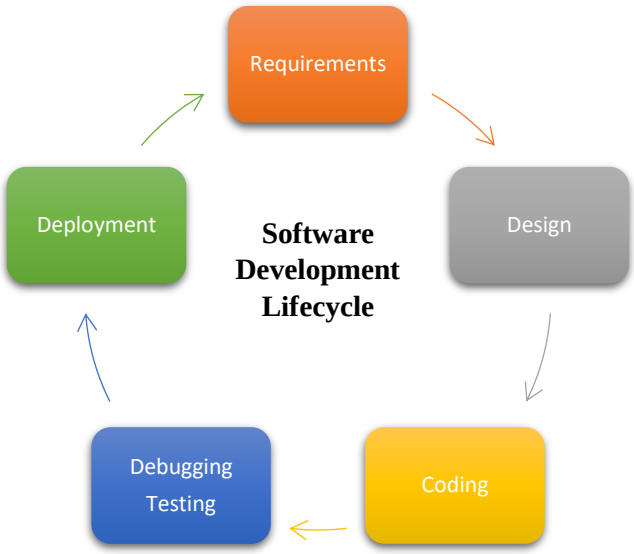
CMPSC-132: Programming and Computation II

Department of Computer Science & Engineering
The Pennsylvania State University

1. Debugging

Debugging involves locating and correcting code errors in a computer program. Debugging is part of the software testing process and is an integral part of the entire software development lifecycle. The debugging process starts as soon as code is written and continues in successive stages as code is combined with other units of programming to form a software product. The best place to debug the code is away from your computer, working on your understanding of what should be happening.



Once testing has uncovered problems, the next step is to fix them. Many beginners do this by making more-or-less random changes to their code until it seems to produce the right answer, but that is very inefficient and often only corrects the one case being tested. Debugging your code can be more time consuming than writing the code, but it is the more efficient way to test it and make sure it works correctly. Although each debugging experience is unique, certain general principles can be applied in debugging:

- *Know what the code is supposed to do*: in order to diagnose and fix problems, we need to be able to tell correct output from incorrect. If we can write a test case for the failing case, then we are ready to start debugging. If we can't, then we need to figure out how we are going to know when we have fixed things.
- *Make it fail every time*: we can only debug something when it fails, so we need to find a test case that makes it fail every time. Make sure that you are actually exercising the problem that you think you are (calling code with the right data or configuration parameters).

- *Change one thing at a time, for a reason*: replacing random chunks of code is unlikely to do much good. Therefore you should change one thing at a time, and however small the change is, you should re-run your tests immediately. Be aware that fixes made to code can introduce bugs as well.
- *Keep track of what you have done*: records are particularly useful when the time comes to ask for help. People are more likely to listen when you can explain clearly what you did, and you are better able to give them the information they need to be useful.
- *Version control revisited*: version control is often used to reset software to a known state during debugging, and to explore recent changes to code that might be responsible for bugs.
- *Ask for help*: explaining the problem aloud is often useful, since hearing what we are thinking helps us spot inconsistencies and hidden assumptions. However, there is a difference between asking "What's wrong with my code?" and "I wanted to do X, so I wrote <some-code>. I expected to see Y when I ran this code, but instead I saw Z. What am I missing?"

2. Errors

Programmers frequently encounter errors, either they are just beginning or they have been programming for years. Encountering errors and exceptions can be very frustrating at times, and can make coding feel like a hopeless effort. However, understanding what the different types of errors are and when you are likely to encounter them can help a lot. Once you know why you get certain types of errors, they become much easier to fix.

2.1 Syntax Errors

Syntax errors are usually easy to fix once you figure out what they are. Unfortunately, the error messages are often not helpful. The most common messages are *SyntaxError: invalid syntax* and *SyntaxError: invalid token*, neither of which is very informative. On the other hand, the message does tell you where in the program the problem occurred. When you forget a colon at the end of a line, accidentally add one space when indenting under an *if* statement, or forget a parenthesis, you will encounter a syntax error, which means that Python could not figure out how to read your program:

```
def hello_function()
    msg = "Hello, world!"
    print(msg)

hello_function()
```

```
  File "<stdin>", line 1
    def hello_function()
                       ^
SyntaxError: invalid syntax
```

In this case, Python informs that there is a *SyntaxError* on line 1, and even puts a little arrow in the place where there is an issue. In this case the problem is that the *hello_function* definition is missing a colon at the end.

2.1.1 Tabs vs spaces

Indentation errors are a common type of syntax error, and they can be insidious, especially if you are mixing spaces and tabs. Because they are both whitespace, it is difficult to visually tell the difference. By default, one tab is equivalent to eight spaces, so the only way to mix tabs and spaces is to use them like

that. In general, it is better to just never use tabs and always use spaces, because it can make things very confusing.

```
def hello_function():
    msg = "Hello, world!"
    print(msg)
     return(msg*2)
hello_function()
```

```
  File "<stdin>", line 4
    return(msg)
    ^
IndentationError: unexpected indent
```

Both *SyntaxError* and *IndentationError* indicate a problem with the syntax of your program, but an *IndentationError* is more specific: it always means that there is a problem with how your code is indented. In the above example, lines in the function definition do not all have the same indentation.

2.2 Runtime Errors

Once a program is syntactically correct, Python can import it and at least start running it. One of the simplest runtime errors occur when the program does absolutely nothing. This usually occurs when the code consists of functions and classes but does not actually invoke anything to start execution. Always check first that you ate invoking a function to start execution.

2.2.1 Program Hanging

If a program stops and seems to be doing nothing, we say it is hanging. Often that means that it is caught in an infinite loop or an infinite recursion. There are several hints that can help you to locate the loop or recursion that is causing the problem:

- If there is a particular loop that you suspect is the problem, add a *print* statement immediately before the loop that says entering the loop and another immediately after that says exiting the loop. If you get the first message and not the second after you run the program, you have got an infinite loop.
- An infinite recursion will usually cause the program to run for a while and then produce a *RuntimeError: Maximum recursion depth exceeded error*.

2.2.1.1 Infinite Loop

If you have an infinite loop and you think you know what loop is causing the problem, add a *print* statement at the end of the loop that prints the values of the variables in the condition and the value of the condition.

```
while x > 0 and y < 0:
    # Statement to modify x
    # Statement to modify y
    # ...
    print("x: ", x)
    print("y: ", y)
    print("Condition: ", (x > 0 and y < 0))
```

When we run the program from above, you will see three lines of output for each time through the loop. The last time through the loop, the condition should be *False*. If the loop keeps going, you will be able to see the values of *x* and *y*, and you might figure out why they are not being updated correctly. When using

this technique, it is important to understand what the algorithm should be doing, otherwise printing or inspecting the values for *x* and *y* is of little use.

2.2.1.2 Infinite Recursion

If a function or method is causing an infinite recursion, it will often cause the program to run for a while and then produce a *Maximum recursion depth exceeded error*. To fix this error, you should start by checking a base case (condition that will cause the function or method to return without making a recursive invocation). If you cannot establish a base case, it may be necessary to rethink the algorithm until you can identify a base case. If there is a base case but the program does not seem to be reaching it, add a print statement at the beginning of the function or method that prints the parameters. Now when you run the program, you will see a few lines of output every time the function or method is invoked, and you will see the parameters. If the parameters are not moving toward the base case, you will get some ideas about why not.

```
def fact(n):
    print(n)
    if n == 0:
        return 1
    else:
        return n*fact(n-1)
```

Base case

```
fact(3) outputs:
3
2
1
0
```

Flow of execution

If you are not sure how the flow of execution is moving through your program, add *print* statements to the beginning of each function with a message like 'entering function X'. Now when you run the program, it will print a trace of each function as it is invoked.

2.2.2 Variable Name Errors

This type of error is also very common and occurs when you try to use a variable that does not exist. Variable name errors come with some of the most informative error messages, which are usually of the form *NameError: name 'variable_name' is not defined*. This error typically occurs for the following reasons: (i) you meant to use a string, but forgot to put quotes around it, (ii) you forgot to create the variable before using it, (iii) you made a typo when you were writing your code.

```
print(hello)
```

```
NameError: name 'hello' is not defined
```

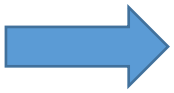
```
for number in range(5):
    count = count + number

print("The count is:", count)
```

```
NameError: name 'count' is not defined
```

2.2.3 Index Errors

Index errors have to do with containers (lists, strings) and the items within them. If you try to access an item in a list or a string that does not exist, then you will get an error.

```
fruits = ['apples', 'bananas', 'pears']
print("Fruit 1 is", fruits[0])
print("Fruit 2 is", fruits[1])
print("Fruit 3 is", fruits[2])
print("Fruit 4 is", fruits[3])
```



```
Fruit 1 is apples
Fruit 2 is bananas
Fruit 3 is pears

IndexError: list index out of range
```

More runtime errors

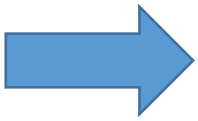
Additional common runtime errors are:

- *TypeError*: You are trying to use a value improperly (indexing a string, list, or tuple with something other than an integer), passing the wrong number of arguments to a function or method.
- *KeyError*: You are trying to access an element of a dictionary using a key value that the dictionary does not contain.
- *AttributeError*: You are trying to access an attribute or method that does not exist.

2.3 Semantic Errors

If there is a semantic error in your program, it will run successfully in the sense that the computer will not generate any error messages. However, your program will not do the right thing. It will do something else. Specifically, it will do what you told it to do. In some ways, semantic errors are the hardest to debug, because the compiler and the runtime system provide no information about what is wrong. Only you know what the program is supposed to do, and only you know that it is not doing it. The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

```
c = 35
f = (2/5) * c + 32

print(f)
```



35 °C is equal to 95 F, however,
the program outputs 46

3. Exceptions

Whenever a runtime error occurs, it creates an *exception* object. The program stops running at this point and Python prints out the traceback, which ends with a message describing the exception that occurred:

```
print(55/0)
      Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
      ZeroDivisionError: division by zero
```

Dividing by zero creates an exception. In this case, the error message on the last line has two parts: the type of error before the colon, and specifics about the error after the colon. Sometimes we want to execute an operation that might cause an exception, but we do not want the program to stop. We can handle the exception using the *try* statement to wrap a region of code.

3.1 The *try* and *except* statements

To handle possible exceptions, we use a *try-except* block:

```
try:
  x = int(input("Please enter a number: "))
  print(55/x)
except ZeroDivisionError:
  print("x cannot be zero!")
```

Python will try to process all the statements inside the *try* block. If a *ZeroDivisionError* occurs at any point as it is executing them, the flow of control will immediately pass to the *except* block, and any remaining statements in the try block will be skipped. In this example, we know that the error will occurs when the user gives the number zero as an input. You may also want to react in different ways to different kinds of errors. You should always try to pick specific rather than general error types for your *except* clauses. It is possible for one *except* clause to handle more than one kind of error, you can provide a tuple of exception types instead of a single type:

```
try:
  x = int(input("Please enter a number: "))
  print(55/x)
except(ValueError, ZeroDivisionError):
  print("Something went wrong!")
```

A *try-except* block can also have multiple *except* clauses. If an exception occurs, Python will check each *except* clause from the top down to see if the exception type matches. If none of the *except* clauses match, the exception will be considered unhandled, and your program will crash.

```
try:
  x = int(input("Please enter a number: "))
  print(55/x)
except ValueError:
  print("Please enter a number")
except ZeroDivisionError:
  print("x cannot be zero!")
```

For Python built-in exceptions, read https://docs.python.org/3/library/exceptions.html

3.2 The *else* and *finally* statements

There are two other clauses that you can add to a *try-except* block, the *else* and *finally* statements, where *else* will be executed only if the *try* clause does not raise an exception:

```
try:
  x = int(input("Please enter a number: "))
except ValueError:
  print("Please enter a number")
except ZeroDivisionError:
  print("x cannot be zero!")
else:
  print(55/x)
```

Let's say that you want to print the result of 55/*x* only if *x* is not zero and the integer conversion succeeds. In the examples from Section 3.1, we put this *print* statement directly after the conversion inside the *try* block. Either way, the statement will only be executed if the conversion statement does not raise an exception, but putting it in the *else* block is better programming practice that reduces the potential source of error that we want to handle.

The *finally* clause will be executed at the end of the *try-except* block no matter what, if there is no exception, if an exception is raised and handled, if an exception is raised and not handled, and even if we exit the block using *break*, *continue* or *return*. You can use the *finally* clause for cleanup code that you always want to be executed.

```
try:
  x = int(input("Please enter a number: "))
except ValueError:
  print("Please enter a number")
except ZeroDivisionError:
  print("x cannot be zero!")
else:
  print(55/x)
finally:
  print("Goodbye!")
```

3.3 Raising exceptions

You can raise exceptions yourself by using the *raise* statement:

```
try:
  x = int(input("Please enter a number: "))
  if x<0:
    raise ValueError("Number must be positive")
except ValueError:
  print("Input is not a number")
except ZeroDivisionError:
  print("x cannot be zero!")
else:
  print("The result is ", 55/x)
```

For this example, you can raise our own *ValueError* if the input provided by the user is a valid integer, but it is negative. In this case, it has exactly the same effect as any other exception and the flow of control will immediately exit the *try* clause at this point and pass to the *except* clause. The *except* clauses can match our exception as well, however, since there is nothing stopping you from using a completely inappropriate exception class, you should try to be consistent.

Appendix 1. Standard Exceptions

Exception	Description
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ImportError	Raised when an import statement fails.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
MemoryError	Raised when a operation runs out of memory.
RecursionError	Raised when the maximum recursion depth has been exceeded.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
ArithmeticError	Base class for all errors that occur for numeric calculation. You know a math error occurred, but you don’t know the specific error.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisonError	Raised when division or modulo by zero takes place for all numeric types.
FileNotFoundError	Raised when a file or directory is requested but doesn’t exist.

IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist. Also raised for operating system-related errors.
PermissionError	Raised when trying to run an operation without the adequate access rights.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
Exception	Base class for all exceptions. This catches most exception messages.
StopIteration	Raised when the next() method of an iterator does not point to any object.
AssertionError	Raised in case of failure of the Assert statement.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, it causes the interpreter to exit.
OSError	Raises for operating system related errors.
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
AttributeError	Raised in case of failure of an attribute reference or assignment.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

References

https://docs.python.org/3/tutorial/errors.html

https://www.tutorialspoint.com/python/index.htm