Python

A Project-based learning approach

After all, tomorrow can only grow out of today—and if one seeks success tomorrow, the factors of success need to be prepared today. --Errico Malatesta, 1892.

Hello and welcome to our introductory Python class. Here's how we suggest you use this document:

1. This document serves as a guide covering most of the material in class. Use it for your reference if you are stuck anywhere during your tasks.
2. The majority of your learning will be done via doing, rather than memorizing. The more you code, the better you will become!
3. The notes in here will definitely not be comprehensive! It's impossible to squeeze the entirety of programming in Python into a single book, let alone a training document like this. You are more than welcomed to use the Internet to look up more tricks and techniques.
4. Remember to have fun!

Image of python from https://pixabay.com/photos/snake-ball-python-python-regius-1455223/
Modified with image to ASCII generator from https://cloudapps.herokuapp.com/imagetoascii/

Table of Contents

Font Usage

Words written in `monospace` are commands to be typed, or references to specific objects like script names or environments. For example:

```
def hello()
```

Terminal output will be `grey out monospace`. For example:

```
ls /
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
```

Setting up your development environment

Before we start programming, it is very important to set up your development environment. Like having a good workshop, having a proper development environment setup will enable you to program efficiently, and most often correctly. Here, we are going to show you how to set up Anaconda along with your programming IDEs.

We will be using the latest stable version of Python (3 in this case) unless otherwise stated in this course. Python, like all other programming languages, is being continuously modified and improved in relation to enhancements to the language as implemented by the community, or in response to identified problems with a certain way the coding was done that can only be fixed by changing the syntax of the scripting language itself.

Unfortunately, when this happens, breakages may occur as packages written in Python2 fail to work without the code being rewritten to fit the Python3 syntax. Other times, you may have no choice but to use Python2 for your projects, which is where setting up Anaconda for a Python2 environment will make coexisting projects less of a headache.

Anaconda, not the Snake!

Anaconda was developed to solve a very specific problem, which is to manage Python packages for various Python-based projects, often times to do with data science. Packages may depend on other packages, which are referred to as dependencies. Sometimes, there may be conflicting dependencies, which can result in a very untidy development environment, as your programs randomly fail to compile.

With Anaconda, we can create specific environments targetting a particular project, with it’s own particular dependencies. We will cover how to install and use Anaconda on Ubuntu, a very popular Linux distro, and Windows 10.

We will know you are ready to do Python programming once you can run the script in Code 1.

Code 1: Your first Python script!

```
def hello():
    """Print "Hello World" and return None."""
    print("Hello World")


hello()
```

Installing Anaconda

You should always consult the official installation guide rather than relying on guides like ours for the most up to date installation methods for your operating system. We will however, endeavour to provide the most basic summary to get you up and running as soon as possible.

Keep the official guide open in your browser window at all times and keep track of what stage of the installation process you are at.

Installing on Ubuntu (and other Linux distros)

The official installation guide for Linux recommends installing some software dependencies first. If you are using Ubuntu, follow the guide under Debian, as Ubuntu is a Debian-derived Linux distro. Run the following in your terminal:

```
sudo apt-get install libgl1-mesa-glx libegl1-mesa libxrandr2
libxrandr2 libxss1 libxcursor1 libxcomposite1 libasound2 libxi6
libxtst6
```

You will notice that we have included `sudo` at the beginning of the command. This is because we need root privileges to install these dependencies system-wide. Make sure you have admin privileges to your account before you proceed.

Once your software dependencies have been installed, download the Anaconda shell script installer. When you follow the download link, make sure you are getting the 64-bit Individual Edition, which is freely distributable under an open source license. Download the installer to your Downloads folder so when you run the installation command from the terminal, it will locate and run the installer.

Note that the version of Anaconda that you have downloaded may be different from the command below. If so, change the filename accordingly.

```
bash ~/Downloads/Anaconda3-2020.02-Linux-x86_64.sh
```

You will first be asked to read through the license agreement. Press the Spacebar to quickly scroll to the bottom. You need to type `yes` to proceed.

Thereafter, we recommend accepting all default options in the installer unless you know exactly what you desire to customise. You should be able to launch Anaconda Navigator now by running `anaconda-navigator` in the terminal, or through Ubuntu's app launcher by typing Anaconda and clicking on the Anaconda icon.

Installing Anaconda in Windows 10

Just like with Ubuntu, it is important to follow the official Windows installer guide for getting Anaconda set up right. We will summarise the steps that you need to follow, but urge you to read and follow the official instructions which may change by the time you read this document.

First, download and run the Windows installer. Make sure that you are getting the 64-bit edition of Anaconda Individual Edition. You may choose to accept all the default settings for the installation.

Creating your first Anaconda environment

Let's prepare to write our first Python program using the Anaconda paradigm of development, by setting up an environment first for your project.

1. Launch Anaconda Navigator
2. Click on the Environment tab, and click on the Create button.
3. Name our environment hello_world, and make sure the box for Python is checked. Apply your settings and wait for the environment to be generated.
4. In the terminal, run:

   ```
   conda info –envs
   ```

   To check and see hello_world is listed as one of the available base environments.

Note that if you followed the recommended default installation options, the base environment will already be created and selected for you by Anaconda.

Installing an IDE

An integrated development environment (IDE) is a sophisticated program that makes your life easier while programming by including all the basic tools to compile and run your programs. There are several IDEs we can use depending on the type of project you wish to run.

All IDEs have there own individual quirks, which are geared towards different workflows. In comparing between the three IDEs we will be installing here, VS Code is the most generalist of the IDEs, as it allows one to pivot to other programming languages if needed. Jupyter is heavily-oriented towards statistical programming in Python and R with line-by-line code execution being it's speciality. Spyder occupies the ground of a general purpose Python IDE, which means you can use it to write Python apps, or use it for statistical programming.

Different instructors may have different preferences too for IDEs. For now, it is good enough to know these are the options you have, and that you can switch between them, and other IDEs you may find online until you find one that fits your idea of the perfect development workflow.

Installing Visual Studio Code

Visual Studio Code (VS Code) is a versatile IDE that can be used with many different programming languages, based upon the extensions that are available for you to install. We recommend using this

IDE largely because it can be used beyond Python if you decide to learn another programming language.

As is the case with Anaconda, we recommend referencing the official guides (Linux or Windows) for installing VS Code for your operating system. If you are using a Linux distro that is not listed in the official guides, check out the package repositories for VS Code to see if it has already been packaged for your distribution.

For Ubuntu, and other distros that support the Snap store, you can install VS Code by running the following in the terminal:

```
sudo snap install --classic code # or code-insiders
```

You can also install VS Code by downloading and running the deb installer.

For Windows, you can download the installer from the official site and run it to set up VS Code. We recommend the User Setup unless you need to have VS Code available to all users on your computer.

Installing Spyder and Jupyter

Spyder and Jupyter are IDEs that are designed specifically for statistical programming. You should install them if your project is heavily-oriented towards using Python for data analysis. Because Anaconda itself originated as a tool to make data analysis as painless as possible, you can easily install these two IDEs from Anaconda.

Installing Spyder and Jupyter is as easy as opening the Home tab in Anaconda Navigator, finding them, and clicking the Install button. In fact, they may even be installed by default!

Hello World!

The “Hello world” and it’s variants are traditional ways to check whether our development environment is ready to go! We do this by asking our program to output a message (also known as printing). If that message can be read, that means you are all set for whatever else that comes next.

We provide a visual metaphor of the `print` command in Figure 1, where taking a photo and seeing its output on your phone as an example. This is how you know your camera and phone works and now you are also ready to be an amateur photographer!

hello_world[1] with Visual Studio Code

1. Click on VS Code in the Home tab of Anaconda Navigator.
2. Once VS Code launches, it will offer to download and install Python-based extensions as it detects the hooks from Anaconda. If it does not, saving your file as `hello_world.py` will also trigger the recommendation for Python extensions.
3. In VS Code, take a look at the bottom left-hand corner of the IDE. You will see that it is pointed towards the base Python environment from Anaconda. Click on it and select the hello_world environment instead.

1 This code snippet was adapted from Spyder’s introductory hello world code.

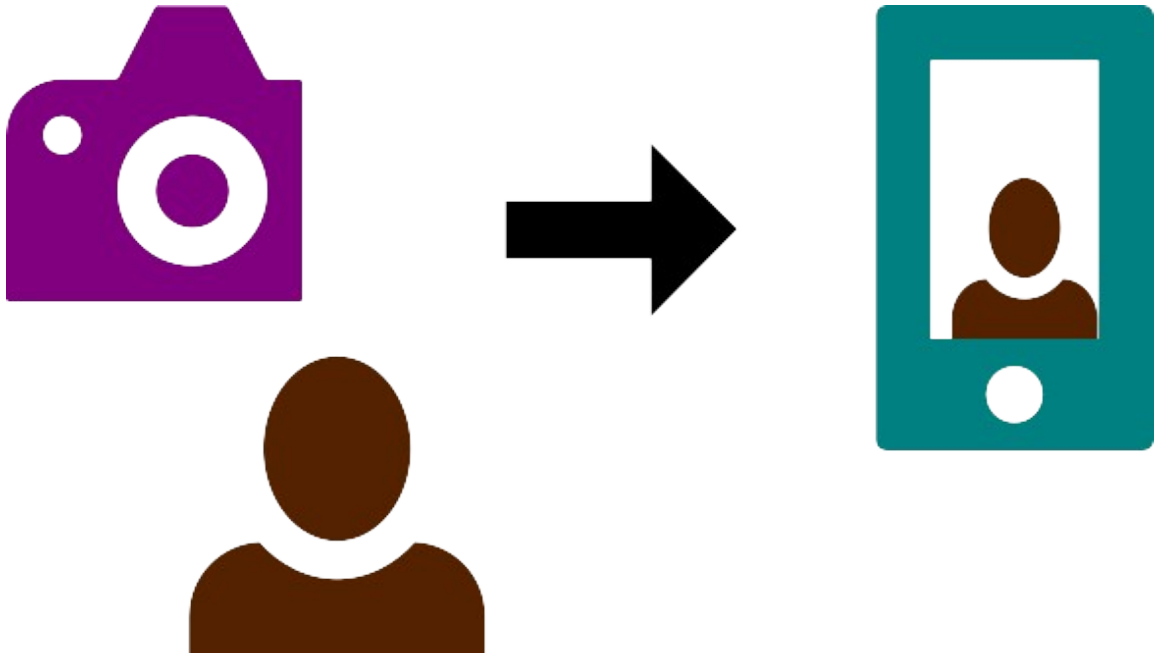

Figure 1: A visual metaphor of the print command. Like taking a photo, and seeing your portrait, the print command simply outputs anything that you feed into it.

4. Click on File > New File, and type in the script in Code 1.
5. Click on File > Save As, and save your file `hello_world.py`.
6. If you have installed the Python extensions as recommended in step 2 above, you should see an arrow pointed towards the right in your scripting tab. Click on it and you will see a bunch of output in the terminal and most importantly:

   ```
   Hello World
   ```

Congratulations! You have successfully run your first Python program in VS Code.

hello_world with Spyder

1. Click on Spyder in the Home tab of Anaconda Navigator.
2. In the scripting pane, write down the hello_world code from Code 1.
3. Press F5 or click on the green arrow on the menubar that says “Run file” in the hover text.
4. You should see a bunch of output from the console and the output:

   ```
   Hello World
   ```

Congratulations! You have executed your first Python script in Spyder!

hello_world with Jupyter Notebook

1. Click on Jupyter Notebook in the Home tab of Anaconda Navigator. Jupyter will open in a browser tab.
2. Select New > Notebook > Python 3

3. In the input cell, enter the hello_world script from Code 1.
4. Click on the Run button. You should see the output with your coding cell showing:

```
Hello World
```

Congratulations! You have executed your first Python script in Jupyter!

Updating Anaconda for a specific environment

The convenience of using Anaconda is you can choose to keep separate environments running with their own distinct set of Python packages with specific dependencies intact. Let's test it out by updating packages in the `hello_world` environment and compare the version with that in the `base` environment.

1. In the Environment tab, select hello_world
2. To filter out only updatable packages, select Updatable from the pull down menu from the default Installed filter setting. Take note of the version number of the program to be updated.
3. Click on the green Apply button and in the verification page, click on Apply again.
4. Once the installation is complete, take a look at the version of the updated package. Change back to the base environment. You should see that the package in the base environment remains the same.

Understanding Python

Now that you successfully wrote, compiled, and ran your first Python program, we can dive into what are the parts of the script in Code 1, and their purpose, line by line.

Your first Python script explained

Firstly, "Hello World" and it's variants are by tradition, the first program one would write and run when learning a new programming language. We use them to determine whether or not you have set up your development environment correctly.

Let's look at the first line you wrote:

```
def hello():
```

We are creating our function using `def`, and the name of our function is `hello()`. End the function defining syntax with a colon (:). At this point, we have an empty function, that does nothing. Let's look at the next line:

```
    """Print "Hello World" and return None."""
```

Notice that there is spacing before this line compared to the `def` line before it? That indentation is how the Python interpreter knows this line belongs to the `hello()` function above.

This line is known as the documentation string, or *docstring* and describes what the program does. It is useful for documentation purposes, and may use many lines depending on the complexity of the software.

Docstrings always start with a capitalised letter and end with a period. Here they are enclosed in triple double quotes (”””) but can also be declared using triple single quotes (’’’). Because this docstring fits into a single line, it is also know as a one-line docstring.

The star of the script is in the next line:

```
    print("Hello World")
```

This line prints the output “Hello World” into your console when you execute the program.

Finally, we execute our function with:

```
hello()
```

Let’s make a game!

It of course, would be a downer if all we did after setting up your ideal development environment culminated in making a program that only prints the same line over and over. Let’s work on creating another classic introductory learning program, the number guessing game[2]. There is a lot more code to work with, as you can see in Code 2, but don’t worry, we will spend some time explaining how every single line fits.

For the introductory part of this exercise, we will be using VS Code to script our programs. Feel free to experiment with other IDEs during this exercise if you want to try something else!

First lets write down what we want our game to do:

1. Randomly select a number between 1 and 100.
2. Ask the player to input their guess in that range.
3. Tells the player whether their guess is too low, or too high, and repeats step 2.
4. When the player makes a correct guess, congratulate the player and ends the game.

We start of with a line that will make your game convenient to execute on Unix-based OSes, like Ubuntu or the MacOS, without having to prepend `python` before the name of your script file. The following command allows you to run the program directly from the terminal without having to load up your IDE:

```
#!/usr/bin/env python
```

You may find variations of this line online. We have chosen this one as it automatically selects the default Python interpreter in your Unix-base OS. On Windows, you will still need to directly reference the Python interpreter first outside of the IDE.

2 Adapted from code at https://www.geeksforgeeks.org/number-guessing-game-in-python/

Code 2: What's the number?

```
#!/usr/bin/env python

import random

def main():
    """Start a number guessing game between 1 - 100."""
    print("Guess the number!")

    x = random.randint(1, 100)
    guess = None

    while x != guess:

        guess = int(input("Pick a number between 1 to 100: "))

        if x == guess:
            print("You genius!")
            break
        elif x > guess:
            print("Try a bigger number.")
        elif x < guess:
            print("Try a smaller number.")

main()
```

Next, we need a way to generate random numbers. Rather than code that, we can save time and effort by reusing code that has already been written and made available in Python, called modules. We can import the module responsible for random number generation with the line:

```
import random
```

Now let's define our game function, which we are calling the main function:

```
def main():
```

Although we are technically only running one function in total, and you can technically rewrite the code here without invoking a function, it's good practice to do so as you build your program, as you

can then conveniently borrow code previously written without copying wholesale the entire code redundantly. Think of it as creating your own module to import!

Next, a one-line docstring describing the function we are about to write:

```
"""Start a number guessing game between 1 – 100."""
```

We first write a line welcoming the player to the game. Substitute the message with something else by replacing the string between the double quotes:

```
print("Guess the number!")
```

A random number between 1 and 100 can be chosen via the following line:

```
x = random.randint(1, 100)
```

First, we initialise a variable, `x`, and assign it a random number from the `random` module using a method found within it called `randint`, which you may correctly guess stands for "random integer". VS Code's Python extension by default allows you to see the correct syntax[3] to fill out the information we want, which is to set a range of between 1 to 100. Feel free to tweak the numbers if you want to make the range larger or smaller.

Next, we need to initialise a blank variable that is ready to receive our player input. Let's call this variable `guess`, and set the value to `None`.

```
guess = None
```

Now let's define a while loop so the game doesn't end even if the player makes a wrong guess:

```
while x != guess:
```

The `while` command starts the loop, and next we define the logical test to keep the game running as long as the player enters the wrong number. The symbols "`!=`" when combined mean "not equal to". So we can read the whole line as, "restart from the beginning of the while loop as long as the player's guess does not equal the random number generated by the game."

Next we will use the guess variable we initialised earlier to fill in a number, and leave explicit instructions for the player:

```
guess = int(input("Pick a number between 1 to 100: "))
```

The int command tells Python to convert the input into an integer. The `input` function comes with a `prompt` parameter, which allows us to set a message for the user before the input. Change the value in the double-quotes to personalise your code.

The next code chunk sets up the logical test for each of the three scenarios that are possible if the use inputs a number between 1 and 100.

1. The number is correct.
2. The number is too small.
3. The number is too big.

---

3 `def randint(self, a: int, b: int)`

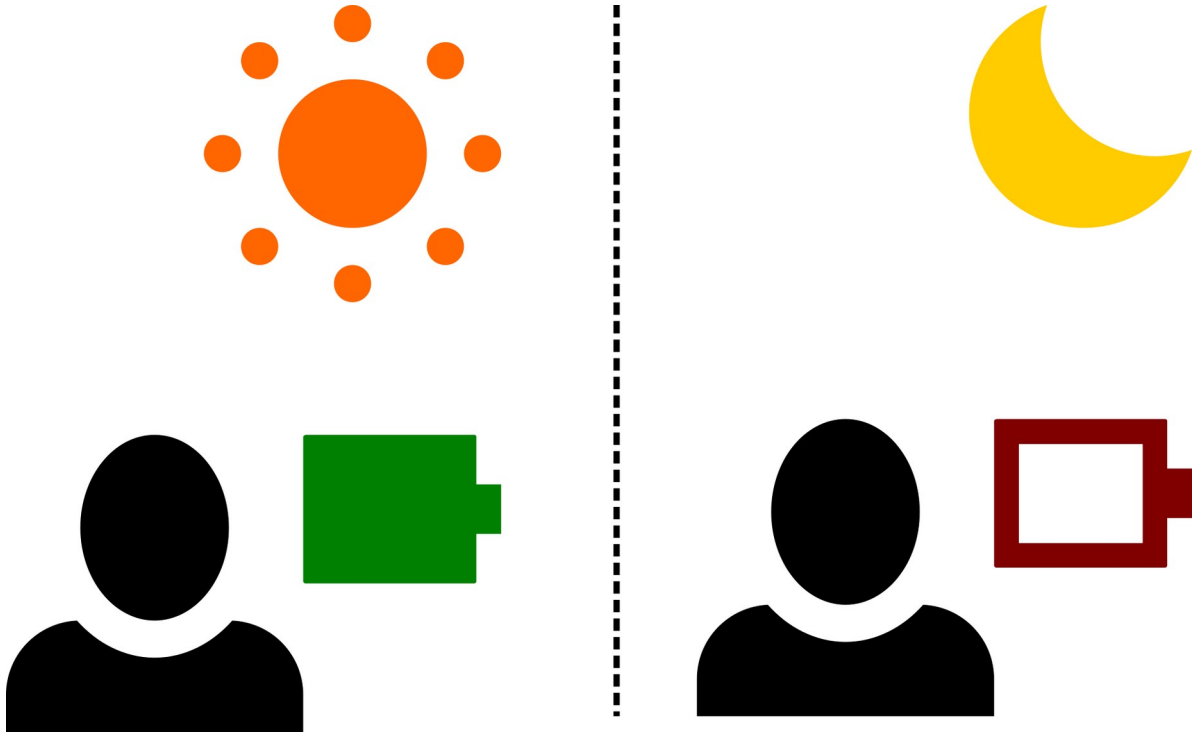

Figure 2: Visual metaphor of an if-else statement. If it's daytime, your batteries are fully charged and ready for the day. If it's night-time, you are probably getting ready for bed.

```
if x == guess:
    print("You genius!")
    break
```

Because `if` is one indent to the right of the `while` function above, it will follow the lead of our `while` statement's setup. This means it will also continue to print the output "You genius!" as long as your guess is the same as the random number generated.

We therefore add the `break` statement to end the while loop.

The next blocks of code are `elif` statements. It is as you may have guessed, a conjugation of the `else` and `if` statements. If our game only had two possible outcomes, for example whether the number we chose matches the random number, and our outcome is either "You got it right" and "Sorry try again", then we could set up our game using only the `else` statement. Figure 2 on page 12 explores this idea with the metaphor that `if` you see someone during the day, they are probably energised and ready for anything, `else` when you spot someone walking home at night, they are probably in need of some sleep.

An `elif` statement is to put it in the simplest terms, multiple `if` and `else` statements put into a single function. This is most useful when we need a process to repeat itself (`while` loop) until we achieve the desired outcome. We illustrate the idea in Figure 3 on page 14 using the idea of trying to get the ideal water level in our cup, through the dubious process of adding water when it rains to the cup if there isn't enough water, and evaporating excess water off when it is sunny if there is too

much water. Since we have decided as part of our game design to guide the players to the correct answer, we use `elif` to make the function evaluate multiple expressions instead.

```
elif x > guess:
    print("Try a bigger number.")
elif x < guess:
    print("Try a smaller number.")
```

You should be familiar with how these statements work now. Try forming a sentence explaining how these `elif` lines work.

We finally end the script by invoking the function we created so it runs after the code is fed through the Python interpreter.

```
main()
```

Test your understanding by trying to modify the code above. Here are some suggestions:

1. Modify the code to use `else` instead of `elif`.
2. Lookup the `random` module and its methods. Can you change your randomly generated variable to something else? What other parameters did you modify?

Mini Project

Using what you have learnt so far, and resources available on the Internet, try to create a guessing game that challenges the player to identify one of the colours of the rainbow that will be randomly selected. Your program should therefore include the colours red, orange, yellow, green, blue, indigo, and violet as potential choices.

The game should not end if the player makes the wrong selection, but instead offer to allow the player to continue playing. If possible, think of hints you can use to lead the player to the answer, just like in the number guessing game in Code 2.

Try to implement safeguards in your programme to prevent crashes as a result of unexpected user input. For example, how does your code handle a user inputting `Blue` instead of `blue`? What happens if someone types `12345` as an answer?

Your code will be evaluated based upon:

1. Functionality: Did you make a functional colour guessing game?
2. Coding: Does your code have clear comments where necessary?
3. Evaluation: Does your code require major or minor improvements?
4. Elements: Does your code include random functions, variables, user input, output, print, while loops, and if/else statements?

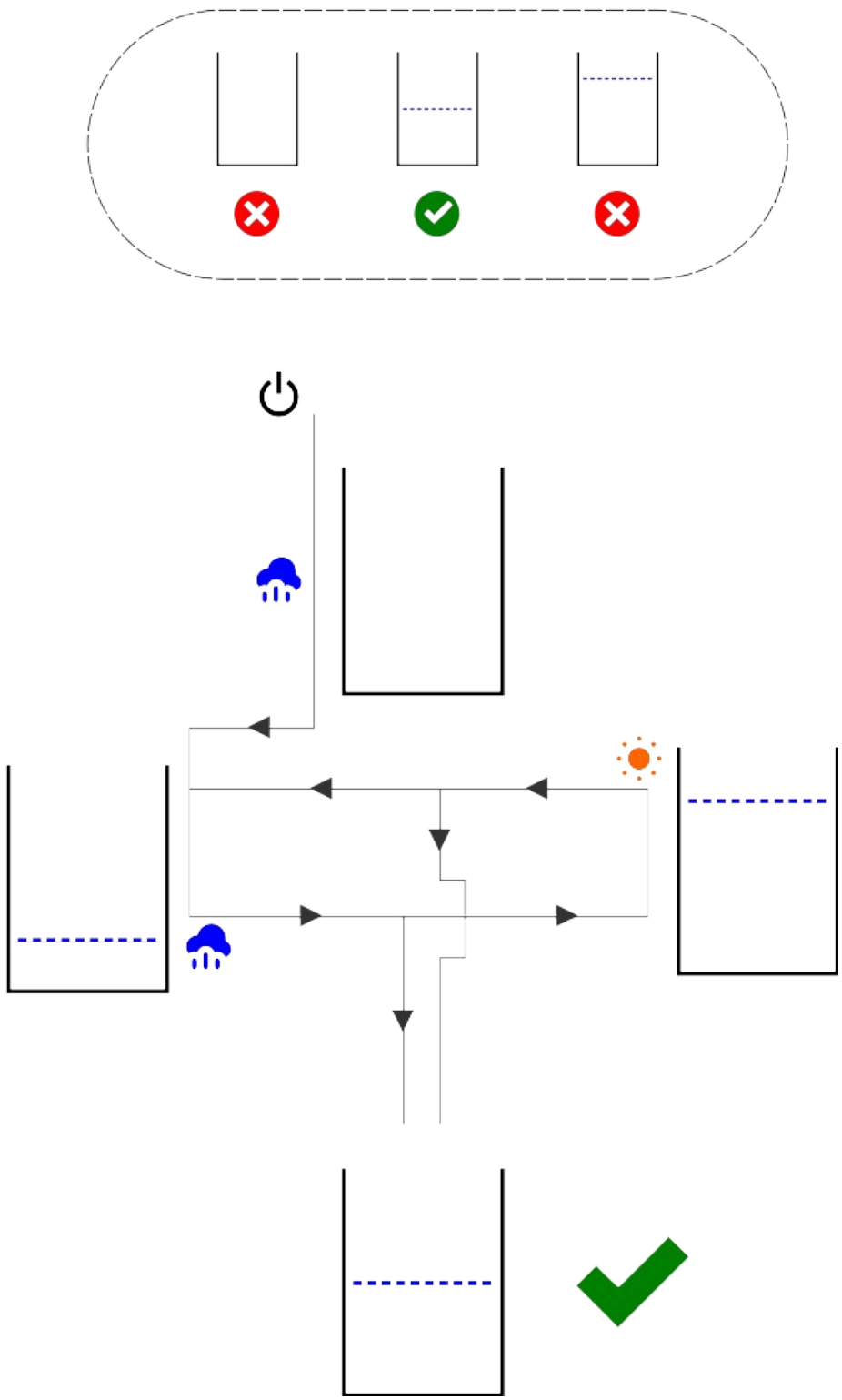

Figure 3: Visual metaphor of a `while` *loop using multiple* `if` *and* `else` *statements, otherwise known as* `elif`*. Here we use an imaginary cup that we want to fill with water until the middle level. We leave the cup in the rain if there is not enough water, and leave it in the sun if there is too much water. Eventually, we get our desired amount of water, and can stop the process.*