Python Workshop B23

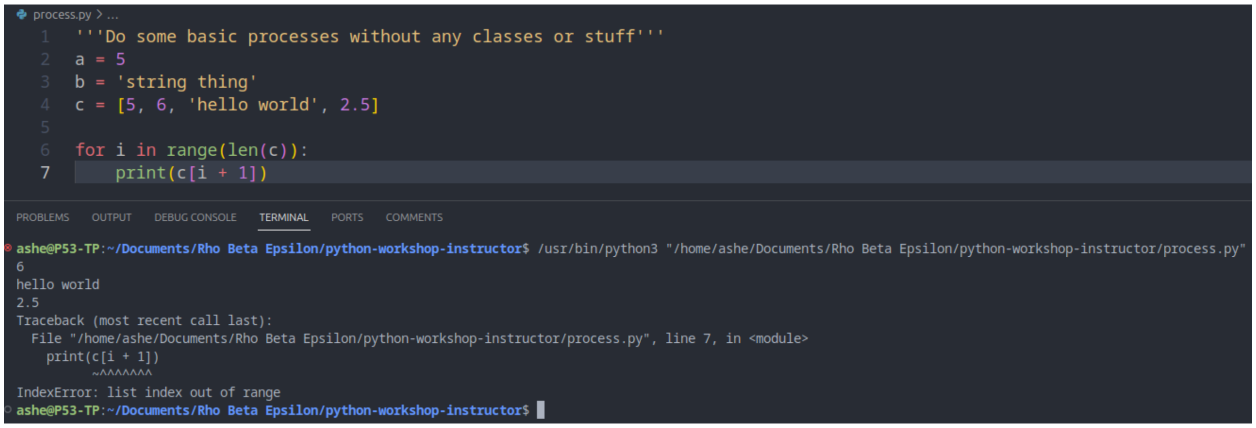
This workshop is intended to be useful to students in either RBE 100X or RBE 3002. There are several sections, some of which cover more basic skills, while others focus on more advanced, focus skills specific to RBE 3002. Feel free to go through the sections that interest you and skip the ones that don't.

Section 1: How to Debug Python

1. Understand Python Exceptions

Python gives relatively descriptive error messages when a script doesn't work. It will tell you the error type, such as TypeError or IndexError, and a blurb about what the error actually means. It will also tell you the line of code where the error occurred.

Below is an example of a script where an IndexError occurs because the for loop tries to access an element outside the given array. Because Python is an interpreted language, it is executed line by line until it is either done or has an error. The last three items of the list are printed successfully, but trying to go beyond the items in the list causes an error.



```
1  '''Do some basic processes without any classes or stuff'''
2  a = 5
3  b = 'string thing'
4  c = [5, 6, 'hello world', 2.5]
5
6  for i in range(len(c)):
7      print(c[i + 1])
```

```
ashe@P53-TP:~/Documents/Rho Beta Epsilon/python-workshop-instructor$ /usr/bin/python3 "/home/ashe/Documents/Rho Beta Epsilon/python-workshop-instructor/process.py"
6
hello world
2.5
Traceback (most recent call last):
  File "/home/ashe/Documents/Rho Beta Epsilon/python-workshop-instructor/process.py", line 7, in <module>
    print(c[i + 1])
          ~^^^^^^^
IndexError: list index out of range
ashe@P53-TP:~/Documents/Rho Beta Epsilon/python-workshop-instructor$
```

2. How to find why your code doesn't work
   a. Print debugging

This method of debugging is fairly straightforward to implement: add a print statement at a given point in your code to verify that the code at that point is running and working.

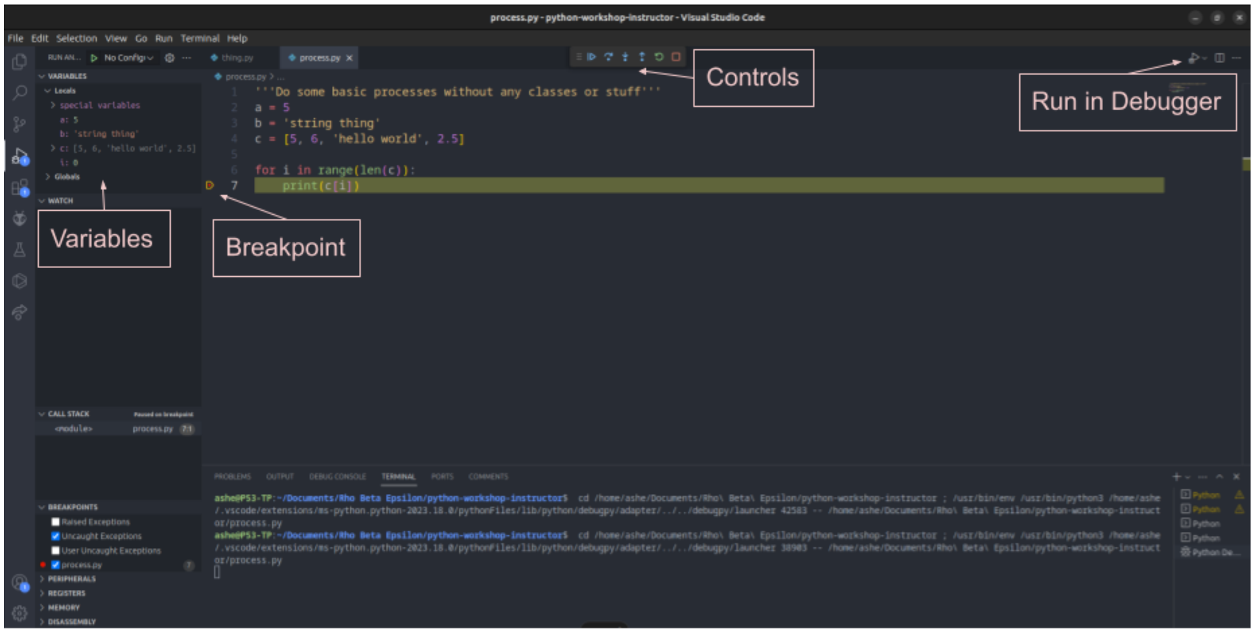
```
a = 5
b = 17

print(b/2)
```

```python
if a == b/2:
    print("a is equal to half of b")
else:
    print("a is either greater or less than half of b")
```

b. The debugger

This is useful when you want to monitor a variable's value throughout a process but don't want to clutter your code with print statements. This method also lets you see which code is being executed. Using the debugger changes slightly between editors and IDEs, but generally you pick a line of code to put a breakpoint, where your code switches from running normally to running in debug mode. You then run the script in debug mode. In debug mode, you can continue through code, step over it, step into it, or step out of it. Each action is explained by [1]. These actions let you navigate through the code around the breakpoint. Additionally, in a side menu, you can see current variables and their values. Terminal outputs appear in the terminal as they happen.


process.py - python-workshop-instructor - Visual Studio Code
File Edit Selection View Go Run Terminal Help
RUN AN... No Config
thing.py
process.py
Controls
Run in Debugger
Variables
Breakpoint
VARIABLES
Locals
WATCH
CALL STACK
BREAKPOINTS
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
special variables
a: 5
b: 'string thing'
c: [5, 6, 'hello world', 2.5]
i: 0
Globals
```

```python
'''Do some basic processes without any classes or stuff'''
a = 5
b = 'string thing'
c = [5, 6, 'hello world', 2.5]

for i in range(len(c)):
    print(c[i])
```

```
ashe@PS3-TP:~/Documents/Rho Beta Epsilon/python-workshop-instructor$  cd /home/ashe/Documents/Rho\ Beta\ Epsilon/python-workshop-instructor ; /usr/bin/env /usr/bin/python3 /home/ashe/.vscode/extensions/ms-python.python-2023.18.0/pythonFiles/lib/python/debugpy/adapter/../../debugpy/launcher 42583 -- /home/ashe/Documents/Rho\ Beta\ Epsilon/python-workshop-instructor/process.py
ashe@PS3-TP:~/Documents/Rho Beta Epsilon/python-workshop-instructor$  cd /home/ashe/Documents/Rho\ Beta\ Epsilon/python-workshop-instructor ; /usr/bin/env /usr/bin/python3 /home/ashe/.vscode/extensions/ms-python.python-2023.18.0/pythonFiles/lib/python/debugpy/adapter/../../debugpy/launcher 38903 -- /home/ashe/Documents/Rho\ Beta\ Epsilon/python-workshop-instructor/process.py
```

3. Common pitfalls

- Don't use ; or {} to end lines or contain functions and loops
    - This is Java/ C/ C++ syntax
- Whitespace matters
    - Particularly indents (1 tab or 4 spaces per level of indent), which indicates what code is inside an if statement/ loop/ function
- Make sure to use self when using a class attribute within a class function

- Don’t try to modify the number of elements in a list and access its elements at the same time

Section 2: Classes in Python

1. Implement a class

Python features classes for object-oriented programming. This allows a programmer to group functionality so that code is compartmentalized, encapsulated, and organized. The syntax for implementing a class is brief.

```
class Example: # name of the class
    def __init__(self):
    # this is the initialization function
    # your class must always have at least one of these
    # the self argument is for setting up the primary functionality
of classes - being able to access internal methods and attributes
        pass # this example function doesn't set up anything
```

Classes can be set up in individual Python files and then imported to the main script. Alternatively, if you only have one class, you can include the class definition in the same file as your script. To separate the class from the script, we use a specific if statement: `if __name__ == “__main__”:` and put all script code within this if statement. Class definition code is before the if statement. This is so that if the script is run directly (hence ___name___ is equal to “___main___”), the script code is run. Otherwise, if the file is imported or run indirectly, the script code is not run, but the class is still available for use.

```
class Example:
    def __init__(self, name):
        self.name = name

    def print_name(self):
        print(self.name)

if __name__ == "__main__":
    my_example = Example("Python")
    my_example.print_name()
```

2. Class attributes

These are variables associated with the class. These variables can be accessed in functions within the class and in scripts that use an instance of the class. To define a class attribute, you include them in the `__init__()` function. Within `__init__`, use the self notation with variables you want defined for the class. These can be variables that are set by input arguments and/ or hardcoded values.

```
class Example:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.height = 6

    def display_height(self):
        print(self.height)
```

Within the Example class, these variables can be called like other variables, using self in the name. In a script that uses an instance of the Example class, these attributes can be accessed using . notation.

```
my_example = Example("Python", 20)
print(my_example.name)
```

3. Implement a class method
    a. Non-static

A non-static method in a class typically relies on an attribute or another method of the class in order to function, though it doesn't have to. These methods include self as the first argument, which enables them to access other things in the class. To use a non-static method in a script, you must create an instance of the class.

```
class Example:
    def __init__(self, name):
        self.name = name

    def display_name(self):
        print("My name is " + self.name)
```

    b. Static

A static method in a class does not rely on any attributes of the class it's in and can be called in a script without creating an instance of the class. These methods are marked with the @staticmethod decorator just above their definition. Notice how self isn't included in the arguments of the function. If you need to access an attribute or another method of the class, you can't use a static method.

```
class Example:
    def __init__(self):
        pass

    @staticmethod
    def add_two_numbers(a, b):
        return a + b
```

Section 3: Exceptions in Python

1. Why you should handle them

Many times, an exception indicates that your code is broken somehow and needs to be fixed. However, sometimes an exception can be used as an indicator that a condition is or isn't met. Rather than have your code fail and stop running, you can handle the exception so that your code does something when you run into one. For example, if you try to write data to a file, but the file doesn't exist, you can handle the resulting exception by automatically creating a new file and writing the data there. This allows your code to be more automated.

2. How to handle one

To handle exceptions in Python, use try and except blocks. In the try block, put the code you'd like to run, assuming everything works. In the except block, program what your code should do in the event something doesn't work.

```
try:
    print(something)
    print("See, I can code")
except NameError as e:
    print(e)
    print("You silly goose!")
```

It is best to be as specific as possible with the exception you're planning to catch. "Exception" can be used for any exception, but maybe you want to change your error handling based on what went wrong. You can also catch multiple kinds of errors with just one except statement.

```
try:
    A = 2**3*
    print(something)
```

```
except (NameError, SyntaxError) as e:
    print("You goofed your variables or your syntax")
except Exception as e:
    print("Oh boy")
    print(e)
```

Section 4: Priority Queue

1. What is a priority queue

A priority queue is a type of queue that sorts items based on an associated priority value. Typically the highest-priority items are removed first from the queue, though the queue can be implemented that the opposite is true [2].

In the case of RBE 3002, a priority queue is useful for implementing the A* search algorithm. This allows you to keep track of which node you should explore next based on their priority: the priority of the node can be determined using your A* heuristic.

2. How to implement one

Using native Python, you can use a list of tuples to implement a priority queue. This works for smaller queues with fewer insertions and can be easier to think about. However, it is not the most efficient method, so larger cases may be slower [3].

```
nodes = [] # list of tuples where the first tuple value is the
priority (bigger number means higher priority
node.append((4, "A"))
node.append((3, "B")) # not sorting here because it's lower priority
than first item
node.append((5, "C"))
node.sort(reverse = True) # sort list in reverse order because higher
priority item was added

while nodes: # while queue not empty
    print(node.pop(0)) # prints C, A, B
```

A more efficient way is to use the heap data structure. Python's `heapq` library allows us to use a minimum heap where higher priority items have lower value [3].

```
import heapq

nodes = [] # list of tuples where the first tuple value is the
```

```
priority (bigger number means higher priority
heapq.heappush(nodes, (4, "A"))
heapq.heappush(nodes, (3, "B"))
heapq.heappush(nodes, (5, "C"))

while nodes: # while queue not empty
     print(heapq.heappop(q)) # prints B, A, C
```

Section 5: State Machines in Python

1. What is a state machine

A state machine is an abstraction for algorithm design where the machine reads inputs and changes between states based on those inputs [4]. In robotics, we use state machines to dictate robot behavior based on certain conditions. For example, we may start in an idle state, where the robot does nothing. Then, if a certain sensor reads a certain value (e.g. distance sensor reads above 10 cm), we may change to a driving state where the robot moves forwards. In the case of a distance sensor reading values, we may stop driving once the sensor reads less than 10 cm. While this and other functions can be done sequentially, a state machine is often more organized, particularly for complex tasks and systems, and a state machine allows for greater automation.

2. How to implement one (completely native Python)

In C++ you can use `enum` to denote your different states, and a `switch` statement to define each state and move between them. Python also has `switch` statements, though we will have to be creative about replacing enums, as those are not native to Python [5].

```
from random import randint

my_states = ["IDLE", "DRIVE"]
current_state = my_states[0]

while True:
    sensor_reading = randint(0, 9)
    match current_state: # use match instead of switch syntax
        case "IDLE":
            if sensor_reading > 2:
                current_state = my_states[1]

        case "DRIVE":
            drive_robot() # pretend we defined this

            if sensor_reading > 2:
                current_state = my_states[0]

        case _: #default case
            break
```
To use enums, you can import the enum library [6].

Section 6: Python and ROS

1. Basic setup

To use ROS in Python, you must have ROS installed on your computer. From there, you can import `rospy` into your Python script, which allows you to create a ROS node. For working with messages, import the classes corresponding to the messages you wish to work with. You may have to do some documentation surfing to figure out where your message type comes from.

```
import rospy
from geometry_msgs.msg import Twist # example of importing a ROS
message
# geometry_msgs.msg, nav_msgs.msg, and std_msgs.msg have most of the
ROS classes you'll need for working with messages

class Workshop:
    def __init__(self):
        # Initialize the node and give it a unique name
        rospy.init_node("workshop_node")

        # Sleep to allow roscore to do some housekeeping
        rospy.sleep(1.0)

    def run(self):
        rospy.spin() # keeps ROS node running until you hit Ctrl-C

if __name__ == "__main__":
     Workshop().run() # initialize Workshop object and then run the
node it creates
```

2. How to use objects from ROS

a. Publishers and subscribers

To create a publisher, we create an instance of rospy's Publisher class and set it to a variable so we can access it later for publishing messages. When creating a Publisher, we must tell ROS to which topic we're publishing, the type of message being published, and how large the queue of the Publisher is. Publishers should be defined in the ___init___ function of our Workshop class so that other methods can access it.

```
self.twist_pub = rospy.Publisher("some_topic", Twist, queue_size = 1)
```

Subscribers are easier to set up because they do not need to be accessed later–we just tell ROS our node will be subscribing to a given topic. We then use rospy's Subscriber class and tell ROS the topic to which we're subscribing, the specific message type we're listening for, and the function we'll run when a message of the correct type is published to the topic. This is again defined in the ___init___ function of the Workshop class.

```
rospy.Subscriber("odom", Odometry, self.update_odometry)
```

Notice that we do not pass any arguments to the function that will be called when a message is published. This is because the message will automatically be passed to that function. The function definition, however, should have an argument for the message it will be receiving.

```
def update_odometry(self, twist_msg):
```

b. Create and build a message

Let's say we want to publish a Twist message (linear and angular velocity). To do so, we first create an instance of the Twist message class with `my_message = Twist()`. We can then set the individual attributes of the message the same way we'd set class attributes from a script. For example, to set the linear x velocity of our new message, we would write `my_messsage.linear.x = 5.0`. To figure out how your message attributes are broken down, check the ROS documentation for that message. Once the necessary attributes are set, we can publish the message with our already defined publisher and the `publish()` function: `self.twist_pub.publish(my_message)`.

Bonus: Type Notation in Python

Python does not prescribe to strict typing, which means in basic Python syntax you can't tell what types of values a function takes in and puts out.

However, Python does support annotating functions to indicate the types of their arguments and return values. Below is an example.

```
def task_one(self, done: bool):
    if done:
        return
    else:
        print("Working on task...")

def task_two(self) -> int:
    print("Working on task 2...")

    a = randint(5, 10) # random integer between 5 and 10 -->
uses the random library
    b = randint(2, 7)

    if a == b:
        return 5
    elif a > b:
        return 4
    else:
        return 6
```

In the function task_one(), we see that the function argument *done* is intended to be a boolean because of the bool annotation next to the argument in the function definition. We can see that the function task_two() should return an integer because of the int annotation after the function definition.

Common type annotations are listed below. You can also use custom classes in your annotations. For example, if you've defined a class Thing, or if a library you're using defines a class Point, you can use those class names in your annotations.

- int → Integer
- float → Float
- bool → Boolean
- str → String
- tuple → Tuple
    - tuple(int, int) → List of tuple of two integers

- list → List
  - list(int) → List of integers
  - list(tuple(int, int)) → List of tuples of integers

Bonus: Numpy and OpenCV

References

[1] https://code.visualstudio.com/docs/editor/debugging#_debug-actions
[2] https://www.geeksforgeeks.org/priority-queue-set-1-introduction/
[3] https://builtin.com/data-science/priority-queues-in-python
[4] https://developer.mozilla.org/en-US/docs/Glossary/State_machine
[5]
https://www.freecodecamp.org/news/python-switch-statement-switch-case-example/
[6] https://www.geeksforgeeks.org/enum-in-python/#