W03 Learning Activity (2 of 2): Troubleshooting Functions

What should you do when your program isn't working? If your program is small, you could examine each line of the program. However, if your program is large or contains multiple functions, there are better ways to find and fix the problems, including writing and running test functions, writing print statements, and using a debugger.

Concepts

Here are the Python programming concepts and topics that you should learn during this lesson.

Types of Mistakes

Broadly speaking, there are two types of mistakes or errors that a programmer might make when writing a program: syntax errors and logic errors. A *syntax error* is a mistake made by a programmer that violates the rules of a programming language such as misspelling a keyword, forgetting to type a closing parenthesis, or forgetting to type a colon (:) at the end of an **`if`** statement. A syntax error will cause the computer to terminate a Python program and print an error message to the terminal window. A *logic error* is a mistake made by a programmer that causes the computer to produce the wrong results. Often, a logic error will not cause the computer to terminate a program or to print an error message. It will simply cause the computer to produce incorrect results.

Error Messages

Regardless of the type of error (syntax or logic), if the computer prints an error message while executing a program, the first thing a programmer should do is read and understand the error message. Example 1 shows a simple Python program that contains a syntax error. The message that the computer printed because of the syntax error is shown below example 1.

```
# Example 1
def main():
```

```
    print("Are you surprised, Clark?)
# Start this program by
# calling the main function.
if __name__ == "__main__":
    main()
```

```
> python surprise.py
  File "C:\Users\cse111\surprise.py", line 3
    print("Are you surprised, Clark?)
                                      ^
SyntaxError: EOL while scanning string literal
```

From the error message, we read that example 1 contains a syntax error and that the error is on line 3 of the program where there is something wrong with the string. By examining the error message and the program at line 4, we learn that the programmer forgot to type the closing double quote at the end of the string. By the way, *EOL* that appears in the error message is an acronym for "end of line." You might also see the acronyms *EOF* and *EOT* which mean "end of file" and "end of transmission."

If the computer prints an error message that you don't understand, you can search the internet for its meaning. Simply copy and paste the error message into the search bar of your browser. Here are two of the search results from Google for the error message "SyntaxError: EOL while scanning string literal."

About 27,500 results (0.49 seconds) SyntaxError- EOL while scanning string literal SyntaxError: EOL while scanning string literal "EOL" stands for "end of line". An EOL error means that Python hit the end of a line while going through a string. Syntax Error: EOL while scanning string literal - AskPython EOL stands for "End of Line". The error means that the Python Interpreter reached the end of the line when it tried to scan the string literal. The string literals (constants) must be enclosed in single and double quotation marks.

Print Statements

If the computer doesn't print an error message, but your program is producing incorrect results, you could add print statements to your program in strategic locations to help you find the mistakes. These print statements should print the value of the variables in your program so that you can examine the values to ensure they are correct. Example 2 contains a program with a complex calculation for computing the remaining balance of a loan. Notice the print statements inside the functions indicated by the "Show variable values for debugging" comment. The programmer wrote these print statements at important points in the program, specifically near the beginning of each function and in the middle of the calculations.

```
# Example 2
def main():
  print("This program computes and prints the remaining")
  print("balance for a loan with a fixed annual percentage")
  print("rate and a fixed number of payments per year.")
  print()
  print("Please enter the following five values.")
  principal = float(input("Principal amount: "))
  annual_rate = float(input("Annual percentage rate: "))
  years = int(input("Number of years in the life of the loan: "))
  payments_per_year = int(input("Number of payments per year: "))
  number_paid = int(input("Number of payments already paid: "))
  balance = compute_balance(principal, annual_rate, years,
          payments_per_year, number_paid)
  print()
  print(f"Balance remaining: {balance}")
def compute_balance(princ, ar, years, ppy, ptd):
  """Compute and return the balance remaining for a loan."""
  payment = compute_payment(princ, ar, years, ppy)
  #Show variable values for debugging
  print()
  print(f"compute_balance({princ}, {ar}, {years}, {ppy}, {ptd})")
  rate = ar / ppy
  power = (1 + rate) ** ptd
  #Show variable values for debugging
  print(f"    payment: {payment}  rate: {rate}  power: {power}")
  balance = princ * power - payment * (power - 1) / rate
  #Show variable values for debugging
  print(f"    balance: {balance:.2f}")
  return round(balance, 2)
def compute_payment(princ, ar, years, ppy):
  """Compute and return the payment per period for a loan."""
  #Show variable values for debugging
  print()
  print(f"compute_payment({princ}, {ar}, {years}, {ppy})")
  rate = ar / ppy
  n = years * ppy
  #Show variable values for debugging
  print(f"    rate: {rate}  n: {n}")
  payment = princ * rate / (1 - (1 + rate) ** -n)
  #Show variable values for debugging
  print(f"    payment: {payment:.2f}")
  return round(payment, 2)
# Start this program by
# calling the main function.
if __name__ == "__main__":
    main()
```

```
> python balance.py
This program computes and prints the remaining
balance for a loan with a fixed annual percentage
rate and a fixed number of payments per year.
Please enter the following five values.
Principal amount: 80000
Annual percentage rate: 0.06
Number of years in the life of the loan: 15
Number of payments per year: 12
Number of payments already paid: 45
compute_payment(80000.0, 0.06, 15, 12)
    rate: 0.005  n: 180
    payment: 675.09
compute_balance(80000.0, 0.06, 15, 12, 45)
    payment: 675.09  rate: 0.005  power: 1.2516208207696773
    balance: 66156.33
Balance remaining: 66156.33
```

Although print statements are simple to understand and add to a program and often helpful, in many situations they are not the most effective way to find logic errors.

Test Functions

Many programmers underestimate the effectiveness of writing and running test functions to find logic errors in a program. Many programmers will write a complete program with multiple functions and never pause to test any part of it. Instead, they will write the entire program and then do something similar to the following steps to test it.

1. Run the program and enter input like a user would.
2. Examine the program's output and discover that the output is incorrect.
3. Review all the program's code, make some small changes, and add print statements.
4. Repeat steps 1–3 again and again and again, spending lots of time trying to find and fix the errors.

This method for finding and fixing mistakes is time consuming because the programmer is trying to test the whole program at once. Instead, the programmer should test each individual function by writing and running test functions as explained in the previous lesson.

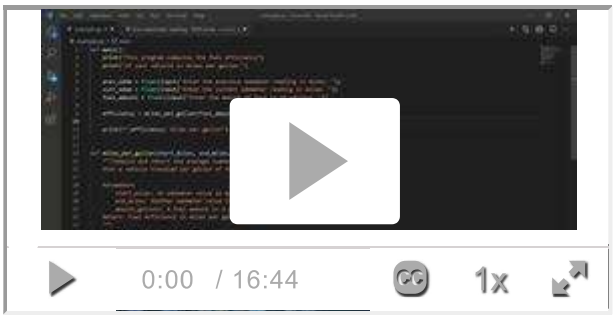
Using a Debugger

A *debugger* is a software development tool that allows a programmer to watch the computer execute the statements in a program. While using a debugger, a

programmer can examine the values stored in a program's variables as the computer executes each line of the program. A debugger is a very effective tool for finding mistakes in a program. Nearly all programming languages and environments include a debugger. Professional software developers use a debugger nearly every single work day. Most companies will not use a programming language unless that language includes a debugger.

The Python programming language includes a debugger that you can use while writing a program in VS Code. It is much easier to learn how to use a debugger by watching someone use it than by reading about a debugger. Watch the following video that shows a BYU-Idaho faculty member using a debugger in VS Code to find mistakes in a program.

- How to Use the Python Debugger in VS Code (17 minutes)



A debugger is also a great learning tool. Using a debugger to step through the statements of your program one at a time and examining the values of the variables after each step will show you exactly how Python works. Try it! If you don't completely understand **`if-elif-else`** statements, **`while`** loops, **`for`** loops, passing parameters, or returning a value from a function, then put one or more breakpoints in a program that contains those elements, start the program in the debugger, and step through it one line at a time. After the computer executes each statement, examine the values of the variables and predict in your mind how the next statement will change the values of the variables.

Summary

During this lesson, you are learning the difference between a syntax error and a logic error. A syntax error is a mistake made by a programmer that violates the rules of a programming language and prevents a program from running. Usually, you must fix all syntax errors before your program will run. After you fix all syntax errors, your program will run but may contain logic errors. A logic error is a mistake made by a programmer that causes the computer to produce the wrong results. Some tools that you can use to find and fix the logic errors in your programs are test functions, print statements, and a debugger.

W03 Checkpoint: Troubleshooting Functions

Purpose

Practice using the Python debugger in VS Code.

Assignment

Do the following:

1. Download and save the **`example-2.py`** file.
2. Open **`example.py`** in VS Code.
3. Follow along with this video about the Python Debugger (17 minutes) from the preparation content and use the debugger in VS Code to step through the **`example.py`** program.

Ponder

As you used the debugger to step through the **`example.py`** program, did you learn anything new about Python and how functions work? Do you think the debugger could help you find mistakes in your code?

Submission

When complete, return to Canvas and report your progress in the **W03 Quiz: Testing and Fixing Functions**.

Useful Links:

- Return to: Week Overview | Course Home