Debugging Operations

by Sophia

WHAT'S COVERED

In this lesson, you will learn about some of the common errors with operations. Specifically, this lesson covers:

1. Common Variable Errors
2. Common Operations Errors

1. Common Variable Errors

The debugging process is similar to what you would do to test code but rather than just identifying errors and issues, you would be working to fix those errors and remove problems also known as "bugs." At this point, the syntax error you are most likely to make is to use an illegal variable name.

EXAMPLE
For example:

- Using the words “class” and “yield” as variable names. These are Python reserved keywords.
- Using “odd~job” and “US$” as variable names. These contain illegal characters like the ~ and $ characters.

Remember the rules for variable names include:

- Must start with a letter or an underscore character.
- Cannot start with a number.
- Can only contain letters, numbers and the underscore character.
- Variable names are case sensitive.

So, if you put a space in a variable name, Python thinks it is two operands without an operator.
EXAMPLE

```
bad variable = 'test'
```

As expected, the spacing caused a syntax error.

```
  File "/home/main.py", line 1
    bad variable = 'test'
        ^^^^^^^^
SyntaxError: invalid syntax
```

Reserved Words

Python has 35 reserved keywords that are used to recognize the structure of a program. Since they are reserved, they cannot be used as variable names. These are the reserved words:

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

Let's see what happens if we try to use one of the reserved words as a variable name.

EXAMPLE

```
class = 'test'
```

We get an error on the console when trying to compile it.

```
  File "/home/main.py", line 1
    class = 'test'
          ^
SyntaxError: invalid syntax
```

The runtime error that you will most likely make is to use a variable before assigning it a value. You can run into this error if you have a typo in the variable name.

EXAMPLE

```
principal = 327.68
rate = 5
```

```
interest = principle * rate
print(interest)
```

This produces an error on output.

```
  Trackback (most recent call last):
    File "/home/main.py", line 3, in <module>
     interest = principle * rate
  NameError: name 'principle' is not defined. Did you mean: 'principal'?
```

As you'll see here, we intended to have `principal` (PAL) instead of `principle` (PLE) when we were doing the calculation. We can also run into an error if we simply did not assign a value to a variable before trying to use it. Those types of errors will generate a NameError as seen above.

↷ EXAMPLE

```
principal = 327.68
interest = principal * rate
print(interest)
```

In this case, `rate` has not been assigned a value. However, we are trying to use it to calculate `interest`.

```
  Trackback (most recent call last):
    File "/home/main.py", line 2, in <module>
     interest = principal * rate
  NameError: name 'rate' is not defined. Did you mean: 'range'?
```

Don't forget that variable names are case sensitive, so `Principal` and `principal` would not be the same.

↷ EXAMPLE

```
Principal = 327.68
rate = 2
interest = principal * rate
print(interest)
```

Now `principle` is throwing an error.

```
  Trackback (most recent call last):
    File "/home/main.py", line 3, in <module>
     interest = principle * rate
  NameError: name 'principal' is not defined. Did you mean: 'Principal'?
```

Note that if you have the variable defined in both ways in a program, the issue becomes a logical error that you would have to catch, as you won't get an error while compiling.

↷ EXAMPLE

```
Rate = 100
principal = 327.68
rate = 0
interest = principal * rate
print(interest)
```

In this case, we have `rate` and `Rate`. If we intended to use the `Rate` of 100 but instead used the `rate` of 0, this would run but return the `interest` value of 0 instead of 32768.

```
0.0
```

---

2. Common Operations Errors

Many of the common errors with operations occur when you are mixing various data types. For example, you may have some numbers that you wanted to add, but you may have accidentally used quotes around them.

EXAMPLE

```
var1 = "3"
var2 = "4"
var3 = var1 + var2
print(var3)
```

What would you assume the results to be?

```
34
```

Oops, the reason for that is the double quotes, which makes them strings that are concatenated instead of added. Instead, we want to ensure that if we intend to add the values, we don't use quotes around them.

```
var1 = 3
var2 = 4
var3 = var1 + var2
print(var3)
```

Now the addition looks correct.

```
7
```

Another situation may be if we have a mixed scenario of using an operator on incompatible data types. This can create an issue because Python doesn't understand if we intended to concatenate the values or add them together since in this case, one is a string and the other is an int. This will cause a TypeError.

EXAMPLE

```
var1 = 3
var2 = "4"
var3 = var1 + var2
print(var3)
```

Here is the expected error on output.

```
  Trackback (most recent call last):
    File "/home/main.py", line 3, in <module>
     var3 = var1 + var2
  TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Another common error is a logical error with the order of operations. For example, if we had three grades and we wanted to calculate the average, we would want to add the three grades and divide by 3. We may do something like the following:

EXAMPLE

```
grade1 = 86
grade2 = 100
grade3 = 72
average = grade1 + grade2 + grade3 / 3
print(average
```

The output throws an error.

```
  File "/home/main.py", line 5
    print(average
          ^
SyntaxError: '(' was never closed
```

That's definitely an error due to the order of operations. What happened in this case is that `grade3 / 3` occurred first. Then `grade1` was added to `grade2` and the result of the `grade3 / 3`. We need to use parentheses to ensure that the grades are added together first, before dividing them all by 3.

```
grade1 = 86
grade2 = 100
grade3 = 72
average = (grade1 + grade2 + grade3) / 3
print(average)
```

Now, this looks better.

```
86.0
```

Remember for mathematical operators, the acronym PEMDAS is a useful way to remember the rules.

PEMDAS Definitions	
P (Parentheses)	Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first: 2 * (3-1) is 4, and (1+1)**(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute * 100) / 60, even if it doesn't change the result.
E (Exponent)	Exponentiation has the next highest precedence, so: 2**1+1 is 3, not 4, and 3*1**3 is 3, not 27.
MD (Multiplication and Division)	Multiplication and Division have the same precedence, which is higher than Addition and Subtraction, which also have the same precedence, so: 2*3-1 is 5, not 4, and 6+4/2 is 8, not 5.
AS (Addition and Subtraction)	Operators with the same precedence are evaluated from left to right, so: 5-3-1 is 1, not 3, because the 5-3 happens first and then 1 is subtracted from 2.

Remember with these logical errors Python has no way of knowing what you actually meant to write. You won't get any error messages—you simply get the wrong answer. It is always good to validate and double-check any calculations outside of Python.

SUMMARY

In this lesson, we learned about **common variable errors**, including spaces in variable names and using Python reserved keywords as variables. We also learned some **common operation errors**, including mixing data types and having an unexpected order of operations.

Best of luck in your learning!