **`approved_users`**.

However, the print statement **`"line 9 - before .append method"`** outputs twice. This means the code calls the **`.append()`** method for both usernames even though one is already in **`approved_users`**. This helps isolate the logic error to this area. This can help you realize that the line of code **`approved_users.append(user)`** should be the body of an **`else`** statement so that it only executes when **`user`** is not in **`approved_users`**.

Key takeaways

There are three main types of errors you'll encounter while coding in Python. Syntax errors involve invalid usage of the programming language. Logic errors occur when the logic produced in the code produces unintended results. Exceptions involve code that cannot be executed even though it is syntactically correct. You will receive error messages for syntax errors and exceptions that can help you fix these mistakes. Additionally, using debuggers and inserting print statements can help you identify logic errors and further debug exceptions.