**Lecture 13: Argparse, defensive programming, unpacking**

Today, we’ll cover some advanced python concepts. First, we’ll look at ways to pass in arguments to your python program from the command line (using **argparse** and sys), then, we'll cover **mypy,**  which is a library that allows you to check for errors in your program before shipping it ( mypy works in conjunction with type hints). Finally, we will look at unpacking with (\* and \*\*) which allows you to create flexible functions/methods that can accept a variable number of arguments (like the built-in print function).

**Argparse and sys**:

In this section, we’ll create a program that collects user information and logs it on file. Initially, we’ll use sys.argv, and then we’ll move on to argparse which is more powerful.

* **Sys:**

sys allows you to access command line arguments in your python module.

* 1. Open VS code and create a new file called **student\_info.py**.
  2. Import sys
  3. You can access the commandline arguments using sys.argv, which is a list
  4. Sys.argv[0] is always the name of the python module that got run from the commandline
  5. You can check for the number of arguments entered in the commandline using len(sys.argv)
  6. Task: Define a function called logrow(filename, data), that logs the data to file, with the current timestamp (import time and use time.time() and time.ctime())
  7. Task: Print all arguments, one per line
  8. Task: Print the number of arguments
  9. Task: create a program that accepts an argument named ‘-n’ for the number of students to get information about and save this information using the logrow() function defined in 7 above (use a default storage file). If no argument is entered, default to getting information from 1 student.
* **Argparse:** [**https://docs.python.org/3/library/argparse.html**](https://docs.python.org/3/library/argparse.html)

Argparse builds on sys and handles most of the work for you. It comes with help messages and type checking. We will explore this functionality and extend the previous section so that we get the filename from the user.

1. Import argparse
2. Comment out the code from the previous sys (leave the logrow() function uncommented)
3. Create a new parser using parser = argparse.ArgumentParser(). Feel free to add the description or usage as arguments to this function.
4. Add an argument using parser.add\_Argument(‘-n’, ‘--numtimes’, default = 1, type = int, help = “help message for the argument”)
   1. One letter flags are preceeded by a single –
   2. Named flags are preceded by double --
5. Task: add an argument for the file name designated -f, or --filename, make it optional. use required = False and add a help message
6. Parse the arguments into a variable called args with args = parser.parse\_args().
7. On the command line, run **python** **student\_info.py -h** or **python student\_info.py --help** to see a help message about your program and its usage**.**
8. Task: toggle whether an argument is required or and see how it affects the help message.
9. Task: access the arguments using args.n and args.f and use them to implement the functionality from the previous section. Note: if you specify both a single letter and full name for a parameter, use the full name to access it (e..g. args.filename if you had both -f and --filename)

**Mypy and defensive programming:**

Python is a dynamically typed programming language, which means that variables can take on different types of data. Strongly typed programs (eg. C,C++,ava etc) require you to specify the types of variables. During compilation, these programs will catch type errors. We can use type hints in python: <https://docs.python.org/3/library/typing.html> in conjunction with mypy to “catch” these errors in python. Mypy checks whether your program is adhering to these type hints. Documentation at: <https://mypy.readthedocs.io/en/stable/>

1. Install mypy in your conda environment using **conda install mypy**
2. Create a new python file named **dawgs.py**
3. Define a function named woof(n) that prints ‘woof’ n times based on user input.
4. Use type hints in your function to designate n as an int
5. On your terminal, run mypy dawgs.py to see if you have any type errors
6. Add type hints to where you assign the user entered variable e.g number:int = input(“woof how many times”)
7. Task: ensure that mypy gives your program a clean bill of health
8. Run your program on the commandline : python dawgs.py
9. Add a type hint for the return value of woof(n:int)->None: and try and use it in your file e.g. by printing it and run mypy on your file to catch this error
10. Fix the woof(n:int)->None: function so that it returns a string that then is printed. Use typehints for all the variables

**Unpacking:**

Previously in the course, we looked at ways to unpack tuples and lists into variables. With e.g my\_tuple = (1,2,3) followed by x,y,z = my\_tuple.

We’ll create a currency converter that takes in a wallet with some currencies, e.g. KES, GPB, EUR, YEN, YUAN, INR and gives us the total in USD. Then we’ll look at how to create flexible functions with positional arguments (\*args) and keyword arguments (\*\*kwargs)

* **\* and \*\*:**

1. Create a new file called **unpacking.py**
2. Create a function called Converter(KES, GPB, EUR, YEN, YUAN, INR): that takes the different currencies in a wallet and returns the total in USD.
3. Create a wallet list variable that contains specific amounts of the different currencies arranged positionally. E.g wallet = [1000, 500, 80,150,200,8000]
4. Task: implement the logic that calls the Converter function based on the wallet list and prints the total USD
5. Task: Use list unpacking to improve your implementation: Converter(\*wallet)
6. Task: call the converter function using named parameters: Conveter (KES = , GPB = , …)
7. Task: change the wallet into a dictionary and use it to call the Conveter () function
8. Task: Use dictionary unpacking to improve your implementation using Converter(\*\*wallet)

* **\*args and \*\*kwargs:**
  1. Create a function that takes an arbitrary number of positional arguments (\*args) and keyword arguments (\*\*kwargs) called GenericFunction(\*args, \*\*kwargs):
  2. Task: within the body of the function, print the positional arguments using args
  3. Task: within the body of the function, print the keyword arguments using kwargs
  4. Call the function with different values for the positional and keyword arguments and stress test it.