BYU IDAHO

CSE 212 | Programming with Data Structures

W03 Prepare: Reading

Table of Contents

Stacks

A **stack** is a data structure that is characterized by the order in which items are added and removed. Often called "Last In, First Out" (LIFO), the stack can be used to accomplish various tasks and can be implemented using a Python list.

I. STACK OF PANCAKES

If you were going to make pancakes for your family or friends, you probably would have a plate ready to stack the hot pancakes on as they finished cooking. Each time we put a pancake onto the stack, we call this a **push** operation. In our culinary example, we might say that each new pancake goes onto the top of the stack. However, since we are going to implement our stacks in Python, we will say that the pancake is actually added to the **back**. When we take a pancake off to eat, we call this a **pop** operation. Notice that we push and pop from the back of the stack. Removing from the middle of the stack is not generally allowed (especially at the dinner table). Notice that the pancake at the **front** is the very first pancake that was cooked. If the pancakes are made faster than they are eaten, then this first pancake would get cold. A LIFO (Last In, First Out) structure like the stack can result in data not being used for a long time. This might not work well for a rotating stock system in a grocery store, but the real benefit of the stack is the ability to remember where we have been.

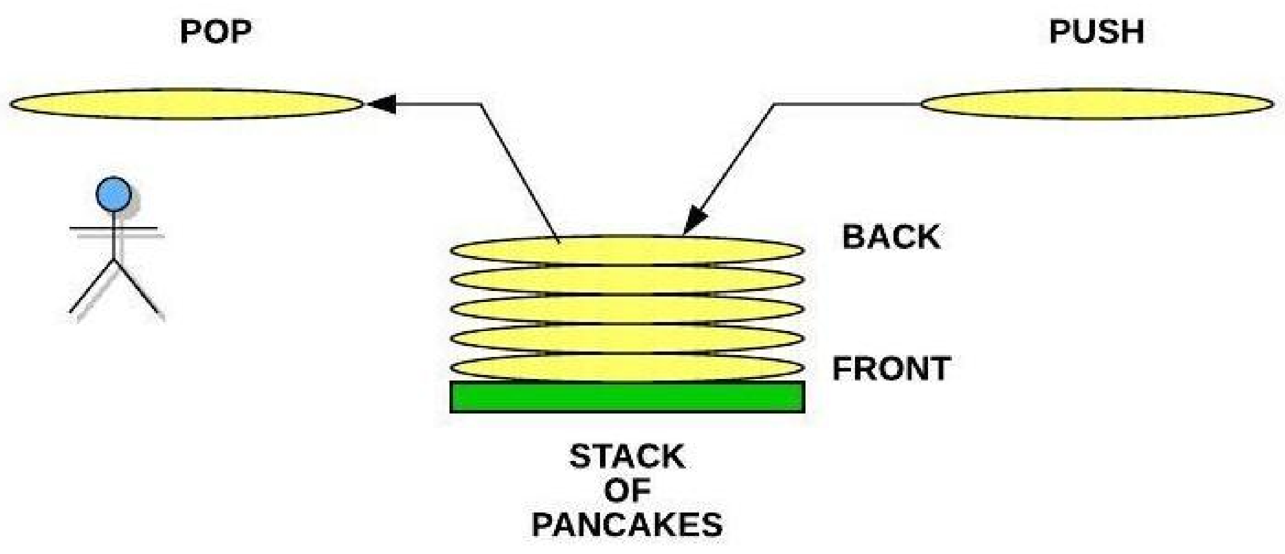


Figure 1 - Stack of Pancakes

II. THE "UNDO" OPTION AND THE STACK

One of the most common stacks that people use on a computer is related to the Undo option in word processors and editors. When we type something on the keyboard, the item is both displayed to the screen and also added to a stack. If we type the phrase **`"The rain in Spain stays mainly in the plain"`**, we would expect the following commands to be put pushed onto the stack: **`Type "The"`**, **`Type "rain"`**, **`Type "in"`**, and so forth. The last item to be pushed would be **`Type "plain"`**. If we press the Undo button, the software will pop the stack and receive: **`Type "plain"`**. The software will then do the opposite of this which would result in the word **`"plain"`** being removed from the screen.

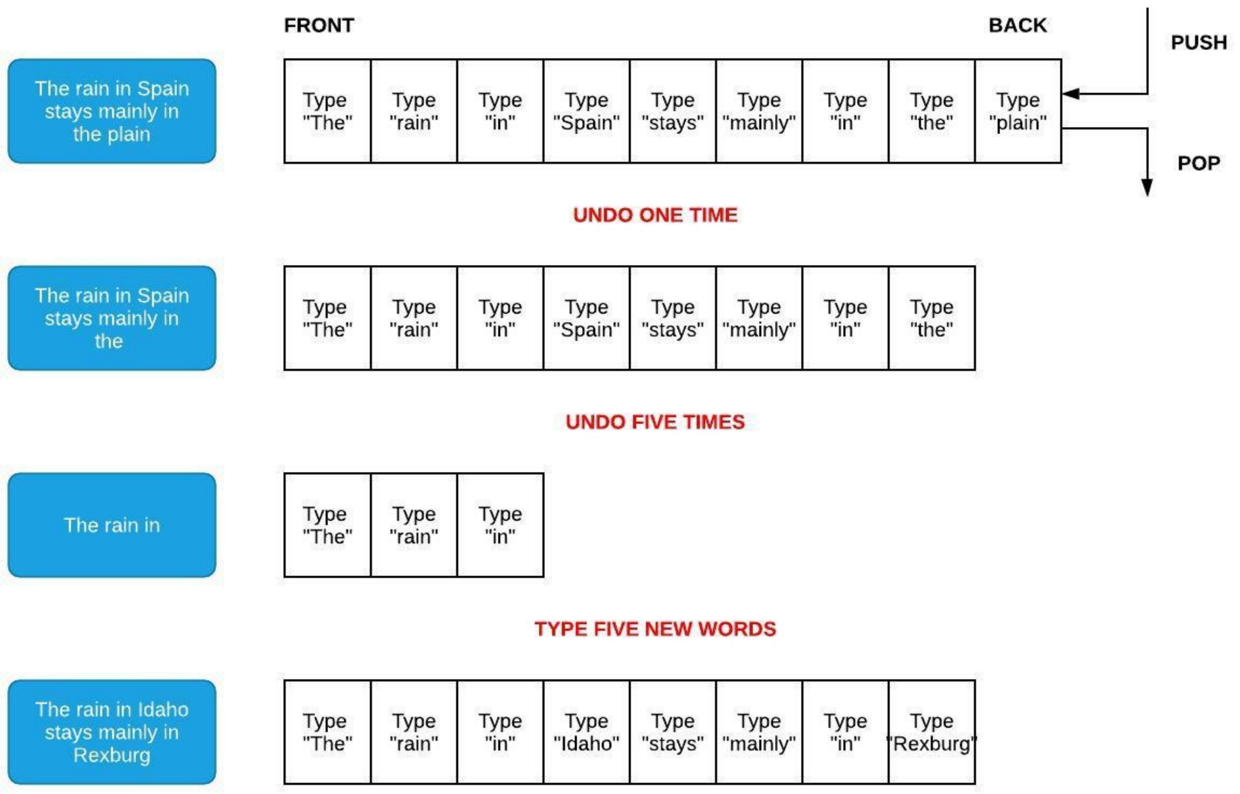


Figure 2 - Stack used to Undo Text

Since the stack is maintaining a history of what was typed, we can guarantee that pressing the Undo button will revert changes in the right order. If we popped five additional times, we would have **`"The`**

`rain in"` remaining on the screen. If we type **`"Idaho stays mainly in Rexburg"`**, we would see five new pushes to the stack. The original first three commands to display **`"The rain in"`** still remains at the front of the stack. If the Undo button is pressed enough times, then these initial words would eventually be removed.

Stacks are useful when we need to maintain history and perform an operation (eg. Undo function in an editor) backwards.

III. SOFTWARE AND THE FUNCTION STACK

Even if we didn't know what a stack was before today, we have actually been using stacks in all software we have written. When we call a function in our code, we are telling the computer two things:

- Which function we want to call
- Which function to go back to when we are done

The first of these is clear in our code. If we are currently in function A, then we expect to call function B. However, how do we tell the computer that we want to return to function A when function B is finished. This can be even more complicated by the fact that function B will need to call functions C, D, and E before it can finish. The computer accomplishes this by using a function stack. When a function is called, it is pushed to the stack. The current function running is always on the back of the stack. When the function finishes, it is popped off the stack. The result is that the function to return to is the one that is on the back of the stack.

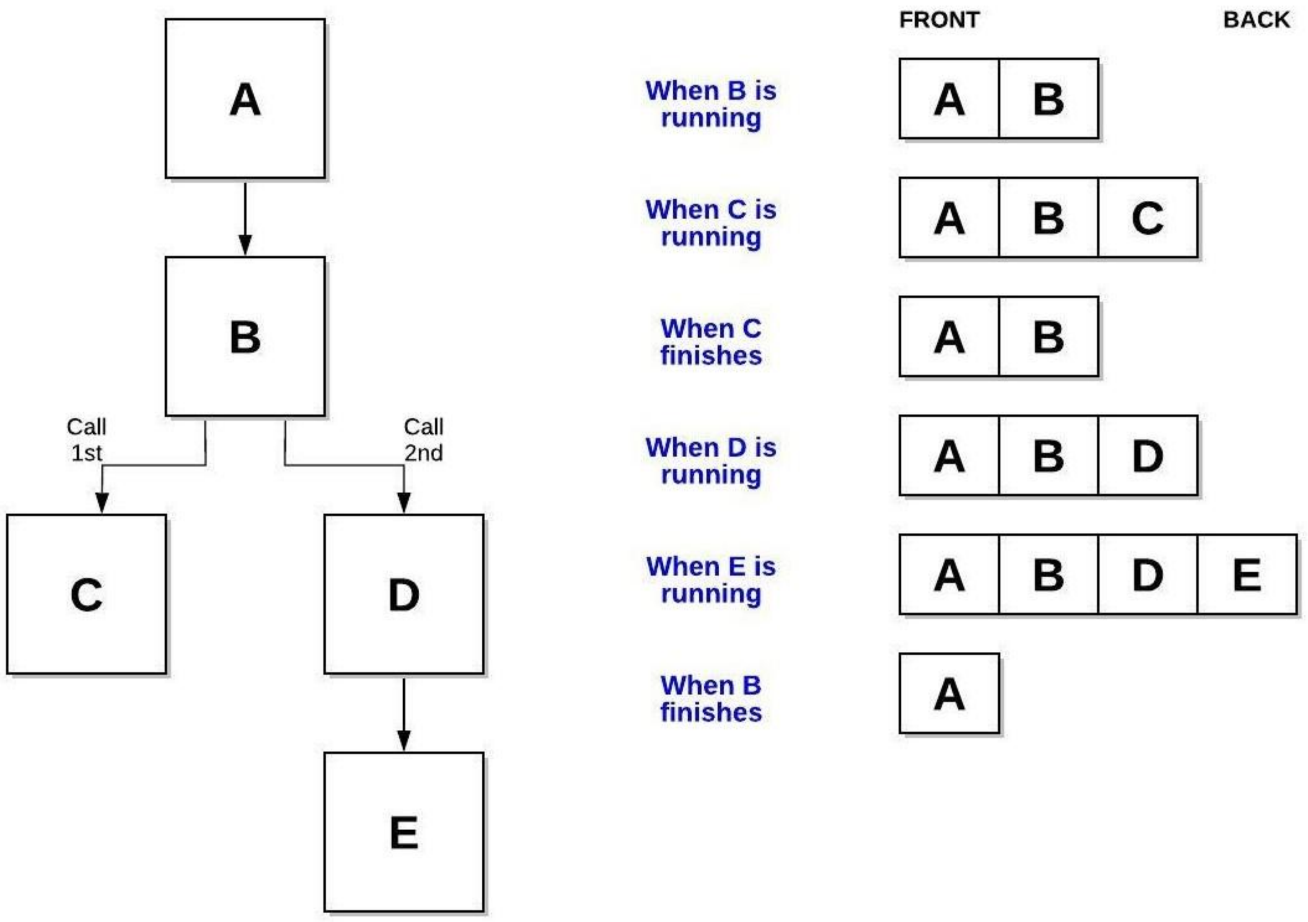
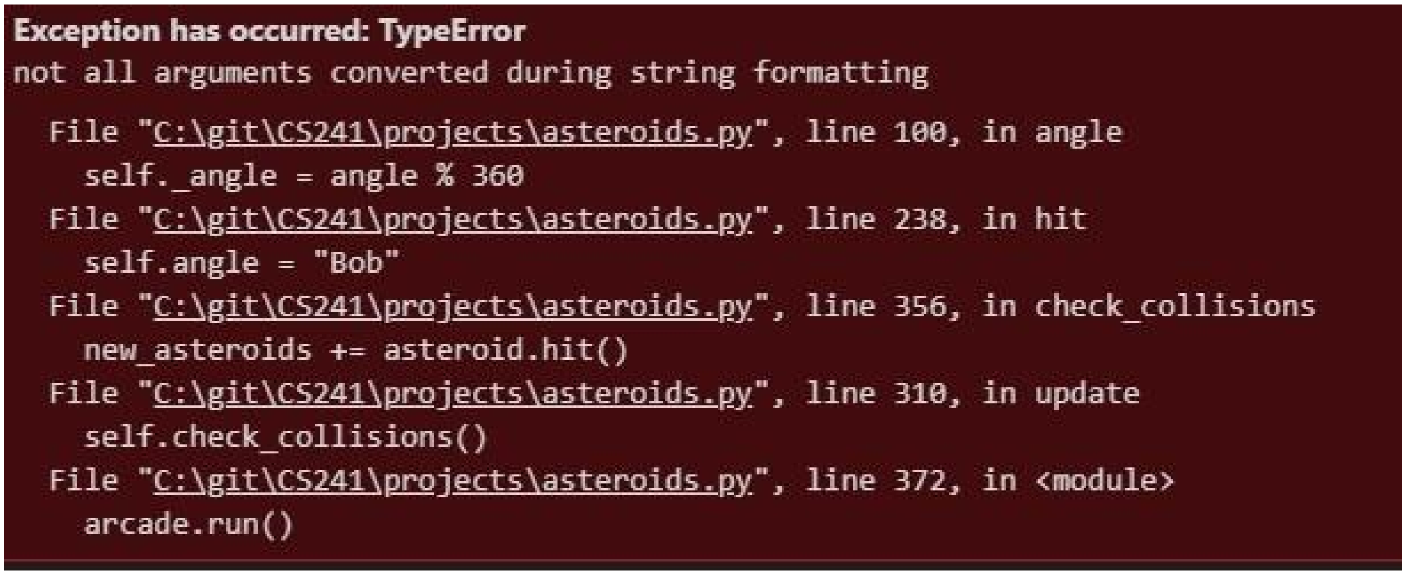


Figure 3 - Function Stack in Programming

In addition to keeping track of the function name that is running, the stack also allows us to see where in the function we were when a function was originally called as well as the memory that we were using in our function. Stacks work well for remembering where we've been and the circumstances we were in during that previous time.

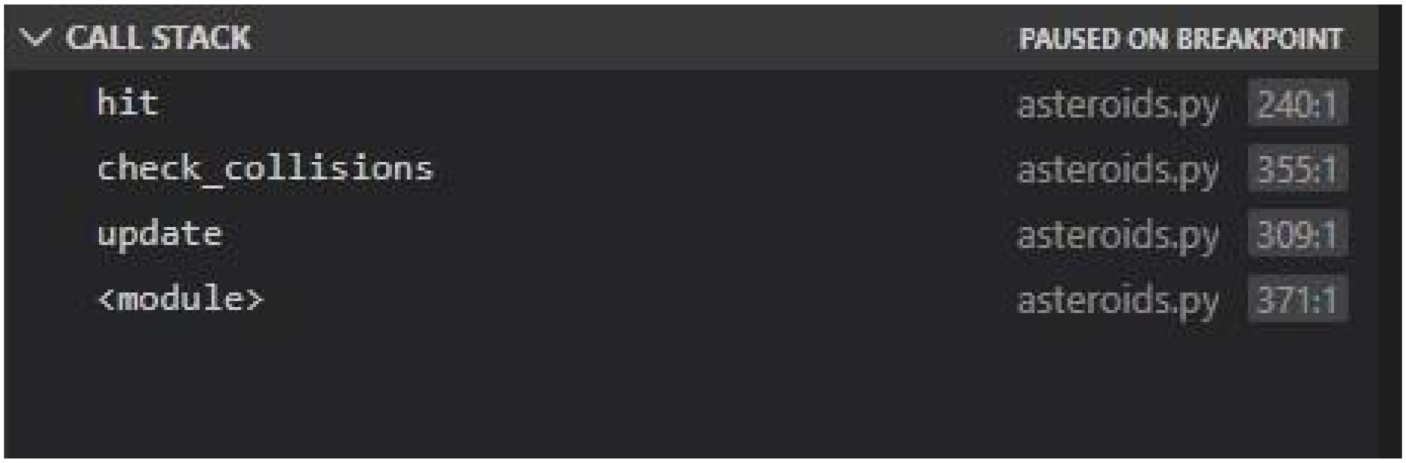
When using Python or other programming languages, we will often see error messages that look like the following. Notice that the information is showing which functions have called which functions up until the point of error. This display of information comes directly from the function stack.



```
Exception has occurred: TypeError
not all arguments converted during string formatting
  File "C:\git\CS241\projects\asteroids.py", line 100, in angle
    self._angle = angle % 360
  File "C:\git\CS241\projects\asteroids.py", line 238, in hit
    self.angle = "Bob"
  File "C:\git\CS241\projects\asteroids.py", line 356, in check_collisions
    new_asteroids += asteroid.hit()
  File "C:\git\CS241\projects\asteroids.py", line 310, in update
    self.check_collisions()
  File "C:\git\CS241\projects\asteroids.py", line 372, in <module>
    arcade.run()
```

Figure 4 - Python Exception showing Function Stack

Many code editors also include a debugger. Debuggers can be used to pause execution of software so that we can see what is occurring within our code step-by-step. Part of the debugger capability is the display of the function stack (or frequently called the call stack) when the software is paused (due to a breakpoint or an exception).



```
CALL STACK                      PAUSED ON BREAKPOINT
    hit                         asteroids.py  240:1
    check_collisions            asteroids.py  355:1
    update                      asteroids.py  309:1
    <module>                    asteroids.py  371:1
```

Figure 5 - Debugger showing Function Stack

IV. STACKS IN PYTHON

In Python, a stack can be represented using a list. To push an item to the back of the stack, the **append** function can be used on the list. To pop items from the back of the stack, the **pop** function can be used. The **pop** function will also delete it from the list. The size can be determined by using the `len` function on the list. The performance of the stack using a Python list is based on the performance of the dynamic array.

Common Stack Operation	Description	Python Code	Performance
push(value)	Adds "value" to the back of the stack.	`my_stack.append(value)`	O(1) - Performance of adding to the end of a dynamic array
pop()	Removes and returns the item from the back of the stack.	`value = my_stack.pop()`	O(1) - Performance of removing from the end of a dynamic array
size()	Return the size of the stack.	`length = len(my_stack)`	O(1) - Performance of returning the size of the dynamic array
empty()	Returns true if the length of the stack is zero.	`if len(my_stack) == 0:`	O(1) - Performance of checking the size of the dynamic array

Understanding Code Using Reviews

I. CODE REVIEWS

We have frequently been asked to write code for either school or work. However, how often have we been asked to complete the companion activity of reading code? We have likely looked at code from websites and books, but there is a significant difference between looking at code and reading code. When we read code, we are attempting to understand the code like we would a book. If we open a book and read a few random pages, we might get a high level summary of what the book is about. However, to fully understand the book, we would need to not only read the book cover to cover, but we would also need to become acquainted personally with the diverse set of characters, follow a potentially winding plot, and discover the underlying messages woven in the story from the author. This type of reading takes effort. Reading code for understanding takes an equal amount of effort.

There are multiple reasons why we might be asked to read code in our teams:

- A team member has written some code and has asked other engineers to review the code for correctness against the design and for compliance against company standards.
- A team member is asked to modify some code that was written by a former employee with or without the aid of a well documented design.
- A team member needs to integrate their software with an external library which included several code examples that demonstrate how to properly use the library.

When we read code others have written, we refer to this activity as a **review**. A review should follow a methodical process. Many companies will include a review checklist for engineers to use to ensure that they both understand the code and that they have checked all the coding standards. When you review code, there are several strategies that you can use including the following:

- Read code "cover to cover"
- "Execute" the code manually
- Analyze the use of data structures

Each of these methods will be discussed below. When working with these methods, we should try to employ principles learned from the scientific method. The scientific method requires us to form a hypothesis about what we think the code should be doing. As we read through the code, we will test our hypothesis and look at the results. If we are incorrect, then we can correct our misconceptions and form a new hypothesis. This iterative process is necesessary to fully understand code. There are no shortcuts. We can be tempted to search for the answer online or in a book. While this may produce a faster answer, we will have not obtained full understanding of the code.

In this process, it is possible that we might find defects in the code that we were given. Code reviews are a common tool for increasing software quality. Performing these steps will require us to keep a good notebook to record our hypothesis, experiments, and conclusions. A good reviewer will always have a pen and paper ready to complete their task.

II. READ CODE "COVER TO COVER"

Unlike a book, code does not begin on page one. We need to find where the code begins and follow it as it calls functions, runs loops, and branches in different directions with decision statements. If the code has multiple functions, the creation of a **structure chart** (also called a calling tree) will be helpful. The structure chart will use boxes to represent functions and arrows to represent functions calling functions. Frequently drawn with the starting function at the top and working downwards, these diagrams can help us navigate through the code. On the arrows, we may frequently write the inputs and outputs related to each function. This will help us better understand the data in the code and which functions are responsible for creating, modifying, and using that data.

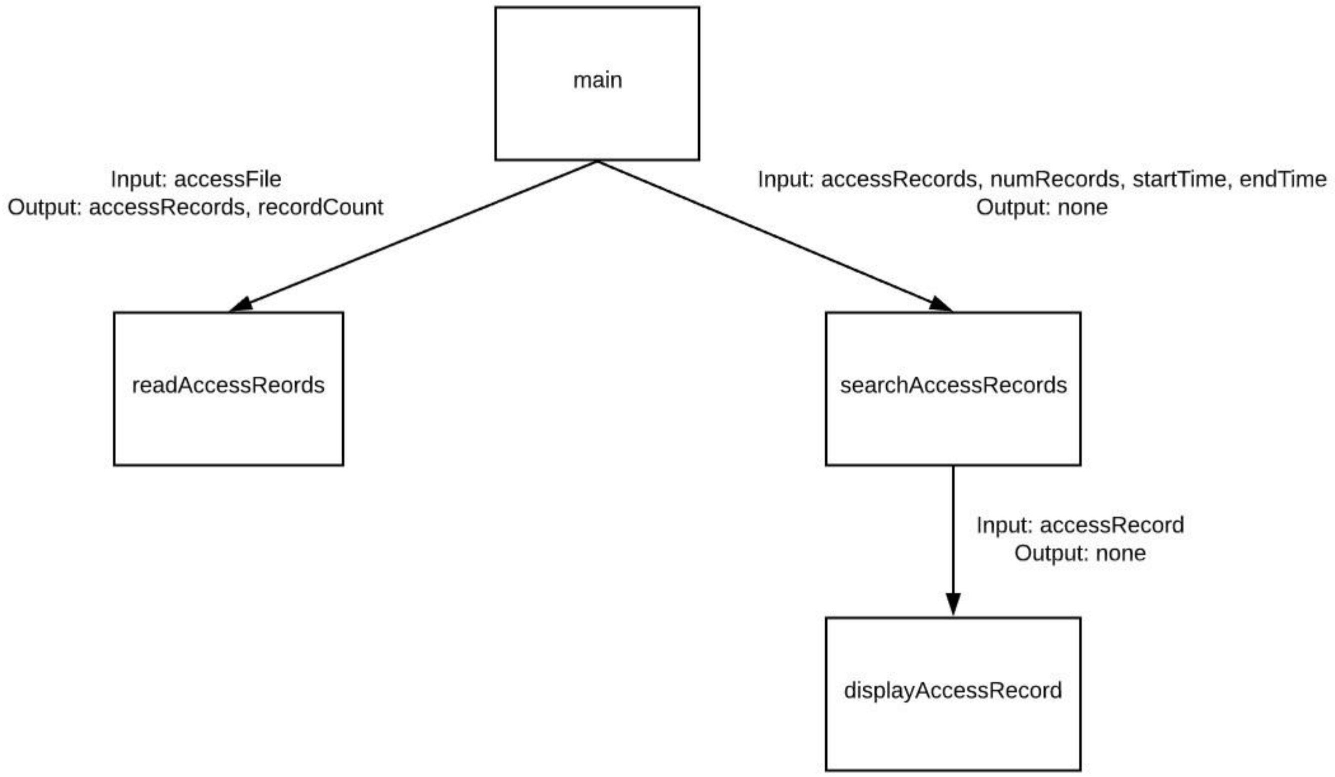


Figure 6 - Structure Chart

If the code contains classes, then creating a **UML** (Unified Modelling Language) class diagram to show the classes (with member data and member functions) and the relationships between the classes will help us to visualize the software and enable us to read different sections of the code based on their dependencies to others. For example, in the diagram below, we can see that the Order HAS-A (object composition shown with the filled in diamond) list of Products and that each Product IS-A either a PerishableProduct or an ElectronicProduct (inheritance shown with the open triangle). With this drawn, we would probably start with understanding the Product base class first since it has no dependency on other objects. Second, we would review the different types of Products. Finally, we would review the Order class which contains the list of the Product objects that we already understand.

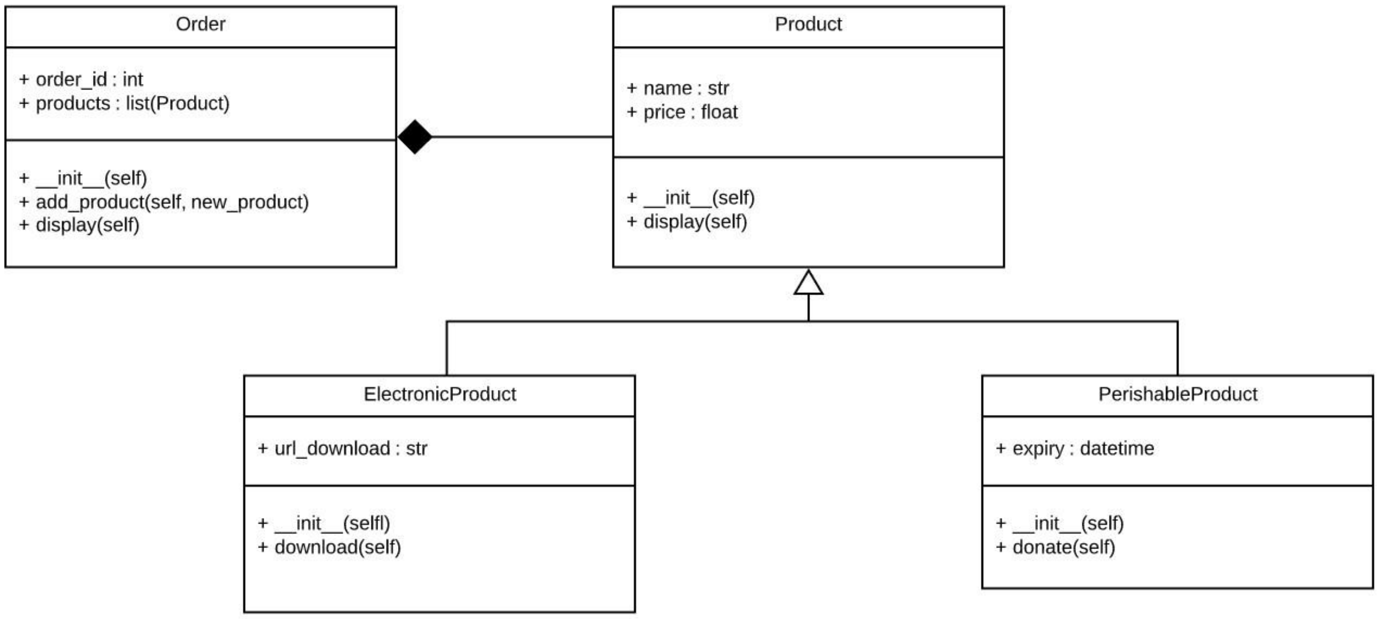


Figure 7 - UML Class Diagram

When we are looking at a single function, it can be useful to diagram the behavior of the function. Simple **flow charts** can quickly give us a better perspective of the loops and decisions in our code. In the flow chart, use diamonds to represent decisions, boxes to represent actions, and arrows to show flow.

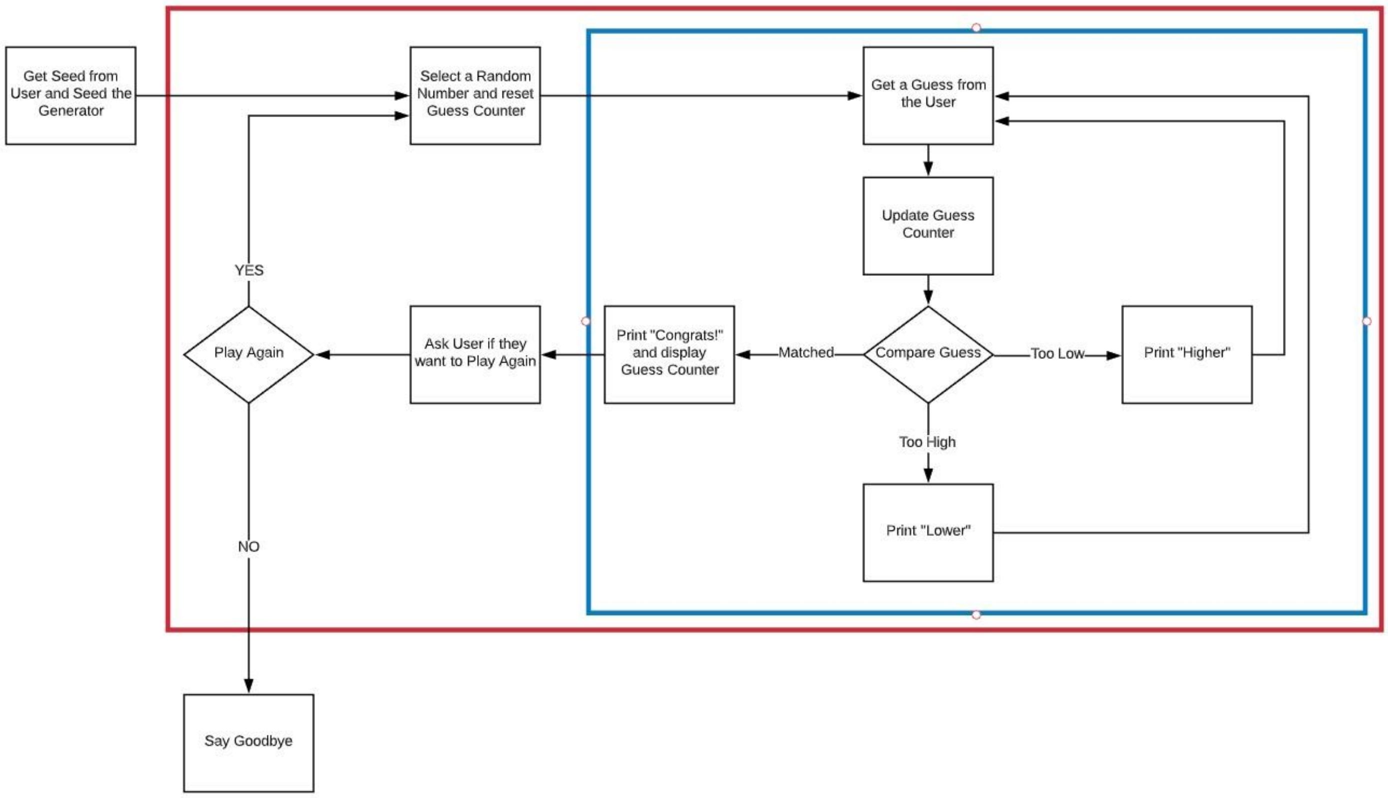

Get Seed from User and Seed the Generator
Select a Random Number and reset Guess Counter
Get a Guess from the User
Update Guess Counter
Compare Guess
Matched
Too Low
Too High
Print "Higher"
Print "Lower"
Print "Congrats!" and display Guess Counter
Ask User if they want to Play Again
Play Again
YES
NO
Say Goodbye

Figure 8 - Flow Chart

III. "EXECUTE" THE CODE MANUALLY

It is not always practical to run code that we are given. If we can, then running the code with inputs that we generate can be helpful to understand the software. However, the goal in this process is to understand the software without running the code on the computer. Instead, we are going to run the software in our minds and on paper.

If we created diagrams in the previous step from our reading of the code "cover to cover," then we can run the code from the diagrams. This is an incomplete approach but is very helpful in gaining more understanding of what the software will perform.

To execute code manually in our minds and on paper, we must start at the beginning (or if we are looking at one piece of the software, perhaps start at the beginning of one of the functions). If inputs are provided at the beginning (or at any other place along the way), we will have to develop useful inputs to see what will happen. For example, consider the following code:

```python
def do_something(text):
    new_text = ""
    for letter in text:
        if letter != " ":
            number = ord(letter)
            new_letter = chr(number + 1)
            new_text += new_letter
        else:
            new_text += letter
    return new_text
```

The function needs text and so we will propose some text like "Hello". We will then step through each line of code and record in our notebook the value of each variable. If we come across a code function that we don't understand (e.g. the ord and chr commands), then we will need to go online to read about those. The `ord` function will return the ASCII numeric code that represents a character. The `chr` function will convert an ASCII numeric code into a letter again. When we finish the code we get `Ifmmp` which appears to be a form of simple childhood encryption in which each letter in the original text is changed by one letter higher in the alphabet. When we ran the program, we noticed that you had to check if the letter was a space. It would be good to try to the test again with spaces. If we tried "Hello World", we end up with `Ifmmp Xpsme`. Not only does the function perform the encoding, but it also preserves the spaces.

text	letter	number	new_number	new_letter	new_text
Hello	H	72	73	I	I
	e	101	102	f	If
	l	108	109	m	Ifm
	l	108	109	m	Ifmm
	o	111	112	p	Ifmmp

IV. ANALYZE THE USE OF DATA STRUCTURES

When code contains a data structure like a list or a stack (and others that we will learn during the course), we should consider why the data structure was used. Data structures are used both for storing information but also to use the information in different ways. Looking at the readings above, a stack is used if we want to remember where we have been and potentially go reverse or backwards. Knowing this capability, you can form a hypothesis about what the code is doing if you see a stack being used in the code. During the activities this week, you will be reading code that uses stacks. How the stack is being used will help you better understand the purpose and behavior of the code.

Key Terms

» **back** - Refers to the location in a stack where a push and pop occurs. The last item put into the stack is found in the back.

» **flow charts** - A diagram that models the behavior of a program, algorithm, or function. Actions are shown in boxes, decisions shown in diamonds, and arrows are used to show execution flow.

» **front** - Refers to the location in a stack where the first item added to the stack can be found. Traditionally, this is index 0 of a dynamic array.

» **pop** - The operation to remove the last item pushed to the stack. The item from the back of the stack is removed and returned.

» **push** - The operation to add a new item onto the stack. The item is placed at the back of the stack.

» **review** - A formal process of ensuring code is written correctly. Code is usually reviewed against the design and coding standards. Frequently, checklists are used to help the reviewer.

» **stack** - A data structure that follows a Last In, First Out (LIFO) rule. The stack is used to reverse data or remember previous data including previous results.

» **structure chart** - A diagram showing which functions call which functions. Frequently, the arrows used to show function calls also include parameters that are passed between the functions.

» **UML** - Unified Modeling Language. A formal modelling language to represent object-oriented designs. UML includes many types of diagrams including class diagrams, activity diagrams, and state diagrams.