Hello everyone and welcome to our Medical Diagnosis Bot course! In this course, we will explore **how to design and construct algorithms** by building a simple dehydration diagnosis bot using the Python programming language. We'll base this process on the **4 stages of project management**: definition, design, implementation, and evaluation. At each stage, we'll discuss why that stage is important in any project implementation and fulfill that part of our medical diagnosis bot project.

Learning to design and implement algorithms is important not just in the world of programming but in all areas of life. An **algorithm** is simply an outline of the steps taken to arrive at a solution. We typically associate algorithms with coding but algorithms exist anywhere we use a series of steps taken to solve a problem or complete a task. We can apply the 4 stages of project management to plan, implement, and ultimately, carry out a solution. This doesn't just help us to solve coding challenges; it helps us to be better thinkers and problem solvers in general.

Algorithms are used throughout the medical field to diagnose patients, analyze data, test results, and images, run machines, and more. We'll focus on the diagnosis side of things and will implement a dehydration diagnosis bot from scratch to learn how to write algorithms with a relatable example. The diagnosis algorithm is simple enough and will allow us to plan and implement an algorithm without compromising the overall project management lessons. At the end of it all, you'll have gained problem-solving skills, project management skills, and will have a neat little Python project for your portfolios! That's our course in a nutshell so let's jump right into the lessons!

The **project definition** stage is where we define the goals for our project. We can start by describing the problem(s) that we're trying to solve or the task(s) that we're trying to accomplish, being as descriptive as possible. This is also a good place to mention any **preconditions and criteria for failure/success**. Although not necessary, a good way to outline these goals is using statements like this:

"We want to <perform a task>, given <some preconditions>"

Note that we don't need to specify how we'll go about solving these problems or which tools we'll use; those are defined in the next section.

For our project, we have **2 main goals**: to diagnose a patient's state of dehydration and to keep track of and print all diagnoses. We can phrase them in above format like this:

We want to diagnose a patient's state of dehydration based on a short questionnaire given:

1. The questions are all binary questions (only 1 of 2 possible choices)
2. The response to a given question will determine which question is asked next/which diagnosis to return
3. The final dehydration diagnoses are either no dehydration, some dehydration, or severe dehydration

We want to print out a list of all saved patients and their diagnoses given:

1. We are storing a list of all previous patients and diagnoses
2. The list is accessible to us
3. We can add new diagnoses to the list

Now that we've outlined the goals for our project, we can begin planning how we'll go about achieving them. The **project design stage** is where we outline how the project will work in terms of which tools and systems we'll use and how we'll structure the solution. For us, this means figuring out which programming language to use, how we want the program to run (text-based, GUI, etc.), how we want to structure the code, and which data structures we want to use. Tools such as **UML diagrams** and **flowcharts** are particularly helpful at this stage. Keep in mind that we may change our requirements, goals, and design as we implement the project and even after the project is done so nothing here is set in stone.

Let's apply this planning to our project. We know that we have 2 goals:

1. To run and store a new diagnosis
2. To display a list of previous diagnoses

Although these share some components, they are **two separate functionalities** so we'll need at least one function for each and likely some helper functions, especially when performing a new diagnosis. We'll also need a list to store patients and diagnoses; to keep things simple, we'll just use a list of strings formatted as:

"Patient_name – diagnosis"

We'll need to access this list from the **diagnosis function** in order to store new patients and diagnoses as well as from the print-diagnoses function. We'll dive deeper into the specifics of code implementation in the upcoming tutorials.

Now for the more technical specifications. We'll use Python for the programming language but we're not going to bother with a GUI or building widgets. Our program will be purely text-based and will run entirely within the **Terminal/CMD Prompt**. Rather than providing the user with interactive widgets such as buttons or sliders, our program will repeat these two steps continuously:

1. Display a prompt or ask the user a question
2. Process the user's text response and move to the next step

The next step is determined by **how the user responds** to the current prompt/question and the program only ends when the user enters a "quit" command or manually stops running the program.

We don't have any complex data structures so there's no need for UML diagrams but this decision tree serves as the basis for the diagnosis algorithm:

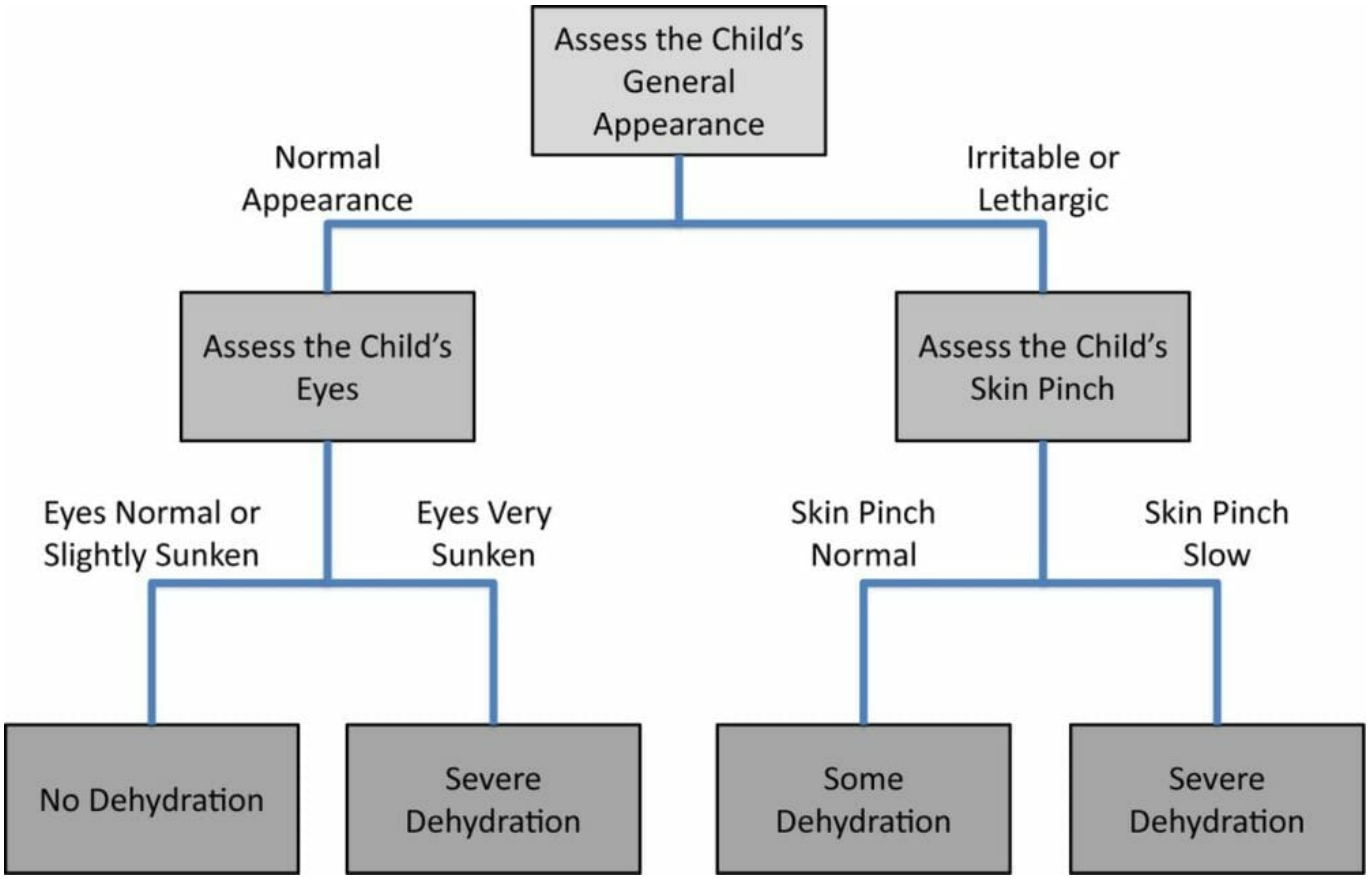


Image Credits: https://www.ghspjournal.org/content/3/3/405

Empirically Derived Dehydration Scoring and Decision Tree Models for Children With Diarrhea: Assessment and Internal Validation in a Prospective Cohort Study in Dhaka, Bangladesh
Adam C Levine, Justin Glavis-Bloom, Payal Modi, Sabiha Nasrin, Soham Rege, Chieh Chu, Christopher H Schmid and Nur H Alam

Global Health: Science and Practice September 2015, 3(3):405-418; https://doi.org/10.9745/GHSP-D-15-00097

This diagram outlines **3 sets of logic** that we'll need to implement:

1. Is the patient irritable/lethargic or do they have a normal appearance?
2. Are the patient's eyes normal/slightly sunken or are they very sunken?
3. Is the patient's skin pinch normal or slow?

Note that for any given diagnosis, there will only be 2 questions asked as the results of the first question determine which branch to take and the second question will output a diagnosis. We'll break this down into smaller steps and go over flow diagrams where appropriate.

Now that we have a rough plan of attack for our project, it's time to start **implementing a solution**. This is the stage where we actually put the plan into action which for us, means writing the actual project code.

Text Input and Output

Before we begin implementing the solution, we need to know how to use the chosen technology. We're assuming that you know some Python but if you didn't, now would be a good time to learn the basics. As our project is based entirely around the Terminal/CMD Prompt, we don't need to learn how to use a GUI or widgets but we do need to know how to **print to the Terminal/CMD Prompt and capture user text input**. We can do so through the use of the **print()** and **input()** functions, respectively.

Let's start a new project so we can write and run some code. Feel free to use any Python environment you desire, as long as it has a text editor to write code and a Terminal/CMD Prompt environment to run the code. I personally recommend **Replit** for tutorials such as this as Replit provides an environment that runs in the browser and requires no prior installation or setup. You can simply create a free account, create a new project, and start writing and running code immediately. The website can be found here:

https://replit.com/

Regardless of which environment you choose, we recommend calling your project something like "MedicalDiagnosisBot" and creating a file called "**main.py**". This will house all of your project code. Once this is done, do a test run to make sure everything is set up. Navigate to your **project folder** through Terminal/CMD Prompt and enter the run command or in Replit, simply press the green Run button at the top of the screen. This will run any code in main.py (which will do for now as we have no code there yet) so repeat this process to run our project code in the future.

Now that our project is set up, we can learn to use the print() function. This is the easiest way to output text to the Terminal/CMD Prompt, often referred to as the console. We simply put whatever we want to print **within the parentheses** like so:

```
print("Welcome doctor, what would you like to do today?")
```

If we run that code, we should see the message: "Welcome doctor, what would you like to do today?" appear in the console. We can put whatever **variable or literal value** we want in a print() function and it will either print the value or, if we put an **object** into the print statement, it will print a **brief description and memory address**. Go ahead and try to print a number, or a True/False value and see what output you get. We can also print **multiple items** separated with commas like so:

```
print("Welcome doctor.", "What would you like to do today?")
```

All items within the parentheses are printed next to each other but each new call to print() puts the value(s) on a new line.

To handle user text input, we can use the input() function. This works by **displaying a prompt, pausing the program's execution until the user enters some text in the console and presses enter, and then returning said text**. For example, If we wanted to get the user to enter their name, we could do something like this:


```
name = input("What is your name?")
```

If we run that code, the program will output the text: “What is your name?”, wait for the user to type out their name and press enter, and then store the text in the variable “name”. The resulting text is **always stored as a string**. Note that there are no spaces or newlines between the prompt and the user’s input. We can fix that by adding a space or the newline character, “\n”, like so:

```
name = input("What is your name?\n")
```

This is a **special character** that doesn’t appear in the actual string but forces the caret onto a new line. We’ll make use of this technique along with input() functions a lot in our program. Try to prompt the user to answer another question such as the first one of the diagnosis algorithm and when you’re ready to move on, we can begin writing the actual project code!

Live Coding – Project Implementation – Part 1

Program Entry Point

We want some sort of a start function that gets the program running and chooses which part of the code to execute. We can call this something like main() or start() or run_program(); I'll call mine main(). This is the **entry point** and simply provides a prompt to the user, handles their selection, and directs them to the next part of the program. Rather than storing all of the string prompts in the functions themselves, let's create them outside of the functions and store them as variables. Our welcome prompt could be something like this:

```
welcome_prompt = "Welcome doctor, what would you like to do today?\n - To list all pa
tients, press 1\n - To run a new diagnosis, press 2\n - To quit, press q\n"
```

This says welcome then prompts the user to enter 1 to list all patients and diagnoses, enter 2 to run a new diagnosis, or enter q to quit the program. Note the newline characters. Feel free to change the wording as you see fit. Once we have a welcome prompt, we can **display** it and **store** user input like this:

```
def main():
    selection = input(welcome_prompt)
```

To run the function, simply add

```
main()
```

When you run the code, you should see the welcome prompt and when you enter your selection, it will be stored in the "selection" variable. Now, we want to **redirect** the user to a **list_patients() function** if they enter "1", a **start_new_diagnosis()** function if they enter "2", or **quit** if they enter "q". Go ahead and create these two functions and just put a print statement in each for now, something like this:

```
def list_patients():
    print("List all patients and diagnoses")

def start_new_diagnosis():
    print("Starting a new diagnosis")
```

To respond to the user's "1", "2", or "q" response and redirect, we can use **if statements** in main() like so:

```
def main():
    selection = input(welcome_prompt)
    if selection == "1":
        list_patients()
    elif selection == "2":
        start_new_diagnosis()
    elif selection == "q":
        return
```


Note how when the user enters “q”, we simply return as that’s our quit command. Go ahead and run the program and test out the commands. You should see that when we enter 1 or 2, the appropriate function is called which prints a statement then stops running the program. However, we don’t want to stop running the program until the user enters the quit command or manually stops running it. To keep the program running, we can enclose the code in main() within a **while() loop**:

```
def main():
    while(True):
        selection = input(welcome_prompt)
        if selection == "1":
            list_patients()
        elif selection == "2":
            start_new_diagnosis()
        elif selection == "q":
            return
```

This ensures that the code keeps running until we enter “q” at which point the function returns which breaks out of any control flow within the function (if statements and while loops included). With this addition, we can keep running new diagnoses and printing patient lists as many times as we want without having to re-run the program every time.

Start a New Diagnosis

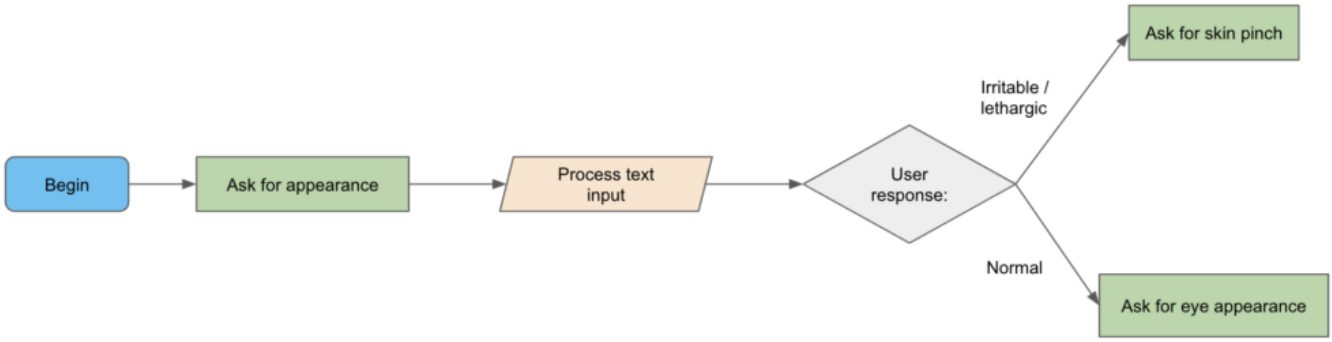
Now that we have an entry point and a way to direct program flow, let's start implementing the actual functionality starting with the diagnosis functions. We are going to use our **start_new_diagnosis() function** to enter diagnosis mode by asking the user for their name and starting the questionnaire. We already have a function to start the diagnosis so let's clear it out and create a prompt so we can ask the user for their name, something like this:

```
name_prompt = "What is the patient's name?\n"
```

We can place that just below the welcome_prompt. Like before, we'll use the input() function to present the prompt and get the user's input, this time to capture their name:

```
def start_new_diagnosis():
    name = input(name_prompt)
```

We can then begin the actual diagnosis. The first step of the diagnosis is to assess the patient's **general appearance**. If you recall from the decision tree presented previously, there are two possible choices the user can make, each of which results in a different next step. To refresh your memory, here is a flowchart of this first decision:



We read this as follows:

1. Display a prompt to the user, asking for their appearance
2. Capture the user's response
3. Make a decision based on that response:
    1. If normal, move to the eye assessment
    2. If irritable/lethargic, move to the skin assessment

The first step requires a prompt and some input capture so let's create an appearance_prompt and a function to assess the appearance:

```
appearance_prompt = "How is the patient's general appearance?\n - 1: Normal appearanc
e\n - 2: Irritable or lethargic\n"

def assess_appearance():
    appearance = input(appearance_prompt)
```


We've displayed the prompt and captured the user input so the next step is to **make a decision** based on the user input. As the skin and eye assessments contain their own logic, we should create a separate function for each:

```
def assess_eyes():
    print("Assessing eye appearance")

def assess_skin():
    print("Assessing skin appearance")
```

We're prompting the user to enter either **"1" or "2"**; if the user enters "1", we move on to the eye assessment whereas if the user enters "2", we move on to the skin assessment. We can use a simple if-elif like so:

```
def assess_appearance():
    appearance = input(appearance_prompt)
    if appearance == "1":
        assess_eyes()
    elif appearance == "2":
        assess_skin()
```

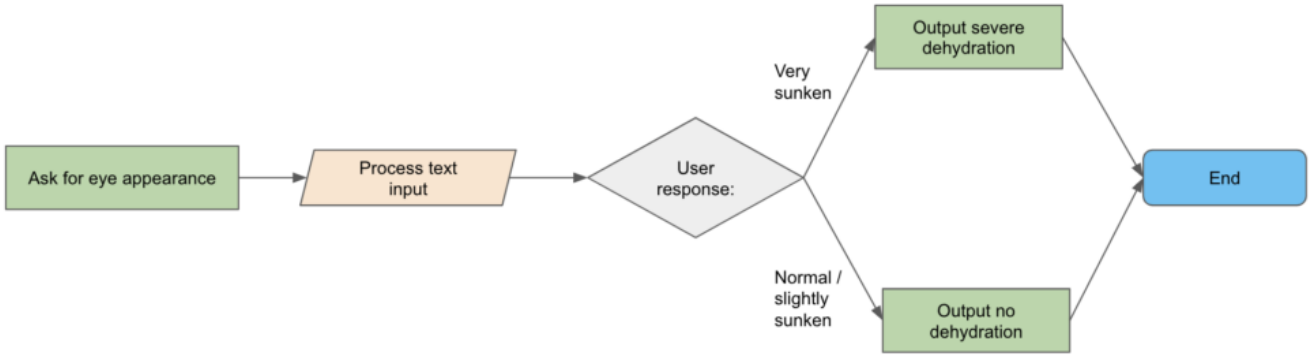
The last step is to **call the function at the appropriate time**. As this is the first step of the diagnosis, we should call it in start_new_diagnosis(), right after capturing the patient's name:

```
def start_new_diagnosis():
    name = input(name_prompt)
    assess_appearance()
```

If you run the code, you should be able to start the diagnosis and get as far as to print either "Assessing eye appearance" or "Assessing skin appearance".

Assess User's Eyes

To complete the diagnosis, we need to assess the user's eyes or skin depending on how the user responded to the appearance question, so let's start with the eye assessment. This time, there is no next step; we need to return a diagnosis depending on user selection rather than calling another function. Besides the end result, this will follow a very similar pattern to the **assess_appearance()** function:



Notice the similarities to the previous flow diagram. The main difference, besides the wording, is that after **we output either no or severe dehydration**; we end rather than moving to the next step. To start, let's create a few string variables to represent the eye appearance prompt and the different levels of dehydration (no, some, or severe):

```
eye_prompt = "How are the patient's eyes?\n - 1: Eyes normal or slightly sunken\n - 2
: Eyes very sunken\n"

severe_dehydration = "Severe dehydration"
some_dehydration = "Some dehydration"
no_dehydration = "No dehydration"
```

Next, we want to implement the **assess_eyes()** function to check the user's response (either "1" or "2") and return some level of dehydration (no or severe dehydration, respectively). However, rather than capturing the input and printing the results, we're going to take in the **user response as a parameter** and return the dehydration string. This will make it easier to test later on. We can do so like this:

```
def assess_eyes(eyes):
    if eyes == "1":
        return no_dehydration
    elif eyes == "2":
        return severe_dehydration
```

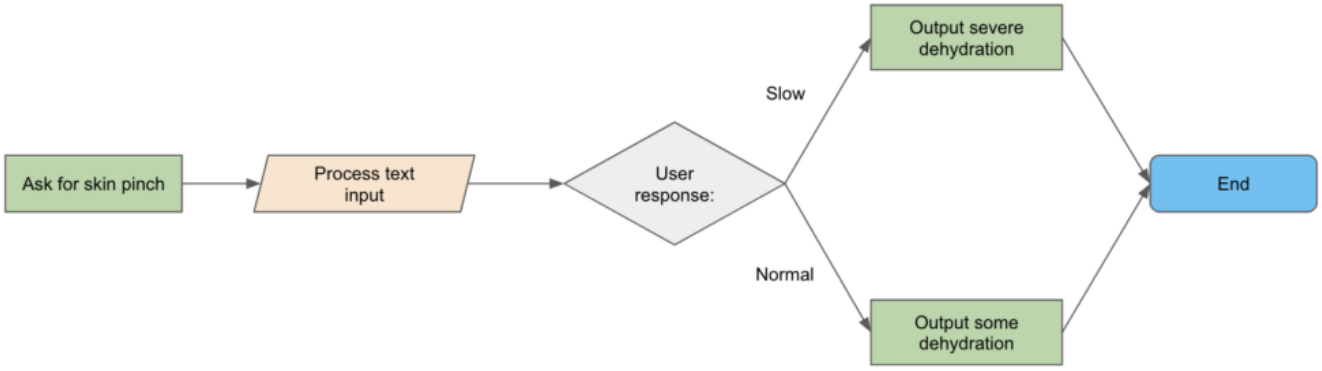
Note the parameter and the return statements. We're already calling the function from within assess_appearance() but we need to make some changes. We first need to **capture the user input** and pass that into the assess_eyes() function. We then need to **retrieve the output** of assess_eyes() which we will then return from assess_appearance(). This allows us to propagate the final diagnosis back into the start_new_diagnosis() function so that we can conveniently save it along with the user's name. We can update assess_appearance() and start_new_diagnosis() like so:


```
def assess_appearance():
    appearance = input(appearance_prompt)
    if appearance == "1":
        eyes = input(eye_prompt)
        return assess_eyes(eyes)
    elif appearance == "2":
        assess_skin(skin)

def start_new_diagnosis():
    name = input(name_prompt)
    diagnosis = assess_appearance()
```

Assess User's Skin

At this point, we've almost finished the diagnosis algorithm! All that remains is to do the same as the assess_eyes() function but this time, apply it to **assess_skin()**. The process will be almost identical; the only differences are the prompts and the final diagnoses:



We can start by adding a skin_prompt:

```
skin_prompt = "How is the patient's skin when you pinch it?\n - 1: Normal skin pinch\
n - 2: Slow skin pinch\n"
```

And then update assess_skin() to mimic assess_eyes(), this time outputting either some or severe dehydration:

```
def assess_skin(skin):
    if skin == "1":
        return some_dehydration
    elif skin == "2":
        return severe_dehydration
```

We now need to update assess_appearance() to **capture the user's response to the skin_prompt and fetch and pass along the final diagnosis** from the skin assessment:

```
def assess_appearance():
    appearance = input(appearance_prompt)
    if appearance == "1":
        eyes = input(eye_prompt)
        return assess_eyes(eyes)
    elif appearance == "2":
        skin = input(skin_prompt)
        return assess_skin(skin)
```

If you run the code, you should be able to fully assess the patient and finalize a diagnosis. However, we're not doing anything with it other than saving it to a local variable for now. Feel free to print it out if desired, otherwise, we'll finish the saving and printing functionality in the next section.

Display Diagnoses

By now, we've almost completed the diagnosis portion of the project. The final tasks are to implement a way to **save the final diagnosis, print all previous diagnoses, and add some error checking**. Let's start with the patients and diagnoses list and printing the diagnoses. We'll keep this simple; we just want a list of strings formatted like so:

"Patient-name – diagnosis"

Go ahead and create a list of strings with some default values so that we have something to print even if we haven't run a diagnosis yet. We can create this just below the prompts like so:

```
patients_and_diagnoses = [
    "Mike - Severe dehydration",
    "Sally - No dehydration",
    "Kate - Some dehydration"
]
```

Feel free to add whichever patient-diagnosis combinations you like; just make sure that the diagnoses **match the 3 possibilities** that we added. In order to print these out, we can use a simple for loop. The best part is that we already have a function and are calling it in the appropriate place. Simply replace the current implementation of list_patients() with this:

```
def list_patients():
    for patient in patients_and_diagnoses:
        print(patient)
```

This **iterates through our list and prints out each element** (or nothing if the list is empty). Go ahead and run the program and try out the print functionality; you should see your entire list printed to the console and then the welcome prompt again.

That's it for the print functionality! All that remains now is to implement the save-diagnosis functionality and perform error-checking. We can handle the error-checking in the next section but let's implement the save function now. We want a function that will **take in a name and diagnosis, create a name-diagnosis string, add it to the list, and print out the final diagnosis**. Something like this can accomplish all of that in just a few lines:

```
def save_new_diagnosis(name, diagnosis):
    final_diagnosis = name + " - " + diagnosis
    patients_and_diagnoses.append(final_diagnosis)
    print("Final diagnosis:", final_diagnosis, "\n")
```

Just **make sure that the final_diagnosis string is in the same format as your current list** of patients and diagnoses. Now we need to call the function and what better place than right after running the diagnosis? We can stick the function call at the end of start_new_diagnosis, passing in the name and diagnosis like so:

```
def start_new_diagnosis():
    name = input(name_prompt)
```



```
    diagnosis = assess_appearance()
    save_new_diagnosis(name, diagnosis)
```

And voila, we're done with the functionality! All that remains is to perform some error checking which we'll handle in the next section. For now, run the code, start and save a new diagnosis, and then print the patients and diagnoses to see that your new diagnosis was, indeed, saved to the list. Note, however, that any new diagnoses will not persist when you stop running the code and start it again. This is because we're not storing the diagnoses in a database and so all the memory gets cleared when you stop and start running the program again.

Handle User Errors

The final implementation step is to handle any possible mistakes made by the user. We cannot guarantee that users will use your program exactly as intended every time; whether by accident or on purpose, **people will use your programs incorrectly** at some point. It's up to us as the developers to ensure that either:

- Users cannot possibly use the program incorrectly or...
- If the program is used incorrectly, we let the user know and gracefully handle the errors without allowing the program to crash

In our case, this is mostly handling potentially incorrect user input, for example, selecting a "3" instead of a "2". There are 3 places that this could occur: assess_appearance(), assess_eyes(), and assess_skin(). For each, rather than doing nothing, **we can return an empty string** if the user enters anything other than a "1" or "2". Then, we can handle any empty strings in the final diagnosis. This is easy enough to implement by adding an "else" clause to each of the 3 functions:

```
def assess_skin(skin):
    if skin == "1":
        return some_dehydration
    elif skin == "2":
        return severe_dehydration
    else:
        return ""

def assess_eyes(eyes):
    if eyes == "1":
        return no_dehydration
    elif eyes == "2":
        return severe_dehydration
    else:
        return ""

def assess_appearance():
    appearance = input(appearance_prompt)
    if appearance == "1":
        eyes = input(eye_prompt)
        return assess_eyes(eyes)
    elif appearance == "2":
        skin = input(skin_prompt)
        return assess_skin(skin)
    else:
        return ""
```

This will basically return an empty string any time the user tries to enter something unexpected. To **prevent the algorithm from saving an empty diagnosis or name**, we can add an extra check in our save_new_diagnosis() function. Let's first create an error message and add it below the prompts:

```
error_message = "Could not save patient and diagnosis due to invalid input"
```


We can then check for an empty name or diagnosis string and output the error_message while preventing the program from saving an invalid diagnosis like so:

```
def save_new_diagnosis(name, diagnosis):
    if name == "" or diagnosis == "":
        print(error_message)
        return
    final_diagnosis = name + " - " + diagnosis
    patients_and_diagnoses.append(final_diagnosis)
    print("Final diagnosis:", final_diagnosis, "\n")
```

And that’s it! Our implementation is now fully complete and so is the project implementation stage. Now, we have to evaluate the project’s performance by writing some unit tests.

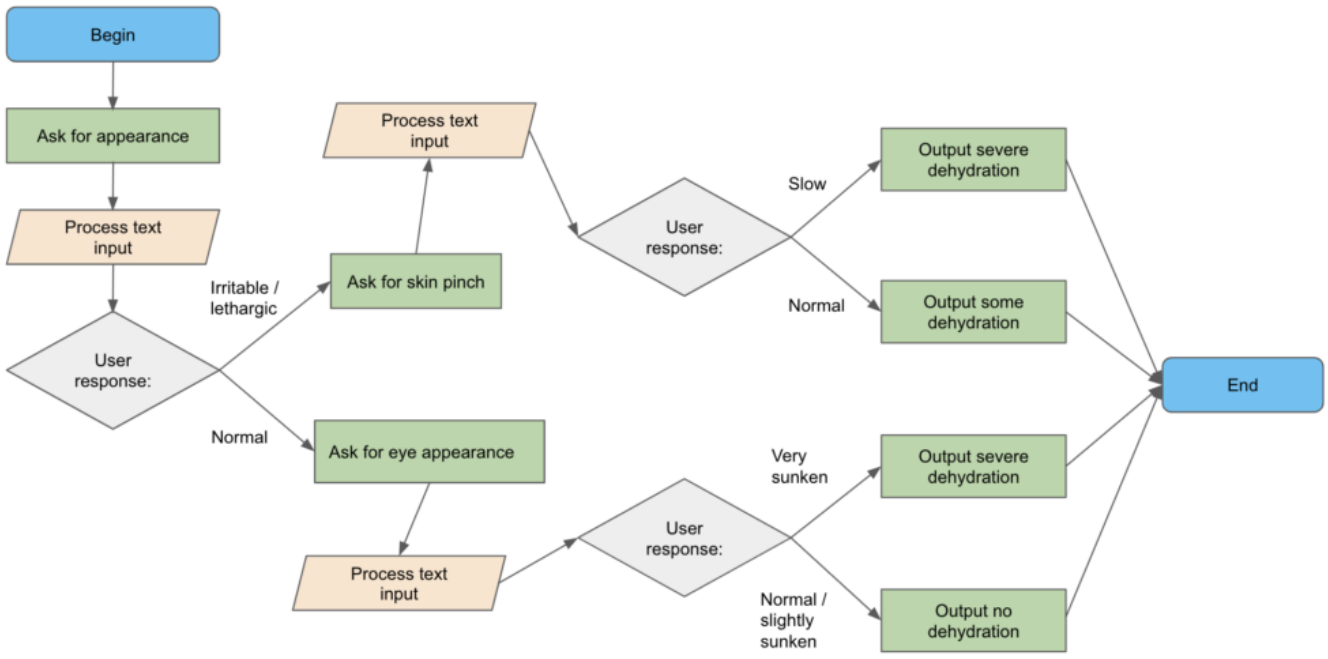
No program is ever complete without undergoing tests to evaluate performance. Tests can come in many forms but we will focus on two types: **unit tests** and **integration tests**.

Unit tests test the performance of individual functions or small blocks of code. These are used to model **all possible input-output scenarios** and ensure that each individual part of the code will hold up under expected and unexpected conditions. When writing the tests for an individual function or block of code, we want to consider all possible input-output combinations to make sure that the code is robust and has no single point of failure. Unit tests provide a quantitative measure of how well a program is performing.

Integration tests test how functions work together when performing a task or set of functionality. They are used to target procedures or overarching functionality and consider the results of a series of functions running together rather than looking at each individual function. Integration tests provide a more qualitative measure of how well a program is performing.

Ideally, we would never have to bother with tests and would write perfect code every time. However, the real world is far from ideal and inevitably, we or someone on our team will make mistakes. Writing appropriate test coverage helps to eliminate, or at least minimize, the amount of and impact of those mistakes. Testing ensures that **the code performs properly under all conditions and environments**. We can also use tests to ensure that **any changes made to the existing code do not break the old code**. Although it can be an annoying and tedious process, testing should always be a part of your repertoire and you should never release code without testing it first.

To demonstrate all the points at which our program could fail, here is the complete flow diagram of our diagnosis algorithm:



We can’t unit test all of our functions. In fact, any of the input-output functions that either use the input() function or just print to the console (list_patients()) cannot be properly unit tested as they either require user interaction or are just used to print. For those functions, we will resort to integration testing. However, we can unit test save_new_diagnosis(), assess_skin(), and assess_eyes().

We don’t have a proper test framework so we will write a few functions to help us to mock the

testing process. These will run the functions with a variety of inputs, compare the outputs to the dehydration strings, and print True or False. **We want to cover every possible input scenario**; for example, for assess_skin(), the 3 types of input are “1”, “2”, and everything else. We could write a test for this function like so:

```
def test_assess_skin():
    print(assess_skin("1") == some_dehydration)
    print(assess_skin("2") == severe_dehydration)
    print(assess_skin("3") == "")
```

This covers all 3 scenarios and will print True for each, considering each test passed. If there was some **flaw in our logic, the function would print False** and we would know that that test was failing somewhere, either due to a faulty test or faulty function implementation. When running this test function, it’s recommended to not run main() and to only run one test function at a time. We can do the same with assess_eyes() like so:

```
def test_assess_eyes():
    print(assess_eyes("1") == no_dehydration)
    print(assess_eyes("2") == severe_dehydration)
    print(assess_eyes("3") == "")
```

When testing **save_new_diagnosis()**, we have **4 scenarios** to take into account: both inputs are empty, either one of the two inputs is empty, or both inputs are valid. We won’t take into consideration that the diagnosis needs to be no, some, or severe dehydration but feel free to include that logic. Also, because the outcome of this function affects the patients_and_diagnosis list, **we have to test the end results against some list values**. We can run the 4 tests like so:

```
def test_save_new_diagnosis():
    save_new_diagnosis("", "")
    print(patients_and_diagnoses == [
        "Mike - Severe dehydration",
        "Sally - No dehydration",
        "Kate - Some dehydration"
    ])
    save_new_diagnosis("Nimish", "")
    print(patients_and_diagnoses == [
        "Mike - Severe dehydration",
        "Sally - No dehydration",
        "Kate - Some dehydration"
    ])
    save_new_diagnosis("", "No dehydration")
    print(patients_and_diagnoses == [
        "Mike - Severe dehydration",
        "Sally - No dehydration",
        "Kate - Some dehydration"
    ])
    save_new_diagnosis("Nimish", "Some dehydration")
    print(patients_and_diagnoses == [
        "Mike - Severe dehydration",
        "Sally - No dehydration",
        "Kate - Some dehydration",
```



```
        "Nimish - Some dehydration"
    ])
```

As you can see, this covers all 4 scenarios. The **first 3 should leave the list unaffected** because the function should return before it saves the name and diagnosis but **the last should append the name and diagnosis string onto the end of the list**. You will have to adjust your test function to take into account whatever dummy values you initialized in your version of patients_and_diagnoses.

When it comes to integration testing, **you'll have to run through the behaviors of each function or set of functions**. This means manually going through each input-output scenario and making sure there is no way that they could produce unexpected results. To test list_patients(), add some patients and diagnoses to patients_and_diagnoses and run the function, ensuring that the printed list matches and changes made. To test assess_appearance(), start_new_diagnosis(), and main(), run the functions individually and ensure that what you're seeing printed to the console matches what is expected. Also make sure to cover all possible cases. Try all 4 possible diagnoses and try running the diagnoses with some invalid inputs.

As long as all of your tests are passing, you're good to go and you can be confident that your code is production-ready! Always be sure to cover every possible scenario to ensure maximum reliability.

With the implementation and testing complete, we're now going to evaluate our project against our design. This way we can ensure everything we built matches the goals we laid out for our project.

Project Definition – Evaluation

As you'll recall, we had two main goals for this project:

- We want to diagnose a patient's state of dehydration based on a short questionnaire given.
- We want to print out a list of all saved patients and their diagnoses given.

In this case, we definitely accomplished this. Through our **start_new_diagnosis()** function, as well as our skin and eye assessment questions, we walk users through diagnosing dehydration with yes or no questions. We are then able to save those patients in a simple array that persists while the program is running. As a bonus, we also tested all these functions to ensure they're working.

We are also able to print all our saved patients with our **list_patients()** function. Remember again that our program cannot save patients between program runtimes, as this would involve databases – which are just a bit beyond the scope of this course.

Project Design – Evaluation

As for the design, we kept it simple and aimed for a program that will run within the Terminal/CMD Prompt. However, as part of this design, we needed the program to be able to:

- Display prompts/questions.
- Process user input.

Our program operates entirely via text input-output, so this is exactly what we built. There is no GUI involved, and we display our questions through text. We then use various variables and parameters to process user input – ultimately storing our patients with simple strings.

Our project also matches the original flowchart that we aimed to replicate.

So, overall, we fulfilled all our goals with the project!

The introduction of algorithms such as this one to the medical field affects society at the social, environmental, and economic levels. There are pros and cons at each level so we will deep dive into each and **treat this as a thought experiment**, theorizing how society would be affected if these algorithms replaced the current diagnosis systems. Keep in mind the pros and cons listed here and just some of many possible effects so feel free to add more to each list.

Social

Pros

- **Faster diagnoses**: In theory, an algorithm should be able to accurately diagnose a patient in just a few seconds to a few minutes based on how long it takes for a user to fill out the questionnaire. This is a huge improvement to having to wait minutes or even hours in crowded wait rooms, only to be bounced around between various medical staff.
- **No human error**: Humans are prone to making mistakes, especially at the end of a 12-hour shift. On the other hand, a machine will perform no worse after 12 hours than at the beginning and wouldn't be biased the way that a human might be.
- **Relieve the burden placed on medical staff**: Ideally, we could replace the diagnosis process entirely with algorithms that would allow staff to move from diagnosis to treatment instead. This could alleviate some of the pressure placed on understaffed medical institutions or allow the institutions to get away with hiring less staff.

Cons

- **Impersonal**: Many people have a deep distrust of machines. Especially when people are anxious about receiving a serious diagnosis, having a real human to talk to can be a comforting experience. However, talking to a robot just doesn't have quite the same effect due to the cold, unfeeling nature of hardware and software.
- **Machine error**: We reduce the chance of human error but there is always the possibility of machine error. Especially when we're dealing with predictions, the results come back as a % certainty rather than an absolute certainty.
- **Loss of jobs**: If algorithms are replacing diagnosis staff, unless the institutions can move the staff into other areas, they will have to let the staff go. Unemployment is bad for everyone so the higher it gets, the worse off society will be.

Environmental

Pros

- **Less test-related waste**: The medical field produces a lot of waste, especially because a lot of tools need to remain sanitary and can only be used once. By replacing testing with diagnosis algorithms, we should be able to reduce the amount of waste accrued through these tests.
- **Less staff**: If diagnosis algorithms replace staff in institutions, there will be less staff driving or commuting to hospitals, clinics, etc. which means less pollution which is better for everyone.

Cons

- **Energy-intensive**: Depending on how sophisticated diagnosis algorithms are, they could become very energy-intensive, using energy not only for computation but also for cooling. This is becoming a bigger problem the more computationally complex systems become. Less energy consumption in general is better for the environment.

Economic

Pros

- **Hire less staff**: If diagnosis algorithms replace staff in institutions, hospitals, clinics, etc. will not have to hire as many staff. This will save them a lot of money in the long term.
- **Perform more diagnoses**: If algorithms can perform diagnoses faster than humans, institutions will be able to diagnose and treat more patients per hour. Many state-funded hospitals receive stipends based on their performance so the more patients they diagnose and treat, the more money they get.
- **Spend less on equipment**: If algorithms replace expensive testing procedures, hospitals will have to spend much less on testing equipment.

Cons

- **Staff lose jobs**: Generally, it's bad for the economy when unemployment rates rise. If hospitals are replacing diagnosis staff with bots, there will be many people out of jobs.
- **Liable for incorrect diagnoses**: Many people receive incorrect diagnoses and treatments and often go on to sue the institutions that gave them incorrect information. This will only be worse for people who receive incorrect diagnoses from machines and already distrust machines.
- **Manufacturers lose money**: If diagnosis algorithms are making many tests obsolete, many medical equipment manufacturers may go out of business. Although this seemingly only affects them, there can be odd trickle-down effects throughout the economy that would be worth taking into consideration.

Here, we want to conduct another thought experiment. This time, we want to **compare what our program would look like when compared to a real-world, production-level version of this program**. How would the actual implementation of the program be different? How would the data collected look different? Again, there are many possible answers, each equally valid but below are just a few that I came up with so feel free to expand upon these lists.

Implementation

- **More sophisticated questions**: We asked very simple binary questions in our implementation. A production-level bot may ask scale questions, multiple-choice, or even text-entry type questions.
- **Gather more patient data**: We only gathered the patient's name before launching into the questionnaire. A production-level bot would likely gather much more information such as weight, height, occupation, medical history, etc.
- **Likely some machine learning involved**: A sophisticated bot would likely employ some sort of machine learning or AI, especially if processing large amounts of data and multiple variables. This would likely need some sort of neural net to function.
- **May give results as a % probability**: When making a prediction, especially where machine learning is involved, the results come back as a % certainty of each possible outcome. From there, we would pick the most likely outcome with the highest % or we would present all of the potential options with the % probability and let the patient decide.

Data

- **Store data in a table**: If we are gathering multiple pieces of information about a patient, we would likely store this data in a table format rather than just a simple list of strings. This is a better way of storing multiple data associated with a single patient, especially if the data is of multiple types.
- **Store data in a database**: We would want this data to persist and be stored somewhere so that we could access it wherever and whenever we need to. The better way of doing this would be some sort of online database where we could save the data in a highly structured format rather than just a list stored locally.
- **Encrypt data**: All data stored in a database should be encrypted to some extent, especially medical data which is highly sensitive.
- **Password-protected**: Likely the data would be stored behind an authentication wall so that only those with the correct credentials could access it. This would help reduce the likelihood of a data leak.

Congratulations on making it to the end of our course! Here, we will briefly summarize everything that we've covered in this course and talk about our next steps. We learned how to design and build algorithms from scratch with the help of decision trees and flowcharts. We then learned how to integrate algorithms into a bigger program and built a dehydration diagnosis bot. This was all based around the 4 stages of project management: definition, design, implementation, and evaluation.

From here, we recommend first improving upon the current project. This way we can gain an in-depth understanding of the topic and how we might add to it. We can make the algorithm more sophisticated or provide a better UX, perhaps displaying better error messages and making sure that all possible edge cases are covered. Once you're ready to move on, we suggest finding other real-world scenarios to model. Find out where algorithms are used around you to control traffic lights and cars, recommend content to you, power city grids, and so much more. Not only will this give you ideas for future projects that you can add to your portfolio, but it will also help you to gain a better understanding of how the world around you works.

That is it from us here at Zenva. I want to say a big personal thank you to everyone who took this course! I really hope you enjoyed it and could learn a lot from it. Being able to design and implement algorithms will help you to problem solve in all areas of your life. Best of luck with your programming careers and I hope to see you in some more courses!

In this lesson, you can find the full source code for the project used within the course. Feel free to use this lesson as needed – whether to help you debug your code, use it as a reference later, or expand the project to practice your skills!

You can also download all project files from the **Course Files** tab via the project files or downloadable PDF summary.

main.py

Found in the Project root folder

This script (Python code) is a console-based program for doctors to add new patient diagnoses based on symptoms, and view a list of patients and their diagnoses. It offers a simple menu-based user interaction, symptom assessment, saving the diagnoses and error handling. It also includes some test functions for the relevant code.

```
welcome_prompt = "Welcome doctor, what would you like to do today?\n - To list all pa
tients, press 1\n - To run a new diagnosis, press 2\n - To quit, press q\n"
name_prompt = "What is the patient's name?\n"
appearance_prompt = "How is the patient's general appearance?\n - 1: Normal appearanc
e\n - 2: Irritable or lethargic\n"
eye_prompt = "How are the patient's eyes?\n - 1: Eyes normal or slightly sunken\n - 2
: Eyes very sunken\n"
skin_prompt = "How is the patient's skin when you pinch it?\n - 1: Normal skin pinch\
n - 2: Slow skin pinch\n"
error_message = "Could not save patient and diagnosis due to invalid input"
severe_dehydration = "Severe dehydration"
some_dehydration = "Some dehydration"
no_dehydration = "No dehydration"

patients_and_diagnoses = [
  "Mike - Severe dehydration",
  "Sally - No dehydration",
  "Kate - Some dehydration"
]


def list_patients():
  for patient in patients_and_diagnoses:
    print(patient)


def save_new_diagnosis(name, diagnosis):
  if name == "" or diagnosis == "":
    print(error_message)
    return
  final_diagnosis = name + " - " + diagnosis
  patients_and_diagnoses.append(final_diagnosis)
  print("Final diagnosis:", final_diagnosis, "\n")


def assess_skin(skin):
  if skin == "1":
    return some_dehydration
  elif skin == "2":
```



```
    return severe_dehydration
  else:
    return ""


def assess_eyes(eyes):
  if eyes == "1":
    return no_dehydration
  elif eyes == "2":
    return severe_dehydration
  else:
    return ""


def assess_appearance():
  appearance = input(appearance_prompt)
  if appearance == "1":
    eyes = input(eye_prompt)
    return assess_eyes(eyes)
  elif appearance == "2":
    skin = input(skin_prompt)
    return assess_skin(skin)
  else:
    return ""


def start_new_diagnosis():
  name = input(name_prompt)
  diagnosis = assess_appearance()
  save_new_diagnosis(name, diagnosis)


def main():
  while(True):
    selection = input(welcome_prompt)
    if selection == "1":
      list_patients()
    elif selection == "2":
      start_new_diagnosis()
    elif selection == "q":
      return


main()


def test_assess_skin():
  print(assess_skin("1") == some_dehydration)
  print(assess_skin("2") == severe_dehydration)
  print(assess_skin("3") == "")


def test_assess_eyes():
  print(assess_skin("1") == no_dehydration)
  print(assess_skin("2") == some_dehydration)
```

```
  print(assess_skin("3") == "")


def test_save_new_diagnosis():
  save_new_diagnosis("", "")
  print(patients_and_diagnoses == [
    "Mike - Severe dehydration",
    "Sally - No dehydration",
    "Kate - Some dehydration"
  ])
  save_new_diagnosis("Nimish", "")
  print(patients_and_diagnoses == [
    "Mike - Severe dehydration",
    "Sally - No dehydration",
    "Kate - Some dehydration"
  ])
  save_new_diagnosis("", "No dehydration")
  print(patients_and_diagnoses == [
    "Mike - Severe dehydration",
    "Sally - No dehydration",
    "Kate - Some dehydration"
  ])
  save_new_diagnosis("Nimish", "Some dehydration")
  print(patients_and_diagnoses == [
    "Mike - Severe dehydration",
    "Sally - No dehydration",
    "Kate - Some dehydration",
    "Nimish - Some dehydration"
  ])

# To test list_patients, add some patients and diagnoses to  patients_and_diagnoses a
nd run the function, ensuring that the printed list matches and changes made

# To test assess_appearance(), start_new_diagnosis(), and main(), run the functions i
ndividually and ensure that what you're seeing printed to the console matches what is
 expected. Also make sure to cover all possible cases. Try all 4 possible diagnoses.
Try running the diagnoses with some invalid inputs.
```

