

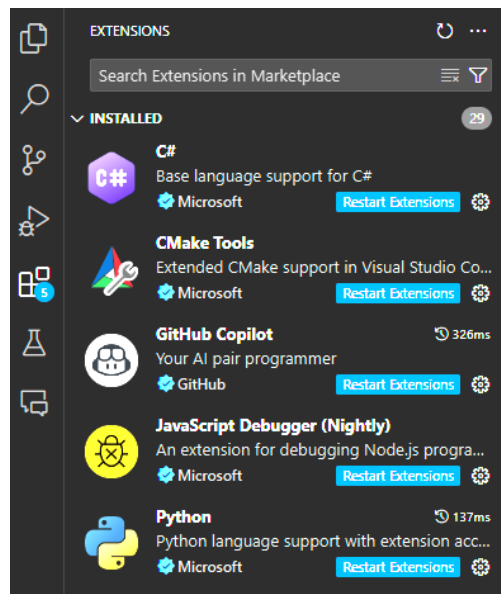
Introduction to Python

The IDE

For this tutorial we will be using Visual Studio Code as our development environment.

After installing VSCode make sure to add the Python extension so you get better coding experience with Python (syntax highlight and autocomplete).

Extensions can be installed from the extensions tab:



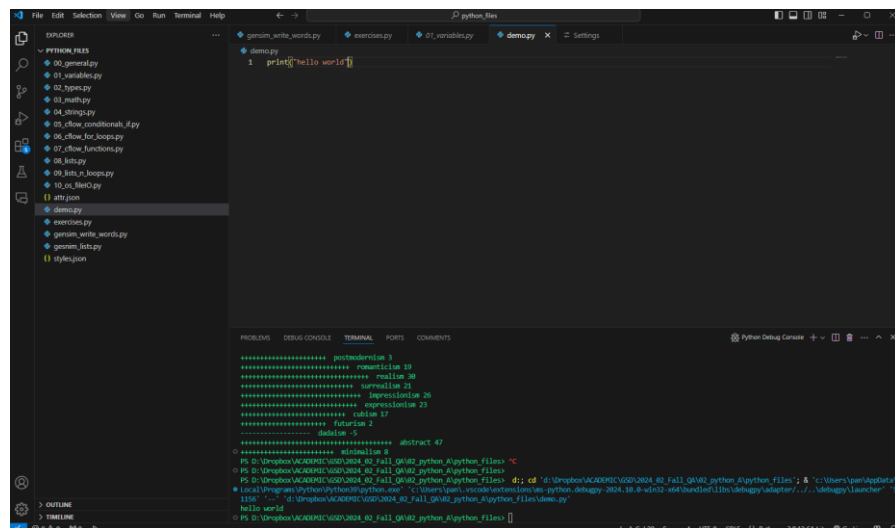
Setup a working folder

In VSCode you can edit files that contain code from anywhere on your hard drive. It is however advisable to open a folder so that it becomes the “working folder” for your VSCode session. This will help with resolving relative paths to other files that you may want to reference through your code.

So just create a folder somewhere on your hard drive and then from VSCode select the File->Open Folder option.

The contents of this folder will appear on the left hand side panel if you select the

“Explorer”  tab:



From the explorer panel you can select the “new file” option and create a file called “demo.py”.

The py extension tells VSCode that this file will contain python code that can be executed.

The “new file” icon appears when you click on a folder name, or you can select the same option from the file menu.

Double-click the demo.py file name and the contents of the file will appear for editing in the main editor panel.

Write the following code and save (File->save)

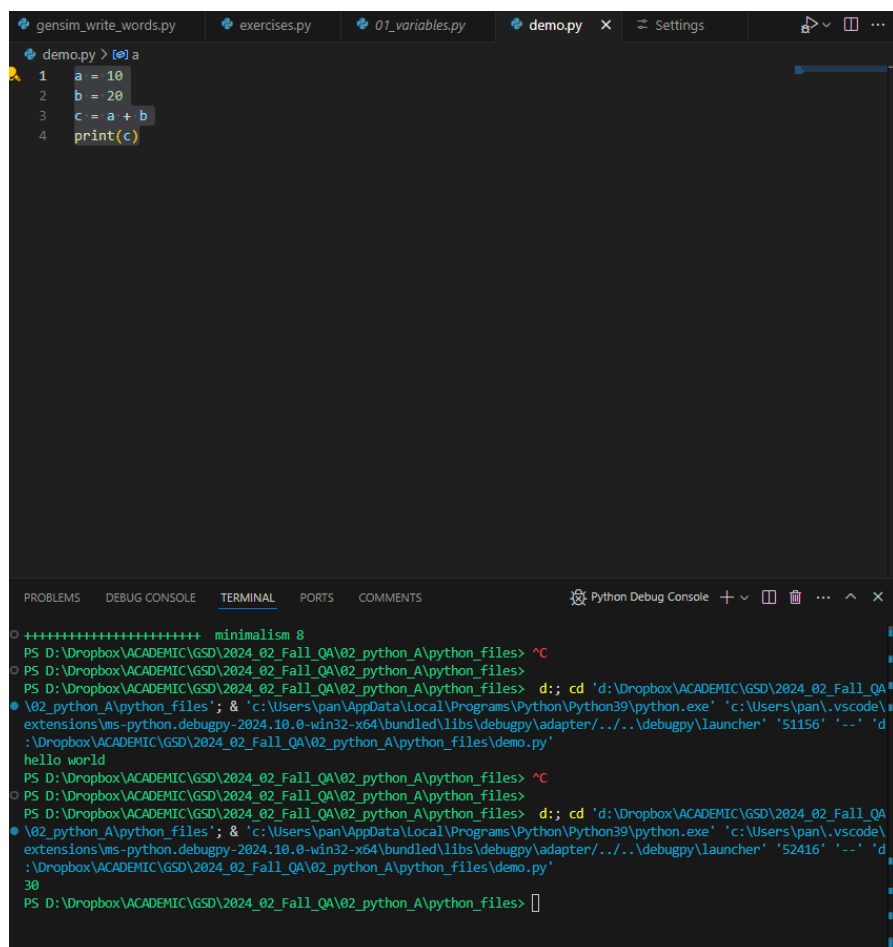
```
a = 10
b = 20
c = a + b
print(c)
```

Debugging

We can run our code from within VSCode. This will basically start the python interpreter as a new process and show all output in the integrated terminal. You don't need VSCode to execute python code but it is convenient.

To execute the code while the demo file is open press the play button on the top

right corner :



The screenshot shows the VSCode editor with a file named `demo.py` open. The code in the file is:

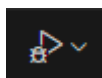
```
1 a = 10
2 b = 20
3 c = a + b
4 print(c)
```

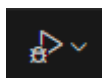
The terminal at the bottom shows the command used to run the file:

```
PS D:\Dropbox\ACADEMIC\GSD\2024_02_Fall_QA\02_python_A\python_files> ^C
PS D:\Dropbox\ACADEMIC\GSD\2024_02_Fall_QA\02_python_A\python_files>
PS D:\Dropbox\ACADEMIC\GSD\2024_02_Fall_QA\02_python_A\python_files> d:; cd 'd:\Dropbox\ACADEMIC\GSD\2024_02_Fall_QA\02_python_A\python_files'; & 'c:\Users\pan\AppData\Local\Programs\Python\Python39\python.exe' 'c:\Users\pan\.vscode\extensions\ms-python.debugpy-2024.10.0-win32-x64\bundle\libs\debugpy\adapter\..\..\debugpy\launcher' '51156' '--' 'd:\Dropbox\ACADEMIC\GSD\2024_02_Fall_QA\02_python_A\python_files\demo.py'
hello world
PS D:\Dropbox\ACADEMIC\GSD\2024_02_Fall_QA\02_python_A\python_files> ^C
PS D:\Dropbox\ACADEMIC\GSD\2024_02_Fall_QA\02_python_A\python_files>
PS D:\Dropbox\ACADEMIC\GSD\2024_02_Fall_QA\02_python_A\python_files> d:; cd 'd:\Dropbox\ACADEMIC\GSD\2024_02_Fall_QA\02_python_A\python_files'; & 'c:\Users\pan\AppData\Local\Programs\Python\Python39\python.exe' 'c:\Users\pan\.vscode\extensions\ms-python.debugpy-2024.10.0-win32-x64\bundle\libs\debugpy\adapter\..\..\debugpy\launcher' '52416' '--' 'd:\Dropbox\ACADEMIC\GSD\2024_02_Fall_QA\02_python_A\python_files\demo.py'
30
PS D:\Dropbox\ACADEMIC\GSD\2024_02_Fall_QA\02_python_A\python_files> 
```

The text in blue that appears in the terminal is the command that the environment executes to run your file. This is useful to check sometimes, as it shows you which python version is been used.

After that you should see the number '30' which is the result of the operation we calculated in the demo.py script.

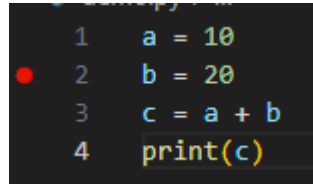


The  button has a little arrow indicating a drop down menu which you can use to select several ways to execute the code. The default is usually to “Run Python File” but for our purposes we almost always want to select “Python Debugger: Debug Python File”

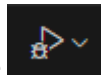
The debugger will be one of your most useful tools when learning how to code. When running in debug mode we can pause the execution of the program and inspect in detail the values of all variables we are using (the state).

The debugger allows us to execute code line by line so in a sense we see our program running in slow motion.

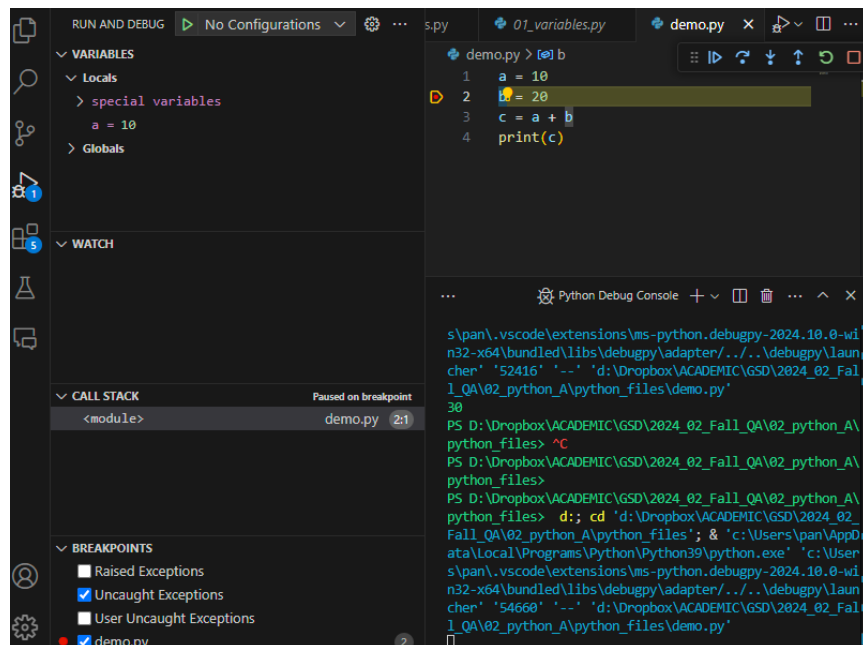
To use the debugger, click on the left border next to a line of code:



A red dot will appear. This represents a “break point”. That means that next time we run our program the interpreter will pause here.



Press the  and select the debug file option now:





The yellow arrow and highlight over the line we added the break point to indicate that the program is running but has paused at this line. On the left panel we see the state inspector where we can see that a variable “a” has been defined with the value 10. That happened on the first line of the program that has already been executed.

We can also see a new toolbar:

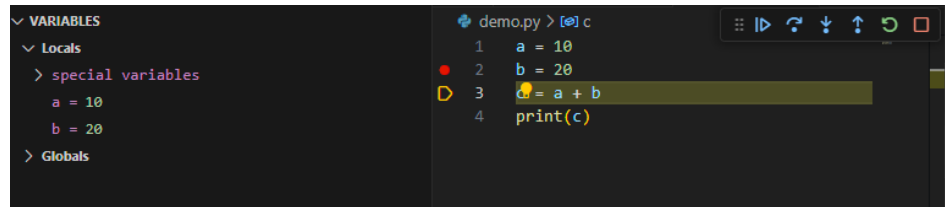


These are the controls for the debugger.

The  button will un-pause the program and continue till the end or the next break-point.

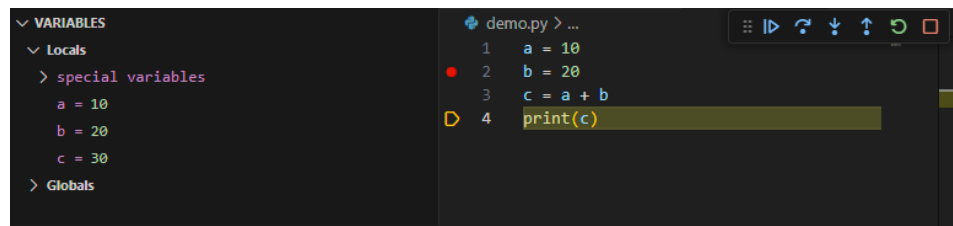
Here we are interested in the  “step over” button. Each time we press this button one line of code is executed.

Press this button once:




You see that the program is now paused at line 3. Because line 2 was just executed we also see in the variable inspector (left panel) that a new variable with the name `b` and value 20 has been created.

Press  once more:



Now a new variable named `"c"` with a value of 30 (the result of adding `a+b`) appeared.

Finally, press  to resume normal program execution and end the debugging session.

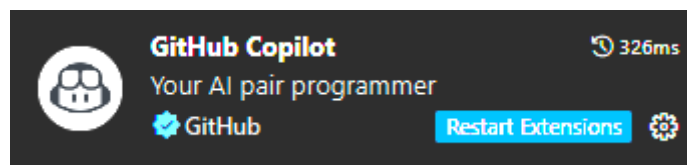
Debugging is an extremely useful way to diagnose problems but also to understand what exactly your code is doing.

Even if you download, copy/paste or have an LLM generate some code, it is a good idea to first run in debug mode with a break point at the beginning and step over each line to try to understand what the program is doing.


Github Copilot

Nowadays it is also convenient to use AI tools as aids when coding. Although they can generate correct code most of the time you still need to be able to formulate what you want the code to do and check if it matches your expectations. It is rare that you won't need to adjust something manually at some point.

Github copilot is one of the AI code assistants that integrates well with VSCode. It requires a Github account to use. After you set it up on github.com then you need to install the github copilot extension in VS code to get inline real-time suggestions while coding as well as the ability to explain existing code.



The first time you run vscode with the copilot enabled it will probably ask you to log in to your github account through a notification that will appear at the bottom right corner of the Visual studio Code window.

When copilot runs you will see the copilot icon  in the status bar:

You can suggest text edits inline when typing code by starting a python comment with the hash symbol and describing what you want to do. E.g. type the following line:

```
#print the odd numbers from 0 to 20
```

After pressing enter copilot will start suggesting line by line a way to achieve what is described in the comment. In my case it generated this code:

```
demo.py > ...
1  a = 10
2  b = 20
3  c = a + b
4  print(c)
5
6  #print the odd numbers from 0 to 20
7
8  for i in range(21):
9      if i % 2 == 1:
10         print(i)
11 print('')
12
```

If you run this code you will see:

```
30
1
3
5
7
9
11
13
15
17
19
```

Which is the expected behavior ("30" is a leftover from the our previous demo program)

You can also use copilot to explain a piece of code. If you select the lines of interest:

```
8  for i in range(21):
9      if i % 2 == 1:
10         print(i)
11 print('')
```

Right click and select Copilot->explain this

You will get a long form explanation of the code.

@workspace /explain Write an explanation for the active selection as paragraphs of text.

```
for i in range(21):  
    if i % 2 == 1:  
        print(i)  
print('')
```

Workspace

> Used 5 references

The provided code snippet is a simple Python script that prints all odd numbers from 0 to 20. Let's break down how it works step by step.

The outer loop, `for i in range(21):`, iterates over a sequence of numbers starting from 0 up to, but not including, 21. The `range` function generates this sequence, and in this case, it produces the numbers 0 through 20. The `range` function is a built-in Python function that returns an immutable sequence of numbers, and it is commonly used for looping a specific number of times.

Within the loop, the `if i % 2 == 1:` statement checks whether the current number `i` is odd. The modulo operator `%` is used here to determine the remainder when `i` is divided by 2. If the remainder is 1, it means that `i` is an odd number. This condition ensures that only odd numbers pass through the `if` statement.

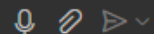
If the condition `i % 2 == 1` is true, the `print(i)` function is called to print the current value of `i`. The `print` function outputs the value to the console. In this context, it will print each odd number from the sequence generated by the `range` function.

After the loop completes, the statement `print('')` is executed. This statement prints an empty line to the console. The `print` function, when called with an empty string, outputs a newline character, effectively adding a blank line to the output. This can be useful for formatting the output, making it more readable by adding a visual separation after the list of odd numbers.

In summary, this code snippet iterates through numbers 0 to 20, checks each number to see if it is odd, prints the odd numbers, and finally prints an empty line for better output formatting.

How can I generate a random number within a specific range in Python?

Ask Copilot or type / for commands



This can be invaluable when learning how to code.

Getting started with Python

Python is a popular programming language that currently dominates ML applications as most ML/AI libraries and research have python implementations and are easily accessible through it.

We will be using python throughout the semester building more complex scripts as we go along.

If you have any programming experience with any other languages these tutorials in the first couple of weeks of our course should be enough to get you started but if you haven't had any programming experience before it is important to make use of the extensive online resources and tutorials.

There are a multitude of books and online tutorials on python. For example:

<https://vscodeedu.com/courses/intro-to-python>

and the official python tutorial series:

<https://docs.python.org/3/tutorial/index.html>

or other resources like:

<https://programming-22.mooc.fi/>

are pretty good. When learning how to program don't just read the tutorials instead open VSCode and type the tutorial code yourself, use the debugger to observe its behavior and try to make small modifications to see the results.

Below is a bare bones condensed explanation of some of the basic elements of the language that we need to get us started. Variables

The following discussion mirrors the files you can download from canvas with some basic code snippets.

1: Variables

Variables are the building blocks of our program. They are names that we use to refer to data in memory.

In python (unlike most other programming languages) a variable is created the first time we assign a value to it using the “assignment operator” =.

```
a = 5
```

The above line **does not** define a mathematical equation. Instead, the = symbol here is called the “assignment operator” and its purpose is to assign a value to a variable.

The line of code above does two things. It first declares that the name “a” is the name of a new variable that we are creating and the “= 5” part says that the initial value of that variable is 5.

A variable as a concept consists of two parts the name “a” and the value (5).

The value can change at any point using the assignment operator:

```
a = 5  
a = 6
```

The above code will create the variable “a” and initialize it with the value 5 but in the next line it will replace its content with the value 6.

Use the debugger with a break point on the first line and step over to see the value changing in the variable inspector.

2: Types

Ultimately all data are represented by bits (0s and 1s) in a computer’s memory, but when programming we want to interpret these data as specific types of objects (numbers, text, images etc...).

A type determines how a piece of data can be interpreted, manipulated and participate in various operations and transformations.

For example, if we have two numbers, we can divide them, but we can’t divide a number by an image (at least there is no clear way to do it).

Each time we assign a value to a variable in python we also assign a type to it. If you have programmed in other languages like the C family (C, C++, C#...) you may have heard that they are “strongly typed”. That means that in those languages when a variable is declared its type is usually fixed. So, a variable that is declared with a numeric value will always represent a number even if the value changes.

Python is not a strongly typed language. A variable that is assigned a number can also be assigned a string (text) or an image or a list of numbers.

```
a = 5  
a = "hello"  
a = [0.4, 3.2, "something"]
```

The above code is valid in python. First the variable a is declared (created) with an initial value of 5 (which is an integer number).

Then its value changes to the string "hello" and therefore its type changes as well to text.

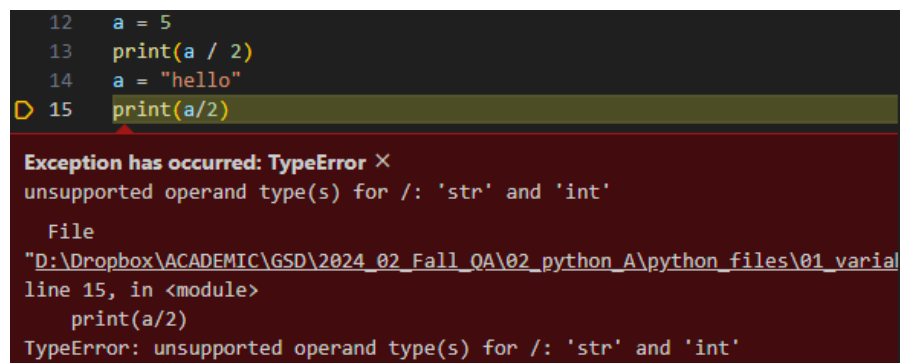
And then its value changes again to a list of two numbers and a piece of text.

After each assignment the variable a has a type which determines what we can do to the value that the variable points to:

```
a = 5
print(a / 2)
a = "hello"
print(len(a))
```

For example, after we assigned a numeric value, we can divide it by another number and after we assigned a string (text) value we can inquire about the length (number of characters) using the len() function.

But the following would fail:



```
12 a = 5
13 print(a / 2)
14 a = "hello"
15 print(a/2)

Exception has occurred: TypeError X
unsupported operand type(s) for /: 'str' and 'int'

File
"D:\Dropbox\ACADEMIC\GSD\2024_02_Fall_QA\02_python_A\python_files\01_varial
line 15, in <module>
    print(a/2)
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

After assigning the text "hello" trying to divide the value of the variable "a" by 2 will crash our program since division between text and number is undefined.

3: Math functions

With numerical variables we can use certain math functions to evaluate more complex expressions. All the usual trigonometric and exponential functions in python are accessed through the math library.

In python we can access functionality from external packages that are not part of the core language using the "import" statement.

For example, if you place the line:

```
import math
```

at the beginning of your file you now have access to all the mathematical functions.

```
a = math.cos(0.5)
print(a)
```

you can use these functions using the "." notation. Typing "math." VSCode will show you a dropdown of all the possible functions you can use.

The dot notation is used to access functions from imported packages or functions and properties of composite objects (e.g. the .x coordinate of a point or the .name of a person's data).

4: strings

Text is represented by the string type in python as it describes a sequence (string) of characters.

We will use strings extensively whether it is to describe inputs to ML models, play with language embeddings or manipulate the file system. There are many things that we can do with or to strings, like convert them to lower or uppercase, search for patterns or change individual characters.

For more information you can check :

<https://developers.google.com/edu/python/strings>

and the official documentation:

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

A string value (or literal) is defined using either single or double quotes. In python the two are interchangeable.

```
a = 'hello'
a = "hello"
```

the above lines have identical effects in both declaring variable “a” and assigning the text value “hello”.

You can concatenate strings (join them together to form a new longer string) using the + operator:

```
a = 'hello'
b = "goodbye"
print (a + "," + b)
```

```
hello,goodbye
```

You can achieve the same using a formatted string interpolation:

```
print(f"{a} , {b}")
```

where the special f prefix tells python that this string is going to contain placeholders for variables. {a} and {b} are replaced by the values of the variables “a” and “b” converted to their text representation. The number 2 in computer memory is different from the character “2”.

You can use any of the multitude of string methods on a string variable using the dot notation:

```
a = "hello"
a_caps = a.capitalize()
print(a_caps)
print("\n\n..... string.upper() function ")
print(a.upper())
```

```
..... string.capitalize() function
Hello

..... string.upper() function
HELLO
```

Because variable `a` is of type string we can use `capitalize()` and `upper()` on it to generate the capitalized and upper case versions of the contained text respectively.

5: lists

We often want to store a set of values and refer to it by a single variable. For example, the x,y,z coordinates of a point, a list of words, the colors of pixels in an image...

This is achieved through various container types. In python the most common ones are the tuples, lists, sets and dictionaries.

The list is the most common type of container and stores a sequence of values. Its closest counterpart in other language is `List<>` in C#, `std::vector` in C++ and built-in arrays `[]` in javascript.

You can create an empty list by simply using the square brackets initializer:

```
a = []
```

or you can populate it with some values:

```
a = [2, 5, -1, 7, 90]
```

In python unlike many other programming languages lists can have heterogeneous data. So, a list that contains both numbers and text is possible:

```
h = [2, 0.5, 3.5, "hello", 3, "again"]
```

and even a nested list (list of lists):

```
h = [[2, 3], [-2, 4], [1, 3]]
```

for example, we could use the above to represent a list of three points with [x,y] coordinates.

You can access elements in a list using the indexing operator `[]`.

```
h = [2, 0.5, 3.5, "hello", 3, "again"]
print(h)

#you can also have lists of lists
h = [[2, 3], [-2, 4], [1, 3]]
print(h)

#you can read and write individual elements from a list using the square
brackets as a subscript
#indexing for lists is zero based, so the first element is [0]
print("\n\n..... reading an element from a list")
print(h[2])

#we can overwrite an element in a list
print("\n\n..... overwriting an element in a list")
h[3] = 12344
print(h)

#you can also access element in reverse with [-1] being the last element in
the list
print("\n\n..... reading an element from a list, indexing starting at
end")
print(h[-2])
```

You can also find the length of a list using the `len()` function. “`len()`” is a core function in python that can be used on any object that is some kind of composite that acts as a container for other objects. This includes strings (list of characters), lists, sets, dictionaries and many other types. `len(a)` evaluates to an integer number that equals the number of elements inside “`a`”, whatever type of container `a` may be.

```
print(len(h))
```

You can add elements to a list using its `append(newelement)` method

```
a = ["red", "green"]
print(a)
a.append("blue")
print(a)
```

and you can remove individual elements or a range of elements by value or by position

```
. a.remove("green")
```

Removes the first element with the value “green” from the list

```
del(a[0])
```

removes the first element in the list `a` (at position 0)

Notice that whenever we manipulate containers, we use the square brackets `[]` either to define lists or access individual elements in them.

5.1: Slices

Python provides a convenient notation for extracting portions of a list that is also used extensively in image manipulation and ML libraries since both deal with data organized in some ordered container, an image being a list of rows of pixel colors or ML libraries using multidimensional arrays called Tensors).

A slice is a portion of a list (e.g. the first 5 elements, or the elements between index 3 and 7 etc...). It is created using the square brackets `[]` operating on a list type variable with the “`:`” operator inside. Here are some examples:

```
a = [0, 1, 2, 3, 4, 5, 6]
#get elements from position 2 (3rd element) to end
print(a[2:])

#get elements between positions 2 and 5
print(a[2:5])

#get the last 2 elements
print(a[-2:])
```

6: Flow Control, conditionals

Until now we wrote simple lines of code that create a variable and then read, print or alter its value.

Most of our programs are structured by a series of special statements that control the flow of the program's execution.

For example, sometimes we want our program to behave differently depending on the value of some variable (conditional branching) or to repeat the same code a few times or over all elements within a list (loop). This is achieved using special keywords and syntax in python.

Conditional statements act as guards in the program where they prevent some code from being executed unless some test is passed.

```
a = 2
b = 5

if a<-5 :
    print('a is less than -5')

print("\n\n..... Simple conditional statement with multiple lines of
code")
if a<b:
    print('a is greater than b')
    print('another line of code')

#conditional statements can define a block of code for each possible
true/false outcome using the if/else structure
print("\n\n..... if/else branching")
if a<b:
    print('a is less than b')
else:
    print('a was is not than b')
```

A conditional statement starts with the special keyword “if” followed by a test and a colon:

Notice that in python the code that is executed conditionally is indented (using tabs) under the if statement

```
if a<b:
    print('a is greater than b')
    print('another line of code')
```

will print the two lines if a<b

but:

```
if a<b:
    print('a is greater than b')
print('another line of code')
```

will print line a if $a < b$ but the second line will always be printed regardless of the $a < b$ test.

The “if” keyword is always followed by something that evaluates to a Boolean type (a special type that can only take the values True or False). For example, when writing $a < b$ we are not expressing inequality in the usual mathematical sense.

Instead, the “<” operator looks at the values of “a” and “b” and evaluates to True if a is less than b or False otherwise. We can write conditional code in this manner:

```
result = a < -5
print(result)
if result:
    print('a is less than -5')
```

Above you can see that the expression $a < -5$ is evaluated to a True or False result that can be stored in a variable and used later.

7: flow control: loops

The second control flow structure is a loop which enables us to repeat a block of code multiple times or until some criteria are met. For example, you may want to visit each pixel in an image to change its brightness, or each vertex point in 3d mesh to move it by a certain amount.

Loops in python begin with the special keyword “for” followed by something that gives a loop a range to iterate over and the colon sign.

The simplest loop is one that iterates over a range of values:

```
for i in range(10):
    print(i)
```

The above code will repeat 10 times and the temporary variable “i” will take the values 0 to 9 successively.

`range(10)` defines a “virtual” list of values from 0 to 9. Range is a very flexible object and can be used to define all sorts of number sequences with arbitrary start and end values as well as step.

As a note, this is equivalent to C languages for loop:

```
for (int i=0; i<10; ++i) {
    }
}
```

```
for i in range(-3, 5):
    print(i)
```

the above code will iterate from -3 to +4. The final value of the range is exclusive.

We can use a loop to sum over some values:

```
total = 0
for i in range(1, 11):
    total += i
    print(f'step {i} : total = {total}')
```

The above code sums all the values between 1 and 10 using a temporary variable “total” as an accumulator.

Notice the special operator +=. This special symbol is a shorthand for `total = total + i`

This again may seem odd if you try to interpret it as a mathematical equation. Remember the “=” assignment operator is not an equality test, instead it assigns the value of the right expression to the variable on the left. What this line says is take the current value of “total” add the value of “i” to it and then replace the value of “total” with the result of that addition. In short it says “increment the value of “total” by “i””

8: functions

Functions enable us to reuse code as a sort of parameterized block of statements that can be executed repeatedly with potentially different inputs.

We already used a few functions like `math.cos(a)` or `print(“hello”)`.

When we import a package like “math” its functionality is encapsulated in functions we can use in our code by writing the name of the function followed by parentheses that contain the arguments (inputs) to the function.

Functions enable us to write simpler, more compartmentalized and reusable code.

So, if we have a block of statements that solve a problem (e.g. compute the distance between two points) and we want to use it in many places it makes sense to encapsulate those instructions into a function that we can call by name.

Functions are defined using the “def” keyword:

```
def add(a,b):  
    return a+b
```

The above code defines a function that takes as inputs two arguments a and b and evaluates their sum. This is of course a trivial function, but we’ll use it as demonstration.

The first line is the function definition that determines what the function’s name is and how many arguments it accepts. It is followed by a colon : and then indented a block of statements that constitute the body of the function. These statements are what is being executed each time we call the function.

If a function evaluates to something, then we say that it returns a value. That means that somewhere in the body of the function we need to use the special statement “return” followed by the value we want the function to evaluate to.

This also means that the function can appear on the right side of assignment operations as it emits a value that can be assigned to a variable. You can think of return as creating an output socket that can be plugged to a variable using the “=” operator.

```
print("\n\n..... adding two float numbers")  
a = add(0.5, 4.2)  
print(a)  
print("\n\n..... adding two integer numbers")  
b = add(3, 4)
```



```
print(b)
print("\n\n..... adding (concatenating) two strings")
c = add("hello", " again")
print(c)
```

Here is a more complex function that computes the maximum of two numbers:

```
def max2(a,b):
    if a>b:
        return a
    else:
        return b

m = max2(4, 8)
print(m)
```

use the debugger to see how this function branches with different inputs.

9: loops over lists

We saw previously how we can create lists of values and how we can use a loop to repeat a set of instructions.

We can combine the two to iterate over the values of a list (or another container). This is useful for transforming data (e.g. change the brightness of each pixel in an image) or computing some total (like a sum, mean or product...).

If we have a list of values, we can use the for-iterator syntax with a temporary iterator variable that will take the value of each element in the list one by one:

```
data = [0.5, 0.6, 0.7, 1.2, 3.5, "hello", 4]

for el in data:
    print(el)
```

we can use such a loop to compute the mean value of a list of numbers

```
data = [0.5, 0.6, 0.7, 1.2, 3.5]
total = 0
for el in data:
    total += el
print(f'total of all elements is {total}')
average = total/len(data)
print(f'average of all elements is {average}')
```

Assignment: exercises

1. Use a loop to print the following

```
[0][1][2][3][4]
```

Start with an empty string:

```
row = ''
```

then create a loop that repeats 5 times (range 0~4) and add the index of the loop iterator to the row string

After the loop print the string row.

2. Use two nested loops to print the following pattern

```
[0,0][0,1][0,2][0,3][0,4]
[1,0][1,1][1,2][1,3][1,4]
[2,0][2,1][2,2][2,3][2,4]
[3,0][3,1][3,2][3,3][3,4]
[4,0][4,1][4,2][4,3][4,4]
```

This can be achieved by placing the code from exercise 1 in a second loop that does one step for each row of numbers. Then the strings that you add to the row will have the form `f'[{row_index}, {col_index}]`

3. Use two nested loops to print the following pattern

```
.....
0.....
00.....
000.....
0000.....
00000.....
000000.....
0000000.....
00000000.....
000000000.....
```

In this case we want to use a conditional statement inside the loop so that "O" is added if the row index is greater than the column index

4. Use two nested loops to print the following pattern

```
0.0.0.0.0.0.0.0.0.0.
.0.0.0.0.0.0.0.0.0.0
0.0.0.0.0.0.0.0.0.0.
.0.0.0.0.0.0.0.0.0.0
0.0.0.0.0.0.0.0.0.0.
.0.0.0.0.0.0.0.0.0.0
0.0.0.0.0.0.0.0.0.0.
.0.0.0.0.0.0.0.0.0.0
0.0.0.0.0.0.0.0.0.0.
.0.0.0.0.0.0.0.0.0.0
```

In this case we want to use a conditional statement inside the loop so that "O" is added if the sum of column and row index is an even number

5. Use two nested for loops to draw the following (not exactly)



The pattern consists of randomly selected forward or back slashes.

You can use the same loops as before but now we want to randomly select the forward or back slash. To do this you need to import the “random” package into your file.

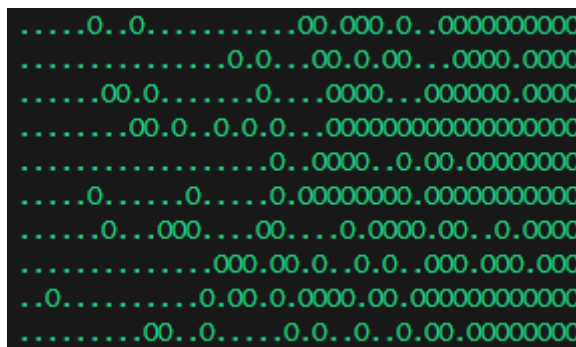
In the inner loop when deciding whether to add / or \ you need to generate a random number R between 0 and 1 using random.random() (see docs). If you want to select either slash with 50% probability you add a conditional statement that will add “/” if R is greater than 0.5 otherwise will add “\”.

By changing the 0.5 threshold to a number between 0 and 1 you can change the probability of either slash being selected.

Here is an example with a different threshold:



6. Create the following pattern (Approximately):



This is an advanced (optional exercise) where we want the probability of selecting “.” Or “O” to change as we move from left to right along each row. The probability goes from 0% “O” to 100 “O”.

