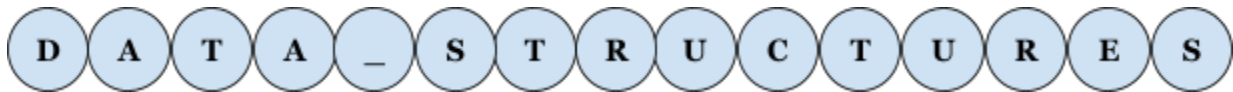

CS261 Data Structures

Coding Guides and Tips

Fall 2023

Tips, Styling, and Debugging



Built-in Python Restrictions.....	3
Restricted Data Structures and Behavior.....	3
Private Variables and Methods.....	4
Built-in Functions.....	6
Styling Your Code.....	6
Why a Style Guide?.....	6
General.....	7
Built-in Functions.....	7
Returning None.....	7
Leaving 'pass' in a Function.....	7
Returns and Exceptions Instead of Else.....	8
For and While Loops.....	8
Range.....	9
Conditional Compression.....	9
Breaks and Continues.....	11
Type Hinting.....	12
Refactoring.....	12
Coding Tips for Efficiency and Readability.....	16
Docstrings.....	16
Naming Conventions.....	17
Variable Naming.....	17
Function Naming.....	18
White Spaces.....	18
Descriptive Comments vs. Cluttering Comments.....	19
Comment Alignment.....	20
Tips for Programming Assignments.....	20
General Tips.....	20
Your Submission Timed Out.....	23
The Autograder Failed to Execute Correctly.....	24
Error Handling.....	24
Terminal Errors.....	24
Common Errors.....	25
AttributeError:.....	25
TypeError:.....	25
IndexError:.....	26
NameError:.....	26
ModuleNotFoundError.....	27
Expected Exception.....	27
Feedback on Gradescope.....	28
Failed Tests.....	28
Miscellaneous tips.....	28

Built-in Python Restrictions

This section is a quick overview of data structures and functions that are forbidden, and some built-in functionality that is allowed. These lists are by no means exhaustive, and we reserve the right to amend them (we would of course let you know). If you have a question about any of these topics, or anything else for that matter, please use Ed Discussion or contact a staff member during their Office Hours on Teams.

Restricted Data Structures and Behavior

There are four standard built-in data structures in Python: list, set, dictionary, and tuple. Tuples are the only data structure you are allowed to use in this course, but only under specific use-cases (see examples below).

Additionally, every callable method from a data structure is also off-limits, including methods of the tuple. So, instances of tuples are allowed, but their methods are not.

```
# Examples of Restricted Data Structures

# Lists
list1 = []
list2 = list()
list3 = [None] * 10
list_comprehension = [number for number in range(5)]

# Sets
set1 = {1, 2, 3}
set2 = set()
set_comprehension = {letter for letter in "apple"}

# Dictionaries
dictionary1 = {'a': 1, 'b': 2}
dictionary2 = dict()
dictionary_comprehension = {number: number**2 for number in range(10)}
```

```
# Tuple - The permitted built-in data structure
a_tuple = (10, 20)

# can unpack tuples like so
first_value1, second_value1 = a_tuple

# or access individually
first_value2 = a_tuple[0]
second_value2 = a_tuple[1]
```

To be clear: you **are allowed** to use bracketed indexing (eg. `first_value2 = a_tuple[0]`) **or** the Pythonic “unpack” to access the values contained in a tuple, but you **can not** use any of the tuple’s built-in methods shown below:

```
# Not allowed
built_in_method = a_tuple.
```

m	index(self, __value, __start, __stop)	tuple
m	count(self, __value)	tuple
	par	(expr)
m	__init__(self)	object
m	__contains__(self, x)	tuple
p	__class__	object
m	__add__(self, x)	tuple
f	__annotations__	object
m	__class_getitem__(cls, item)	tuple
m	__delattr__(self, name)	object
f	__dict__	object
m	__dir__(self)	object

Ctrl+Down and Ctrl+Up will move caret down and up in the editor. [Next Tip](#)

Private Variables and Methods

Variable names that begin with an underscore are considered **private** in Python. You are not allowed to access them directly outside of the class. You may access private variables if you are inside the class (by using `self`, or by another means as discussed below), or access them indirectly by using a **public** method if outside the class definition.

Variables and methods that are considered public do not begin with an underscore, and may be accessed directly anywhere and without using a corresponding method.

The following examples reference `static_array.py`, a file that you will be given on the [Assignment 1: Python Fundamentals Review](#) page. Spend some time looking through the file when it opens!

Consider the following code snippet as being in the `__main__()` block, or anywhere that isn't in a method of the `StaticArray` class

```
# Use the public method, not the private variable

an_array = StaticArray(5)

# Bad! _size is a private variable
size_of_array_bad = an_array._size

# Good! length() is a public method
size_of_array_good = an_array.length()
```

Since `length()` is a method of the `StaticArray`, it can access the private variable, `_size`:

```
def length(self) -> int:
    """Return length of the array (number of elements)."""
    return self._size
```

There is one more consideration regarding private variable/method access. There will be times when a method inside a class is given an object of that class as an argument (stay with me) or such an object is instantiated directly in the method. You may be wondering if this other instance of the same class can have its private variables/methods accessed directly. It can; it's called access on a per-class basis. See [this StackOverflow discussion](#) (although they discuss in terms of C++, Python appears to adhere to the same policy, as can be ascertained from the screenshots below).

The following example is a method in the StaticArray class that accepts a StaticArray object (*note that the following is used for illustrative purposes only - you won't find this method in the file you're given*).

Edge case regarding private access

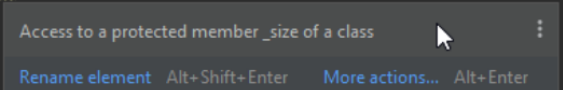
```
def print_size_of_another_array(self, array: "StaticArray") -> None:
    """Print the size of the given static array."""

    # This direct variable access is permitted since the array
    # parameter is an instance of a StaticArray,
    # and this method is inside the StaticArray class definition.
    print(f"The array parameter's size is {array._size}")
```

The reason the interpreter doesn't present a warning is because the array is given the type hint of StaticArray (type hints are discussed in the [Style Guideline](#) section). The given object is a StaticArray and as such, the StaticArray object's private variable, `_size`, can be directly accessed.

```
def print_size_of_another_array(self, array) -> None:
    """Print the size of the given static array."""

    # The interpreter doesn't know what type of object array is,
    # and gives a warning about accessing a private variable.
    print(f"The array parameter's size is {array._size}")
```



Type hint removed. Interpreter says, "Don't even think about it!"

Built-in Functions

There are many functions that are built directly into the Python environment that do not necessarily require a data structure to use them. For this class, you can only use the following built-in functions:

- [abs\(\)](#)
- [enumerate\(\)](#) **starting with A2*
- [help\(\)](#)
- [int\(\)](#)
- [len\(\)](#)
- [min\(\)/max\(\)](#)
- [print\(\)](#)
- [range\(\)](#)
- [tuple\(\)](#)
- [zip\(\)](#)

For reference, further examples, and explanations please see [this list of Python's built-in functions](#).

Styling Your Code

This section will cover some guidelines for styling your code logically. Failure to follow these guidelines will not necessarily result in a loss of points, but always be sure to review the assignment instructions for each and every major assignment to ensure you do not lose points unnecessarily.

Why a Style Guide?

Having a guideline to follow in terms of code style will help both yourself as the code author, and anyone who happens to read the code you write. Some teams in professional projects have their own style guidelines so that code appearance is uniform between authors, improving readability and reducing the amount of time spent trying to figure out what code is doing.

Applying a style guide to your code is most effective when done consistently. Applying it consistently inside a function or module is of greatest importance, followed by applying it consistently across an entire assignment. There are times when you should not apply the guidelines laid out here, specifically:

- If the guideline makes your code less readable.
- If you are trying to be consistent with older code that does not adhere to the guideline.
- Certain situations not listed here, where you make the decision to not apply the style guidelines based on your individual needs.
- Just because an example here does something, does not mean you're allowed to use it in the assignments. For example, **list comprehension is not allowed for any of the assignments**, but an example below uses it to showcase a styling topic.

General

- Lines should not exceed 79 characters.
- Less is more, such that, it is usually possible to condense the code you are writing by removing unnecessary conditionals, refraining from commenting on every line of code, and not overthinking it. After writing a function, think of it as a rough draft and go back to improve on your code after you've discovered the logic behind the task at hand.

Built-in Functions

[List of permitted built-in functions](#)

Returning None

If your code does not return a value, then avoid explicitly returning 'None'. This is more about readability than it is about resource impact. A function that does not require a returned value will terminate on its own after the last line of said function executes.

Examples of returning none when it is not needed:

```
def reduceValues(self, reduction : int) -> None:
    """
    Reduces all the values in self._data by value reduction
    """

    for x in range(len(self._data)):
        self._data[x] -= reduction

    #This return is NOT needed!
    return

def reduceValues(self, reduction : int) -> None:
    """
    Reduces all the values in self._data by value reduction
    """

    for x in range(len(self._data)):
        self._data[x] -= reduction

    #This return is NOT needed!
    return None
```

Leaving 'pass' in a Function

Always delete 'pass' from your functions if it is not required for some conditional logic (it almost never is - if you are using pass you might need to refactor that logic). As with the guideline above, this is more about code readability than resource impact. 'Pass' is included in most of the function skeletons for the course assignments; it is added to prevent your code from throwing errors because of declared functions with no logic.

```
def myFunction(self):
    """
    TODO : Implement this function
    """

    #REMOVE THIS PASS AFTER IMPLEMENTATION!
    pass
```


Returns and Exceptions Instead of Else

Conditionals are powerful tools when used properly, but can reduce the readability of your code by creating unnecessary indentation and clutter. Some examples include adding an 'else' statement after a conditional that would return out of the function, such as raising an exception. Because the else would trigger regardless, and there are no other conditional checks needed, the 'else' does not need to be written as it is assumed by the program. Another example is when your function can do 1 of 2 things, you can remove the else statement by adding a return inside the if conditional to reduce unnecessary indentation.

<pre># Bad Practice def foo(): if index > arr.length(): raise DynamicArrayException else: foo(x, y) bar(x, y)</pre>	<pre># Bad Practice def foo(): if self.head is None: self.head = node(value) else: newNode = node(value) foo(newNode) bar(newNode)</pre>
<pre># Good Practice def foo(): if index > arr.length(): raise DynamicArrayException foo(x, y) bar(x, y)</pre>	<pre># Good Practice def foo(): if self.head is None: self.head = node(value) return newNode = node(value) foo(newNode) bar(newNode)</pre>

For and While Loops

- For loops iterate a set number of times, whereas while loops iterate until a specific condition is met.
- For loops are used to iterate through, while loops until a condition is met.
- For loops know the number of times they will loop, while loops do not.

```

# Iterates a known number of times
for x in range(10):
    print(x)

# Iterates across a known objects length
# Note, this will NOT work on your Static and Dynamic Arrays
for value in tempArr:
    print(value)

# Iterates until a condition is met
while x > 0:
    print(x)
    x = x // 2

# Iterates until a condition is met
while arr.length() > 0:
    print(arr.pop())

```

Range

Range should only be used when we need something performed a certain number of times, not for when we want to iterate across a list or conditional requirements (such as a while loop).

```

# Bad Practice
def produce_evens(total):
    result = []
    num = 0
    while len(result) < total:
        result.append(num)
        num += 2

# Good Practice
def produce_evens(total):
    result = []
    for x in range(0, total*2, 2):
        result.append(x)

```

Conditional Compression

When applicable, you should compress conditional statements into a single line. There are many reasons for this, including readability and control flow.

<pre># Bad Practice if x > 0: if x // 2 == 0: print(x)</pre>	<pre># Bad Pratictce if x > 0: print(x) elif x // 2 == 0: print(x)</pre>	<pre># Bad Practice def bar(): if x > 0: if x // 2 == 0: foo(x) return if x > 0: if x // 3 == 0: foo(x+1) return if x > 0: if x // 5 == 0: foo(x+2)</pre>
<pre># Good Practice if x > 0 and x // 2 == 0: print(x)</pre>	<pre># Good Pratictce if x > 0 or x // 2 == 0: print(x)</pre>	<pre># Good Practice def bar(): if x > 0 and x // 2 == 0: foo(x) elif x > 0 and x // 3 == 0: foo(x+1) elif x > 0 and x // 5 == 0: foo(x+2)</pre>

Breaks and Continues

Breaks and Continues are tempting when writing a loop, especially ones that seem to never stop growing. It may be hard to write a loop condition that meets all the provided restrictions, and with that, a break may feel like the best way to get out of your loop.

However, these one-token statements bloat your code with unneeded conditionals and lines of code, and can actively hinder the debugging process as your loop condition is now embedded somewhere deep within the loop, instead of attached directly to it. It also reduces readability.

Imagine it like so:

You as a reader see while $x < y$ and you know this loop runs until x is greater than or equal to y , which will then break the loop. However, 15 lines down, you find the conditional statement that if $x*y // 2 == 0$, we want to break from the loop. Now this can change your initial assumptions of the loop's actual functionality.

```
# Bad Practice
randNum = random.randrange(1,10)
# Adds random integers to a list to a max size of 10
# If randNum is a 4, we break from the loop
while len(myList) < maxLength:

    if randNum == 4:
        break

    if randNum // 2 == 0:
        myList.append(randNum * 2)
        randNum = random.randrange(1,10)
        continue

    if randNum // 3 == 0:
        myList.append(randNum * 3)
        randNum = random.randrange(1,10)
        continue

    else:
        myList.append(randNum * randNum)
        randNum = random.randrange(1,10)
```

```
# Good Practice
randNum = random.randrange(1,10)
# Adds random integers to a list to a max size of 10
# If randNum is a 4, we break from the loop
while len(myList) < maxLength and randNum != 4:

    if randNum // 2 == 0:
        myList.append(randNum * 2)

    elif randNum // 3 == 0:
        myList.append(randNum * 3)

    else:
        myList.append(randNum * randNum)

    randNum = random.randrange(1,10)
```

```
# Adds values from an array to a new array
# until it comes across a value x
def foo(x):
    tempList = []
    for node in myList:
        if node.value() == x:
            break

    tempList.append(node)

    return tempList
```

Type Hinting

Type hints are an optional, but highly recommended, tool in python used to identify parameter and return types.

- In this example, our parameter num is an integer(num : int) and we return an integer(-> int:)

```
def square(num : int) -> int:
    return num * num
```

- In this example, our parameter is a string (name : str) and we do not return anything from the function (-> None:)

```
def hello_name(name : str) -> None:
    print("Hello " + name)
```

- In this example, our parameter is an array of integers (arr : list[int]) and we return a tuple of 2 lists of integers (-> tuple[list[int], list[int]])

```
def split_list(arr : list[int]) -> tuple[list[int], list[int]]:
    return (arr[:len(arr)//2], arr[len(arr)//2:])
```

- In this example, our parameter is an array of integers and strings (arr : list[int, str]) and we return a tuple of two list, one with integers, the other with strings (-> tuple[list[int], list[str]]). The reason we can safely assume

'else' here is because our type hint clarifies we will receive a list of integers and strings only. Python will make a fuss if this condition is not met, due to the type hinting.

```
def split_strings_from_ints(arr : list[int, str]) -> tuple[list[int], list[str]]:
    intArr = []
    strArr = []
    for ele in arr:
        if type(ele) is int:
            intArr.append(ele)
        else:
            strArr.append(ele)
    return (intArr, strArr)
```

Refactoring

Refactoring is a vital component to creating easily readable code while simultaneously improving flow of logic.

Functions generally shouldn't be longer than 10-20 lines. This is not a hard-set rule, but, if you have a section of code that goes beyond 10-20 lines, ask yourself: 'Can this be broken down into a function on its own' or 'Can I remove redundancy'?

Another reason you may want to break code apart into helper functions is if you have logic you repeat more than once. This can be twice within the same function, or used across multiple different functions.

Do not break your code apart into functions with no reasoning. If you surpass 20 lines of code, don't feel like it is necessary to create a helper function just for the sake of it. Refactoring is to help with logical flow, and by breaking code apart without logic, you aren't doing yourself or the readers a service.

Below is an example (an explanation follows) of how you can go about breaking code apart into smaller helper functions. You can also copy-paste this into your IDE and follow along via the debugger (add fooMain() at the bottom of your code and add a breakpoint there).

```
def fooMain():
    ...

    Get input from a user to create a matrix of inputted size
    ...

    # (matrixX, matrixY, squared(T/F))
    userInput = get_input()

    matrix = create_matrix(userInput[0], userInput[1], userInput[2])
    print_matrix(matrix)
```

```

def get_input():
    """
    Prompt user for matrix size input
    """
    # Prompts
    matrixX = input("How many rows do you want your matrix to have :")
    matrixY = input("How many columns do you want your matrix to have :")
    squared = input("Do you want your Matrix Squared (T or F) :")

    # Squared needs to be True or False
    while squared not in ("T", "F"):
        squared = input("Do you want your Matrix Squared (T or F) :")

    # Set squared to boolean value
    if squared == "T":
        squared = True
    else:
        squared = False

    return (int(matrixX), int(matrixY), squared)

def create_matrix(matrixX, matrixY, squared = False):
    """
    Create a list of list representing a matrix based on user input
    """
    arrs = []

    # Fills matrix
    for x in range(matrixX):
        arrs.append(create_list(matrixY, x, squared))

    return arrs

def create_list(total, skip, squared = False):
    """
    Creates a single list of size total
    """
    if squared is False:
        return [x*skip for x in range(total)]

    return [(x*skip)*(x*skip) for x in range(total)]

```

```
def print_matrix(matrix):
    '''
    Prints the matrix in a readable way
    '''
    for row in matrix:
        print(row)
```

Explanation:

- fooMain -
 - This section is the main portion of our program, which in essence is creating a matrix (a list of list) based on 3 different inputs from the user via get_input.
 - After we get the user input, our program calls create_matrix and stores the returned value (a matrix whose dimensions and values we're given to use via get_input) into the variable matrix.
 - Finally, our program calls print_matrix to send the matrix to the terminal in an easily readable format.
- get_input -
 - Notice how in fooMain this is one line of code. However, if we look at it, it is actually a little beefy function.
 - First, we ask the user for 3 different inputs. The first two are our X-axis and Y-axis of our matrix. The third is a Boolean expression to see if we want our values inside the list to be squared.
 - After getting the initial 3 inputs, we have a while conditional check to make sure we get a True or False response from the user.
 - Then, we transform the squared value to a Boolean.
 - Finally, we return our user inputs in the form of a tuple, an easy-to-use data structure that is able to hold and pass information in a compact form throughout our functions.
- create_matrix -
 - The tuple from user_input got broken into 3 sections, and passed to this function with our X-axis, Y-axis, and our squared boolean.
 - This method actually utilizes ANOTHER helper function, create_list. Because a matrix is a list of list, we initially have to create the list inside our main list.
 - We use a for loop with a range() to create matrixX number of list (our rows), and append to our arrs variable an array that is returned by create_list.
- create_list-
 - create_list is a short and simple function that returns one of two things.
 - In the case our squared variable is False, we return via list comprehension that takes skip (x value passed by create_matrix that is represented by its position in the for loop)

multiplied by `x` (defined in the list comprehension). This means in a matrix of 3x3 we'd have the values `[0, 0, 0]` in the first list, `[0, 1, 2]` in the second list, and `[0, 2, 4]` in the third list.

- In the case our squared variable is `True`, we return via list comprehension that multiplies `skip*x` by itself. This means in a matrix of 3x3 we'd have `[0, 0, 0]`, `[0, 1, 4]`, `[0, 4, 16]`.
 - Note: This function uses list comprehension, but you will not be allowed to use list comprehension in this class.
- `print_matrix` -
 - Finally, we print our matrix by calling this function. Again, it is just a simple loop that prints each row on a new line.

Coding Tips for Efficiency and Readability

The chances that you will be the only person to ever read the code you write is extremely slim. You may be working on a large-scale project where a whole team of people need to go over code you've written, a potential employer may be looking at a project you've pushed to GitHub, or in the case of this class, a ULA is looking for ways to give you feedback on how to improve your code. In all these scenarios, it is important that people other than yourself understand not only what your code is doing, but also that they can do so with ease.

Because of this, we have made this brief style guide to help you write clean code, not only for this class, but for all future projects and assignments.

Please note that this section contains information about docstrings and not following this information will result in lost points in your assignments.

Docstrings

Not only are docstrings important, **you will lose points if you do not add them.** Documentation in Python is centered around docstrings, so implementing them properly and with thought is the best way to increase the readability of your code. Your docstrings should be written such that any random programmer will know how to use your code with no other knowledge.

- A good docstring, **at minimum**, includes a brief description of the function, parameters, and return.
 - This can be done as shown below, but there are alternative methods.
- **A docstring is NOT the assignment's function description.**
- They shouldn't stretch into paragraphs, with each element taking up only about a line.
- Each 'section' of the docstring should be separated by a line.


```

# Bad Practice
def foo(arr, y):
    tempArr = []
    for num in arr:
        if num % y != 0:
            tempArr.append(num)
    return tempArr

# Good Practice
def foo(arr : list[int], y : int) -> list[int]:
    """
    Create a modified list of values not divisible by y from arr.

    :param arr: an array of integers
    :param y: integer to divide against for a remainder of 0

    :return: modified list w/o values divisible by y
    """
    tempArr = []
    for num in arr:
        if num % y != 0:
            tempArr.append(num)
    return tempArr

# Utilizing Python's Docstrings
print("__doc__:")
print(mysteryMethod.__doc__)

print("help:")
help(mysteryMethod)

```

Naming Conventions

Your naming conventions must remain consistent throughout your program.

Picking and sticking to a naming convention is key to creating consistent, readable code.

The three popular conventions you are recommended to use are camelCase, snake_case, and PascalCase.

One reason naming conventions are so important is because of predictability. If you are working on a project and know your function name is "insert at index", knowing the convention saves you time when trying to remember if it was insert_at_index, insertAtIndex or InsertAtIndex.

An exception to a single convention is using different conventions for different elements of your code. Classes should start with a capital letter, so you're more likely to use PascalCase for them, whereas with variables you may use camelCase, and your functions/methods use snake_case. What's important is that your naming conventions are obvious and consistent.

```
myString = "The quick brown fox jumped over the lazy dog"
newCap = 10

#snake_case
my_string = "The quick brown fox jumped over the lazy dog"
new_cap = 10

#PascalCase
MyString = "The quick brown fox jumped over the lazy dog"
NewCap = 10
```

Variable Naming

Your variable names should be descriptive of the element it represents concisely.

A good aim, though not a requirement, is between 10 and 16 characters. Not only should it be easy to write, it should be easy for anyone reading your code to understand what it is representing.

For example, total is short, but is not descriptive. Total what? Hours? Cash on hand? Bees? On the other hand, and yes this is dramatic, totalCashInAfterCashOut is way too long. Not only does it take a long time to write out, it has poor readability.

```
# too short, un-descriptive
total = 1000

# too long, hard to remember, read, and write
totalCashInAfterCashOut = 1000

# descriptive but concise
totalRevenue = 1000
```

Function Naming

Function names follow the same general guidelines as variable naming.

It should be descriptive enough that, when used in your code, it is clear what that line of code is doing; but concise enough that it is easy to write and read.

When working with a Class with private variables, you'll find that you'll need to 'get' information via class methods. Because of this, there will usually be a get method inside a class, maybe more than one. get() on its own is poorly-named as there is no indication as to what the method is getting. get_at_index() is a better example as it is short, but tells you exactly what it is doing, which is getting the value at a specified index.

All *required* methods in this course will be pre-named for you; however you are highly encouraged to create helper methods ([modularization](#)).

White Spaces

- When indexing or passing parameters into a function, don't add white spaces between your elements and the parentheses/brackets.
- On the other hand, when writing out equations, make sure to add spaces between your storing variable, equal sign, and the equation.

```
# Bad Practice
self.data[ 3 ]
self.set_at_index( 1, 1 )
z=x+y
x+=1
self.examples( x = x + 1, y = z)
```

```
# Good Practice
self.data[3]
self.set_at_index(1, 1)
z = x+y
x += 1
self.examples(x+1, z)
```

Descriptive Comments vs. Cluttering Comments

Every block of your code should have comments *briefly* describing the purpose of the variables, control flow statements, or calling function directly below. This **DRAMATICALLY** increases the readability of your code, and will save you time during your debugging process. It will also give your graders opportunities to give *helpful* feedback on your code.

With the above said, relying on comments alone can clutter your code and decrease its readability. If you find yourself needing to write a lot of comments to explain logic, you should consider if your variable names are contributing to any confusion.

```
def zab():
    # Descriptive comment describing variable assignment
    variableZero = 0
    variableOne = 1
    variableTwo = [foo()]

    # Descriptive comment describing function and loop
    bar(variableZero)
    for x in variableTwo:
        variableOne += faz(x)

    # Descriptive comment describing returning block
    variableThree = 3
    return foo(variableOne, variableThree)
```

```
# Bad Practice
```

```
while x != 0:  
    y += x  
    x -= 1
```

```
foo(z)
```

```
# Good Practice
```

```
# Stores the summation of the value x
```

```
while x != 0:  
    y += x  
    x -= 1
```

```
# Prints all the prime values from 0 to y
```

```
foo(y)
```

```
# Bad
```

```
# This variable is used to store a temporary value
```

```
x = z
```

```
# squares our temporary value
```

```
self.foo(x, y)
```

```
# Insert value x into index y
```

```
self.bar(x, y)
```

```
# Good
```

```
# Inserts an altered value into our array
```

```
tempValue = z
```

```
self.square_and_divide(tempValue, denominator)
```

```
self.insert_at_index(tempValue, idx)
```

Comment Alignment

Although it has already been briefly mentioned above, commenting can either make your code easy to read, or a headache and a half. For the most part, you should comment above blocks of code to briefly describe what it is doing. There may be occasions where leaving a comment right-aligned makes more sense, such as describing variable assignments at the start of a function, and when doing so Python allows you to multi-tab to align comments, making it easier to read.

```
# Bad Practice
```

```
for x in arr: #Iterates across a list of nums  
    sqr = x*x #squares our value  
    print(sqr) # Prints the square of x  
    tempArr.append(sqr) # adds squared value to sqrList
```

```
# Good Practice
```

```
# Prints and adds squared value of all items in arr
```

```
for x in arr:  
    sqr = x*x  
    print(sqr)  
    tempArr.append(sqr)
```

```
# Situational - use ONLY if the code and the comment are *short*
```

```
tempValue = x      # Value to be modified and returned
```

```
newCap = y         # Replaces capacity
```

```
total = z          # While counter for number of elements in array
```

Tips for Programming Assignments

This section covers some tips for successfully completing the assignments in this course.

General Tips

- **Start early**

This is probably the most important advice we can give. The assignments are not designed to be tricky, but they do take a fair amount of time to complete. We highly recommend starting as soon as possible.

Note, you are given 4 'free days' that you can use if you anticipate you will not be able to make a deadline. However, in order to take advantage of the late policy, you must email the instructor with the subject [CS-261 - Using Free Days for Assignment X] **before the assignment is due** and only two free days can be used on any given assignment. *Any assignment not submitted after the two day window will receive a score of 0.*

If you wait to start until right before the deadline, it may become difficult, stressful and sometimes impossible to finish the assignment in just a few hours. It is even more important to start early if you anticipate needing a TA's help during office hours. In prior quarters we experienced large spikes in office hour questions right before the submission deadlines. We try to help as much as we possibly can, but to be fair to everyone all questions are answered on a first-come, first-served basis. This means it may take a very long time, or sometimes even be impossible, to get to your question if asked right before the deadline.

In short, please start early. We truly believe that everyone who allocates sufficient time to the programming assignments can get 100% on all of them. Observations from prior courses confirm that.

- **Read and follow the specifications thoroughly**

Failure to follow them will result in the loss of points. You are being held accountable for reading these specs, so take whatever time is necessary to ensure that you understand them and have implemented all of the requirements. If you are unsure of what any specification means, be sure to ask questions as soon as possible on Ed Discussion or in Office Hours.

- **You can (and should) submit to Gradescope many times**

There is no limit to the number of times you can submit to Gradescope, and points earned for the assignment are not affected by how many times you submit. As a result, it is in your best interest to use it as often as is convenient to verify that you are on the right track.

Most assignments consist of several more-or-less independent subproblems. So once you solve a portion of the assignment, it may be a good idea to submit your solution to Gradescope to make sure it passes all tests. If it doesn't, it's probably best to return to the problem and iron out all your bugs before moving on. Pay close attention to the output of any failed tests. As will be discussed further below, it can be very valuable for debugging.

- **You can choose any submission from your Gradescope submission history for grading**

By default, Gradescope grades your most recent submission. If you want a different submission to be graded, you can click Submission History and Activate the submission you want graded. As we grade, we are not going to check everyone's submission history to pick the best one. It is your responsibility to choose the submission you want graded (if it is not the most recent submission).

- **No hidden Gradescope tests**

Unlike CS161 and CS162, in CS261 there are NO 'hidden' tests that are only available after the assignment deadline has passed. All tests are available to you from the moment the assignment opens on Canvas. If you pass all Gradescope tests, you should expect to get full points for the assignment (assuming, of course, that you commented your code well, did not use any prohibited methods / built-in Python data structures, met runtime complexity requirements, and wrote your own solution. These things are checked manually after the assignment deadline).

- **Code examples in the assignment PDF are part of the assignment requirements**

The assignment PDFs include short code examples for all methods you're asked to write. Please note that input / output from those PDF code examples IS CONSIDERED A PART of the assignment requirements. Once you have read the written description of each problem, please confirm your interpretation and understanding by comparing output from the examples to your expectations.

- **Take time to read through and understand all included code files**

The purpose of this class is for you to understand data structures from the ground up by building them yourselves. Python is an incredibly powerful language with a ton of 'batteries included'. For example, complex data structures that just work automagically. For that reason, in this class we will not be using Python's built in data structures for (almost) any assignments. Instead we have provided files (e.g. static_array.py for Assignment 1) which mimic the limited capabilities of data structure building blocks as they actually function beneath the surface. Again, these included data structures have intentionally reduced capabilities compared to the built-ins you are likely used to working with.

Before starting an assignment please take the time to read through the assignment files and understand what those capabilities are. For example, some common questions in Office Hours during early weeks are why loops using the common 'for item_x in array_y' fail or how to properly 'set' or 'get' a value in a static array. These questions are best answered by reading through the static_array.py file and understanding exactly how the code executes and why.

- **Use Gradescope output for debugging**

To help you get started, some common test cases are included in the skeleton code. You should definitely make sure your solution is passing them locally before submitting to the Gradescope, since this is the first thing it will check.

However, Gradescope will subject your solution to many more additional tests. Some of these tests are hardcoded, but there are many that will generate random input on each run. It is possible to pass all tests in the PDF but still fail some tests on Gradescope. If or when that happens, Gradescope will show you the input it ran, its expected output, and the output from your code. A great first step to find the errors causing you to fail is to take the input from the failing test and run it locally. You should get the same failing output. Once you've reproduced the error, fire up your debugger. Step through the code and watch carefully as your logic executes against this known faulty input. If a built-in Python exception is thrown, a good first step is often to look up the exception and any associated message in this document, on Stack Overflow, or in the official Python documentation.

- **Use the PyCharm interactive shell to test hypotheses**

PyCharm provides support for the Python shell. To use this tool, open your code in PyCharm, right-click, and select Run Program in Interactive Shell. This will run the code and open an interactive session where you can enter and execute lines of Python code. PyCharm allows you to click through currently active variables like you can in the debugger. The shell can be useful for quick real-time testing of data structures and their methods - you can create new data structure objects, call their methods with test data, and examine the data stored in these structures as a result of your methods.

- **Don't forget to comment your code**

This is specified in every assignment PDF, but please put informative comments into your code. If you don't, you may not get the full points for the assignment, even if you pass all Gradescope tests. Seriously, we may deduct points for failing to comment and **will** deduct points for no or poor docstrings in your code. No one likes to do this and it feels bad all around. Please include them.

- **A comment and a docstring are not the same thing**

A common early mistake to make (some of your TAs themselves made it initially) is to confuse comments and docstrings.

A comment in python begins with a hash-mark (#) and is interspersed with your code to document it. It only affects text following it on the same line.

A docstring begins with three quotation marks ("""") and affects all text until another three quotation marks ("""") are found. More importantly, docstrings contain an overview of the function's specifications. A function should only have one docstring located on the line immediately after it is defined. This may seem like a strange semantic difference, but as is explained by the Python documentation ([PEP 257](#) -

[Available Here](#)), the docstring actually becomes a special attribute of the function. When you type `help(name_of function)` in your REPL the help function returns this docstring.

Please observe and abide by this difference. Not only is it good Pythonic programming practice, but the use of excess docstrings instead of comments can cause Gradescope parsing errors that will be very difficult for you (or us) to debug.

Your Submission Timed Out

Your submission timed out. It took longer than 600 seconds to run.

Gradescope limits the length of time your submission can run to 10 minutes. If you see this error, it means something in your code is taking more than 10 minutes to run. Sometimes this happens because the tests we run on Gradescope have test cases with a very large amount of elements and your code is not efficient enough. Often this is happening because your code has an infinite loop.

“But I can pass all of the tests locally without an infinite loop!” you may say. To which I reply: You may be able to pass the basic tests included in the assignment files without an infinite loop, but that is because those tests are basic and do not necessarily account for all possible edge cases. The infinite loop is being triggered by an edge case your code does not account for.

Regardless of the cause, if you experience this error, good practice is to first try and isolate the code that produces the loop or extensive runtime. A good way to do this is by incrementally commenting out portions of your code and resubmitting it to Gradescope, until it stops timing out. Once that happens, you’ll know your most recently commented-out block of code is the one causing the issue and you can focus on debugging it to discover what is wrong.

The Autograder Failed to Execute Correctly

The autograder failed to execute correctly. Please ensure that your submission is valid. Contact your course staff for help in debugging this issue. Make sure to include a link to this page so that they can help you most effectively.

This error means that your program exceeded Gradescope’s memory limit while it tested your program. In almost all cases, this is due to convoluted code (for example, excessive if/else blocks with many nested conditions) that would benefit from a clean up in logic. In rarer cases, it is actually a timeout error as described above, and the program exceeds the memory limit before Gradescope’s 10 minute testing limit has passed.

As described previously, you can incrementally comment out portions of your code until you identify the method that is causing issues. Then you can consider refactoring it or starting from scratch.

Error Handling

Terminal Errors

So, you're working on an assignment and think you have everything ironed out and run your code only to be met with an error in your terminal. This can feel super frustrating but these errors provide a lot of valuable information for you to use!

```
Traceback (most recent call last):
  File "[REDACTED]", line 354, in <module>
    print(example_function(my_list))
  File "[REDACTED]", line 343, in example_function
    item = a_list[list_item+1]
IndexError: list index out of range
```

This error is letting me know that some part of the code is causing an index error and that the error is being caused when line 343 is being executed. This is all valuable information to be used in your debugging. If you experience an error and can not make heads or tails of what it means, copy the error directly from your terminal and search a site like [Stack Overflow](#). If you're still struggling, please feel free to make a private Ed Discussion post or ask in office hours. Having done all of the previous steps will help us help you quicker.

Common Errors

See the Python Docs for more details on all exceptions:

<https://docs.python.org/3/library/exceptions.html>

AttributeError:

An "Attribute Error" happens when you attempt to use a specific property of an object or variable that isn't available for that particular object.

```
>>> a = "string"
>>> a.length
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'length'
```

This frequently occurs with a "NoneType" object, indicating that the object you are accessing a property of is set to None. To resolve this, it's typically best practice to use a conditional statement to check if the object is None before accessing the property.

```
>>> a = None
>>> a.value
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'value'
```

BST - Find Min - Test 2 (0/5)

```
Description : BST find_min: Testing with random values
Input       : BST pre-order {  }
            : BST post-order {  }
            : Operation - find_min()
Expected    : Return: None
Student     : Return:
find_min()  crashed with error 'NoneType' object has no attribute 'left'

Test Failed: False is not true
```

TypeError:

A "TypeError" occurs when you attempt an operation that is not supported for a particular object.

```
>>> a = []
>>> b = 1
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
```

IndexError:

An "IndexError" occurs when you attempt to access an element in a list at an index that does not exist. A good way to avoid this is to ensure that the index is always less than the length of the list.

```
>>> a = [1,2,3]
>>> a[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

If this happens on Gradescope it may output "Crashed with error: Index out of bounds":

10 - Sorted Squares - Test 2 (0/7.5)

```
Description : Testing with an array of random integers
Input       : STAT_ARR Size: 2 [-824825810, -92527310]
Expected    : STAT_ARR Size: 2 [8561303095836100, 680337616842156100]
Student      : None
Crashed with error: Index out of bounds

Test Failed: False is not true
```

NameError:

A "NameError" takes place when a local or global name is not found. This usually happens due to a typo or when the variable was never defined within the current scope.

```
>>> print(wrong_name)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'wrong_name' is not defined
```

ModuleNotFoundError

A "ModuleNotFoundError" occurs when you try to import a file that doesn't exist in the local directory. For your assignments, ensure you upload the required files to Gradescope with the correct file names and the correct names for classes, methods, and functions as specified in the skeleton code.

If you encounter this error when running your code locally, make sure that all filenames are correct and that they are located in the proper local directory.

If you receive this error on Gradescope, double-check that you have uploaded the necessary files as specified for the assignment.

```
>>> import dynamic_array
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'dynamic_array'
```

test_utilities_queue_sa (unittest.loader.FailedTest)

```
Test Failed: Failed to import test module: test_utilities_queue_sa
Traceback (most recent call last):
  File "/usr/lib/python3.10/unittest/loader.py", line 436, in _find_test_path
    module = self._get_module_from_name(name)
  File "/usr/lib/python3.10/unittest/loader.py", line 377, in _get_module_from_name
    __import__(name)
  File "/autograder/source/tests/test_utilities_queue_sa.py", line 27, in <module>
    from test_5_sa_queue import Queue, QueueException
  File "/autograder/source/tests/test_5_sa_queue.py", line 16, in <module>
    from queue_sa import Queue, QueueException
ModuleNotFoundError: No module named 'queue_sa'
```

Expected Exception

This one is specific to Gradescope. On Gradescope your code may go through a series of tests to see if an exception is raised if specific conditions are met. If your code fails to raise the exception, Gradescope will display an output stating that it was expecting the specific exception to be raised.

In this example the Gradescope output displays for the student "Expected SLLError for invalid index". This means that an SLLError should be raised, but it was not raised.

SLL - Slice - Test 2 (0/4.5)

```
Description : SLL slice: This is a test with random values
Input       : SLL [40201 -> -27178 -> -60646 -> 51076 -> -41094 -> 73432 -> -79969 -> 88289 -> 42748] Slice index 9 size 0
Expected    : SLLError
Student     : Expected SLLError for invalid index

Test Failed: False is not true
```

Feedback on Gradescope

When you do not pass a test on Gradescope, you will receive feedback on why the test failed.

```
Description : Testing with array of random integers
Input       : STAT_ARR Size: 4 [89085, 42456, 82660, -16159]
Expected    : (-16159, 89085)
Student     : None
Crashed with error: Index out of bounds

Test Failed: False is not true
```

Ah, index out of bounds error, welcome back.

There is always going to be valuable feedback from the failed tests. You can see the input that was used to run the test, the expected results, and what your program produced.

A good way to use this information is to take the provided input and make your own test locally. This may not *always* work because some tests build on previous tests (testing consecutive deletions of an element, for instance), but it is a good first step.

Miscellaneous tips

If you include print statements in your code and upload the file to Gradescope you can see the result of those print statements in your results. This can be used to help you in your debugging for some assignments. Just remember to delete the print statements prior to your final submission!