HyperionDev

Defensive Programming

Additional Reading

Visit our website

Defensive programming

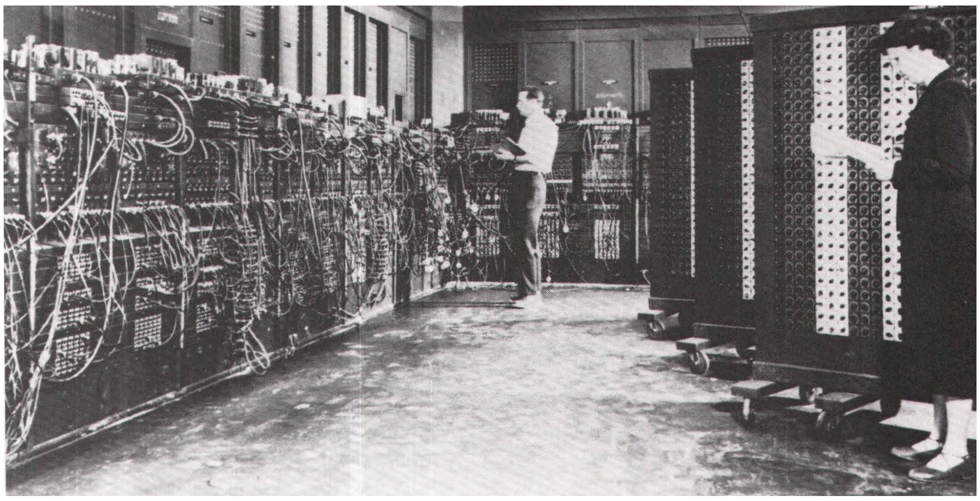
Defensive programming is an approach to writing code where the programmer tries to anticipate problems that could affect the program and then takes steps to defend the program against these problems. MANY problems could cause a program to run unexpectedly!

Debugging

Debugging is something that you, as a software developer/programmer, will come to eat, sleep, and breathe! Debugging is when a programmer looks through their code to try to identify a problem that is causing some error.

The word "debugging" comes from the first computers, back when a computer was roughly the size of an entire room. Bugs (living insects) would get into the computer and cause the device to function incorrectly. The programmers would need to go in and remove the insects, hence the name – debugging.

One such computer is the ENIAC (Electronic Numerical Integrator and Computer). It was the first true electronic computer and was located at the University of Pennsylvania. Computers, as we know them today, are significantly smaller and also much more powerful than older computers like the ENIAC (despite their enormous size). It goes to show how, in just a few years, we can expand our computational power immensely. In terms of the rate of growth on computational power, Gordon Moore, co-founder of Fairchild Semiconductor and Intel (and former CEO of the latter), predicted in the 1960s that the number of transistors in an integrated circuit would double about every two years, a prediction which has been borne out as correct over time. In the world of tech, it is necessary to actively work to keep our knowledge and expertise up to date! If you're interested, you can do some further research about **the ENIAC on the University of Pennsylvania's website.**



The ENIAC (U.S. Army, 1947)

Software developers live by the motto: “try, try, try again!” Testing and debugging your code repeatedly is essential for developing effective and efficient code, and achieving mastery as a developer/programmer.

Error handling

Everyone makes mistakes. Likely you've made some already, and that’s not a bad thing! As you’ve probably already realised, however, just making the mistakes doesn’t facilitate any learning – it's important to be able to DEBUG your code. You debug your code by identifying various types of errors and correcting them. What type of errors might you encounter? Let’s take a look.

Types of errors

There are three different categories of errors that can occur:

- **Syntax errors** are due to incorrect syntax used in the program (e.g. wrong spelling, incorrect indentation, missing parentheses, etc.). The compiler can detect such errors. If syntax errors exist when the program is compiled, the compilation of the program fails and the program is terminated. Syntax errors are also known as compilation errors. They are the most common type of error.

- **Runtime errors** occur during the execution of your program. The program will compile as normal, but the error occurs while the program is running. An error message will likely also pop up when trying to run the buggy program. Examples of what might cause a runtime error include dividing by zero, and referencing an out-of-range list item (if you’re unfamiliar with lists and impatient to learn more, you can **read more on it here**).

- **Logical errors** occur when your program’s syntax is correct, and it runs and compiles, but the output is not what you are expecting. This means that the logic you applied to the problem contains an error. Logical errors can be the most difficult errors to spot and resolve, because you need to identify where you have gone wrong in your thought process. Building **metacognitive skills** can help you gain proficiency in detecting logical errors. A useful HyperionDev resource for building metacognitive skills through HyperionDev’s **Learning Accelerator.**

Now that you know the three types of errors you might encounter, let’s consider each in more detail.

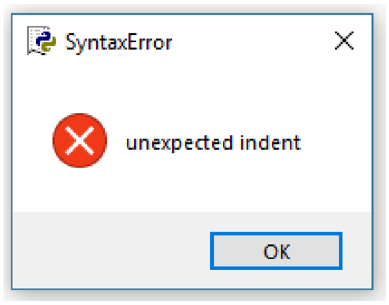
Syntax errors

Syntax errors are comparable to making spelling or grammar mistakes when writing in English (or any other language), except that a computer requires perfect syntax to run correctly.

Common Python syntax errors include:

- Leaving out a keyword or putting it in the wrong place.
- Misspelling a keyword (such as function or variable name) .
- Leaving out an important symbol (such as a colon, comma, or parentheses).
- Incorrect indentation.
- Leaving an empty block (such as **if** statement with no indented statements).

Below is a typical example of a compilation error message. Compilation errors are the most common type of error in Python. They can get a little complicated to fix when dealing with loops and if statements.


SyntaxError

unexpected indent

When a syntax error occurs, the line in which the error is found will often be highlighted, although exactly how depends on your development environment (IDE). The cursor may also automatically be placed at the point the error occurred to make the error easy to identify. However, watch out – this can be misleading! For example, if you leave out a symbol, such as a closing bracket or quotation mark, the error message could refer to a place later on in the code, which is not the actual source of the problem.

Nonetheless, when you get a syntax error, go to the line indicated by the error message and correct the error if possible. If it is not in that line, work backwards through the code until you identify the source of the problem and then correct it. Then try to compile your code again. Debugging is an essential part of being a programmer, and although sometimes frustrating, it can be quite fun to problem-solve the causes of your errors.

Runtime errors

Using your knowledge of strings, take a look at the code below and see if you can identify the runtime error:

```
greeting = "Hello!"
print(greeting[6])
```

This would be the error you get, but why?

```
Exception has occurred: IndexError
string index out of range
```

Look carefully at the description of the error. It must have something to do with the string index. String characters are indexed from zero, and so the final index for “!” would be five, not six. That means that nothing exists for `greeting[6]`, so we get a runtime error. It's important to read error messages carefully and think in a deductive way to solve them. It may, at times, be useful to copy and paste the error message into Google to figure out how to fix the problem.

Logical errors

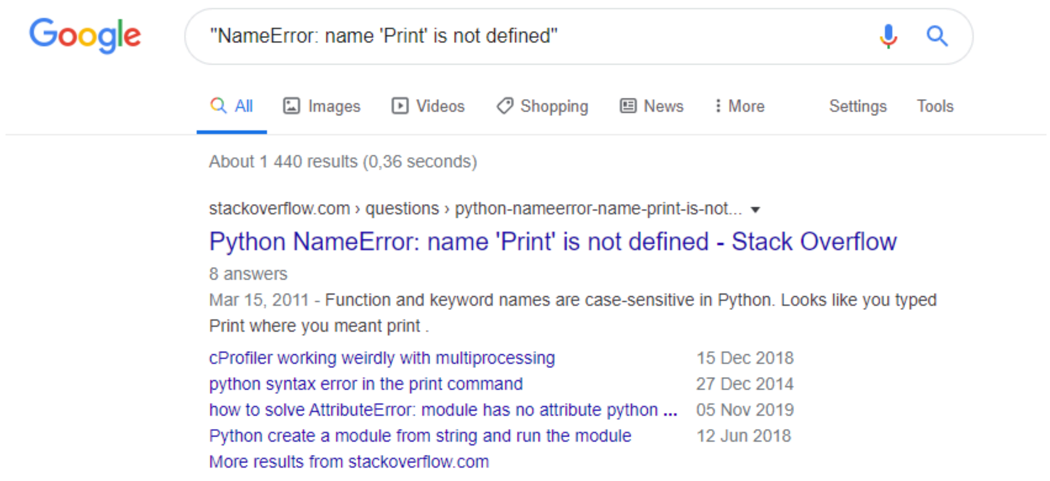
You are likely to encounter logical errors, especially when dealing with loops and control statements. Even after years of programming, logical errors still occur, and more so if you dive into coding solutions to problems without planning the high-level approach using pseudo code. This pseudo code planning, so often skipped by students eager to start coding, is a vital component of logical planning for problem-solving. It takes time to sit down and design an algorithm, but it makes later debugging much easier.

Tips for debugging

Debugging becomes easier with practice and experience. You have probably seen plenty of errors in the Python shell similar to the following error:

```
Print ("Hello World!")
NameError: name 'Print' is not defined
```

Many of the errors are quite easy to identify and correct. However, often you may find yourself thinking, 'What does that mean?'. One of the easiest, and often most effective, ways of dealing with error messages that you don't understand is to simply copy and paste the error into Google.



About 1 440 results (0,36 seconds)

stackoverflow.com › questions › python-nameerror-name-print-is-not...

Python NameError: name 'Print' is not defined - Stack Overflow

8 answers

Mar 15, 2011 - Function and keyword names are case-sensitive in Python. Looks like you typed Print where you meant print .

cProfiler working weirdly with multiprocessing	15 Dec 2018
python syntax error in the print command	27 Dec 2014
how to solve AttributeError: module has no attribute python ...	05 Nov 2019
Python create a module from string and run the module	12 Jun 2018

More results from stackoverflow.com

Many forums and websites are available that can help you to identify and correct errors easily. **Stack Overflow** is an excellent resource that software developers around the world use to get help.

LLMs, like OpenAI's ChatGPT or Google's Bard, can also be really helpful in debugging tricky errors.

In addition, your IDE may have some other built-in features to help you identify and correct subtle errors. Find out if it has a debugger and how to turn it on if it isn't on by default (**here are the instructions for VS Code)**.

Many debuggers will allow you to do things like "Go", "Step", etc. This part of debugger functionality is especially important if you use breakpoints in your code. A breakpoint is a point in your code where you tell it to pause the execution of the code so that you can check what is going on. For example, if you don't understand why an **if** statement isn't doing what you think it should, you could create a breakpoint at the **if** statement. You can follow this **video** to learn how to set breakpoints in VS Code.

Introduction to debugging

Debugging is essential to any Software Developer. In this task, you will learn about some of the most commonly encountered errors. You will also learn to use your IDE to debug your code more effectively.

Some of the most common types of problems to look out for are listed below:

- **User errors:** The people who use your application will act unexpectedly. They will do things like entering string values where you expect numbers. Or they may enter numbers that cause calculations in your code to crash (e.g., the prevalent `ZeroDivisionError`). They may also press buttons at the wrong time or more times than you expect. As a developer, anticipate these problems and write code that can handle these situations. For example, one approach is to check all user input.

- **Errors caused by the environment:** Write code that will handle errors in the development and production environment. For example, in the production environment, your program may get data from a database that is on a different server. Code should be written in such a way that it deals with the fact that some servers may be down, or the load of people accessing the program is higher than expected, etc.

- **Logical errors with the code:** Besides external problems that could affect a program, a program may also be affected by errors within the code. All code should be thoroughly tested and debugged.

Using if statements for validation

Many of the programming constructs that you have already learned can be used to write defensive code. For example, if you assume that any user of your system will be younger than 150 years old, you could use a simple `if` statement to check that someone entered a valid age.

Handling exceptions

Errors that occur at runtime (after passing the syntax test) are called **exceptions** or logical errors.

For instance, they occur when we try to open a file (for reading) that does not exist (`FileNotFoundError`), try to divide a number by zero (`ZeroDivisionError`), or try to import a module that does not exist (`ModuleNotFoundError`).

Whenever these types of runtime errors occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

The following examples illustrate exceptions occurring when trying to divide by zero, using a variable that has not been declared, and trying to concatenate a string with an integer:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name “spam” is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

The above exceptions are all built into the Python language and are just some of the many built-in exceptions. To find out more, you can take a look at the **documentation on built-in exceptions for Python 3**.

A part of defensive coding is to anticipate where these exceptions might occur, make provisions for such exceptions, and trigger specific actions for when these errors occur.

try–except block

When handling exceptions, two main blocks of code are used to ensure that specific actions are taken when the exception occurs, namely, the `try` block and the `except` block. The `try` block is the part of the code that we are trying to execute if no exceptions occur. The `except` block, therefore, is the code that we specify needs to be executed if an exception does occur. Let's look at the following example:

```
while True:
    try:
        x = int(input("Please enter an integer: "))
        break
    except ValueError:
        print("Oops! That was not a valid entry. Try again...")
```

In the above code example, we used a `while` loop to take user input. The `try` block gets executed first. If the input is a valid number, the integer value will be assigned to the `x` variable, and the loop will break.

However, if the user input is a string that is not a valid number, a `ValueError` is raised. The `except` block will then be executed, thereby printing the message to the terminal. It is worth noting that any code can be executed in the `except` block, and it does not have to be a print statement. After the message is printed to the terminal, the loop will start from the `try` block again until the user enters a valid input, and the loop can break.

We can also perform the above task in the following way:

```
while True:
    try:
        x = int(input("Please enter an integer: "))
        break
    except Exception:
        print("Oops! That was not a valid entry. Try again...")
```

This method should be used cautiously as a fundamental programming error can slip through in your programs this way. The above code will run the `except` block no matter what type of exception occurs from the execution of the `try` block.

try – except – finally

There are occasions where a block of code needs to be executed whether or not an exception has occurred. In this scenario, we can use the `finally` block. The `finally` block is usually used to terminate anything after it has been utilised, such as database connections or open resources. In the example below, the file object needs to be closed whether an exception has occurred or not; therefore, the file object is closed in the `finally` block.

```
file = None
try:
    file = open('input.txt', 'r')
    # Do stuff with the file here

except FileNotFoundError:
    print("The file that you are trying to open does not exist")

finally:
    if file is not None:
       file.close()
```

In the code example above, defensive programming is used. The coder has anticipated that the file may not be found. Therefore, the code handles the exception.

Code hack

Beware! Do not be tempted to overuse `try-except` blocks. If you can anticipate and fix potential problems without using `try-except` blocks, do so. For example, don't use `try-except` blocks to avoid writing code that validates user input.

Understanding exception objects

When an exception occurs, an exception object is created. The exception object contains information about the error. We can get more information about this error by printing the error object.

Let's use our previous example, but this time we are going to assign the exception object to a variable by using the `except-as` syntax and then print the error:

```
file = None
try:
    file = open('input.txt', 'r')
    # Do stuff with the file here

except FileNotFoundError as error:
    print("The file that you are trying to open does not exist")
    print(error)
finally:
    if file is not None:
       file.close()
```

When this code executes and the file does not exist, then the following output is generated that gives us some more information about the error that occurred:

```
The file that you are trying to open does not exist
[Errno 2] No such file or directory: 'input.txt'
```

Raising exceptions

There will be occasions when you want your program to raise a custom exception whenever a certain condition is met. In Python, we can do this by using the `raise` keyword and adding a custom message to the exception.

In the example below, we are asking the user to input a value greater than `10`. If the user enters a number that does not meet that condition, an exception is raised with a custom error message.

```
num = int(input("Please enter a value greater than 10: "))
if num <= 10:
    raise Exception(f'num value ({num}) is less than or equal to 10')
```

Reference list

U.S. Army. (1947). *Glen Beck and Betty Snyder program the ENIAC in building 328 at the Ballistic Research Laboratory [Photograph]*. Wikimedia Commons. **https://commons.wikimedia.org/wiki/File:Glen_Beck_and_Betty_Snyder_program_the_ENIAC_in_building_328_at_the_Ballistic_Research_Laboratory.jpg**