1 - Unit 4 - Lesson 2

October 9, 2023

The creation of the lessons in this unit relied heavily on the existing lessons created by Mrs. Fitz-Zaland as well as the **lecture series** produced by Dr. Milaan Parmar.

1 Variables, Expressions and Statements

One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name that refers to a value.

Variables are a fundamental concept in programming, and they are used to store and manipulate values or data. They are like containers that can hold different types of information, such as numbers, text, or logical values, and allow programmers to perform complex operations on that data.

Variables are important in programming because they enable flexibility and efficiency in code. They allow programmers to store and manipulate data dynamically, meaning that the same code can be reused for different data inputs. For example, imagine you are writing a program to calculate the area of a rectangle. Instead of hard-coding the values of the length and width of the rectangle into the code, you can use variables to store these values and calculate the area based on the variable values. This allows you to easily change the dimensions of the rectangle without having to modify the code.

Variables also allow for better organization of code. By giving variables meaningful names, programmers can make their code more readable and easier to understand. For example, instead of using a generic name like "num1" for a variable that stores a number, a programmer could use a more descriptive name like "userAge" or "totalScore" to convey the purpose of the variable.

In summary, variables are important in programming because they allow for flexibility, efficiency, and organization of code. They enable programmers to store and manipulate data dynamically, which makes their code more adaptable and reusable.

1.1 Variables

In Python, a variable is a named container that stores a value. Think of it like a box with a label on it, where you can put something inside and then use the label to refer to it later.

For example, let's say you want to store the number 5 in a variable called `my_Number`.

1. **In a notebook, you can define this variable in a code cell:**

```
my_Number = 5
```

Now, whenever you want to use the number 5 in your program, you can just refer to it using the variable name my_Number.

2. **Output `my_Number` to the screen by entering the following code into another code cell:**

```
print(my_Number)
```

It should have output the number 5 to the screen.

Variables can hold many different types of values in Python, including numbers, strings, and even more complex data structures like lists and dictionaries. By using variables, you can make your code more readable and easier to manage, since you can refer to values by name rather than hardcoding them into your code.

1.2 Assignment Statements

An assignment statement creates new variables and gives them values. 3. **Type the following 3 assignment statements into the shell:**

```
message = 'And now for something completely different'
```

This statement creates a new variable named message and storedstores the string 'And now for something completely different' into the variable named `message`.

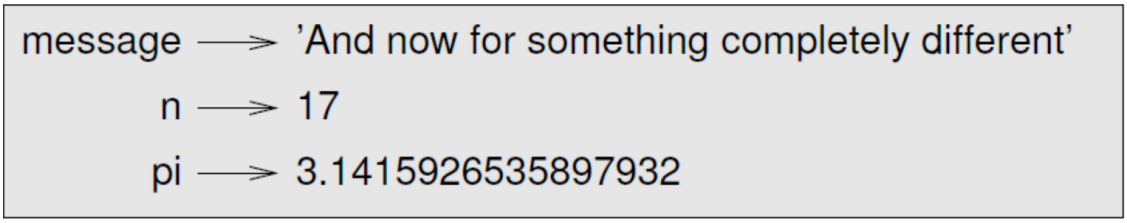
```
n = 17
```

This statement gives the integer 17 to `n`.

```
pi = 3.1415926535897932
```

This statement assigns the (approximate) value of `p` to pi.

A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure is called a **state diagram** because it shows what state each of the variables is in (think of it as the variable's state of mind).



1.3 Variable Names

Giving variables descriptive names makes it easier to understand what a program does. You could have called the `my_Number` variable `x` or `y` and Python would have run the program just the same. But those names don't really tell you much about what information the variable might hold. As you create longer programs, you'll have difficulty keeping track of your variables if you don't give them descriptive names.

Variable names in Python are **case sensitive**, which means the same variable name in a different case is considered a different variable. So `spam`, `SPAM`, `Spam`, and `sPAM` are four different variables in Python.

Variable names are usually lowercase. If there's more than one word in the variable name, it's a good idea to capitalize each word after the first. For example, the variable name `whatIHadForBreakfastThisMorning` is much easier to read than `whatihadforbreakfastthismorning`.

Capitalizing your variables this way is called camel case because it resembles the humps on a camel's back. It makes your code easier to read. Although variable names can be as long as you like, programmers prefer using shorter variable names to make code easier to understand `breakfast` or `foodThisMorning` is more readable than `whatIHadForBreakfastThisMorning`. These are conventions - optional but standard ways of doing things in Python programming. Variable names can contain both letters and numbers, but they can't begin with a number.

1.4 Python Keywords

Keywords are the reserved words in Python.

We cannot use a keyword as a variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language.

In Python, keywords are **case sensitive**.

There are **36** keywords in Python 3.9. This number can vary slightly over the course of time.

All the keywords except **`True`**, **`False`** and **`None`** are in lowercase and they must be written as they are. The list of all the keywords is given below.

Keywords in Python

`False`	`break`	`for`	`not`
`None`	`class`	`from`	`or`
`True`	`continue`	`global`	`pass`
`__peg_parser__`	`def`	`if`	`raise`
`and`	`del`	`import`	`return`
`as`	`elif`	`in`	`try`
`assert`	`else`	`is`	`while`
`async`	`except`	`lambda`	`with`
`await`	`finally`	`nonlocal`	`yield`

Trying to create a variable with the same name as any reserved word results in an **error**:

```
>>>for = 6
File "<ipython-input-1-50b154750974>", line 1
for = 6 # It will give error becasue "for" is keyword and we cannot use as a variable name.
        ^
SyntaxError: invalid syntax
```

4. **Run the following statements code cells to see if they are all valid variable names:**

```
76 trombones = 'big parade'

more@ = 1000000

class = 'Advanced Theoretical Zymurgy'
```

1.5 Expressions and Statements

An expression is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions:

[1]:
```
42
```

[1]:
```
42
```

[2]:
```
n = 17
n
```

[2]:
```
17
```

[3]:
```
n + 25
```

[3]:
```
42
```

When you type an expression at the prompt, the interpreter evaluates it, which means that it finds the value of the expression. In this example, `n` has the value `17` and `n+25` has the value `42`.

A statement is a unit of code that has an effect, like creating a variable or displaying a value.

```
n = 17
print(n)
```

The first line is an assignment statement that gives a value to n. The second line is a print statement that displays the value of n. When you type a statement, the interpreter executes it, which means that it does whatever the statement says.

2 Script Mode

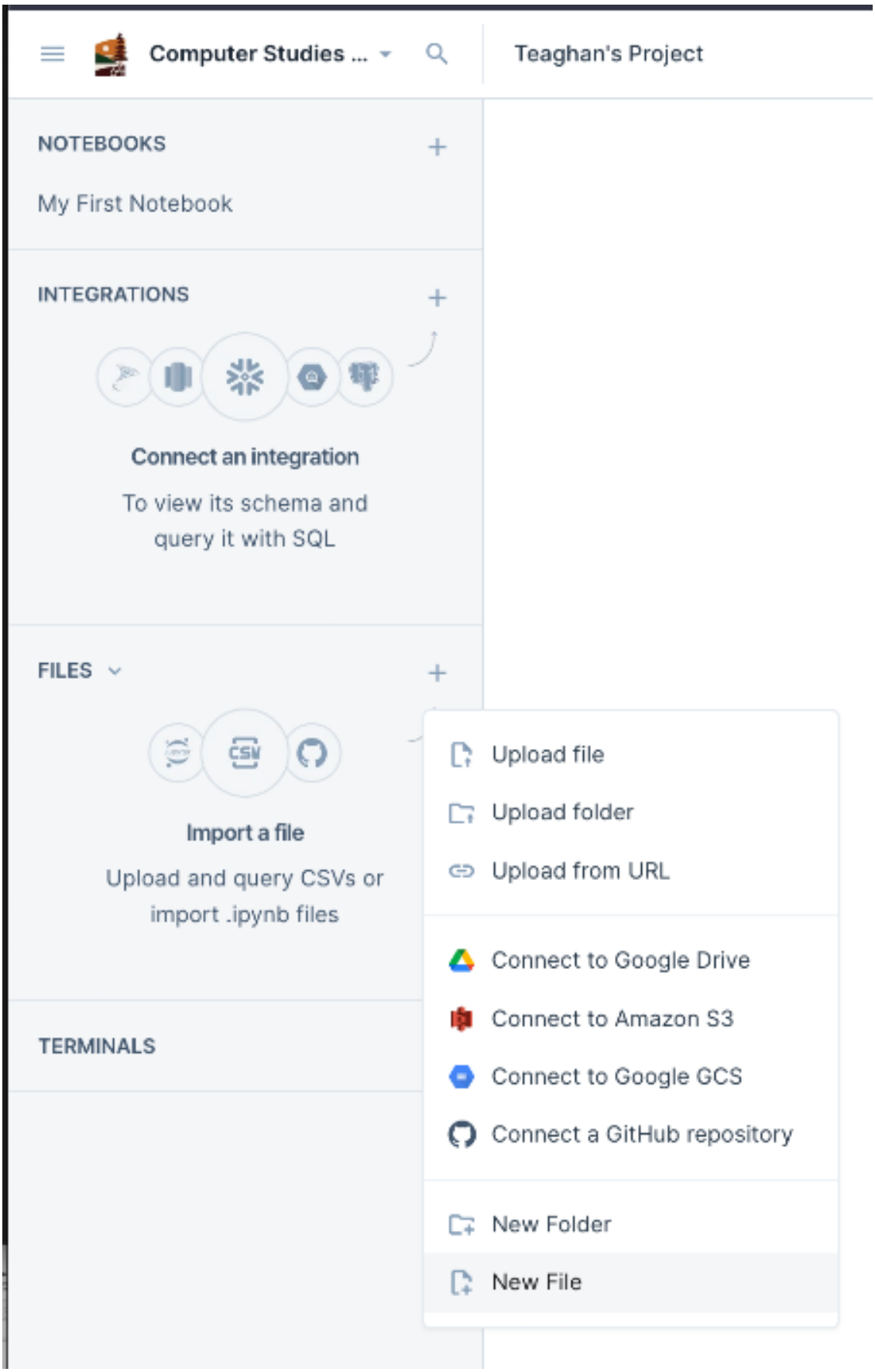
So far, we have run Python in interactive mode, which means that you have been interacting directly with the interpreter using notebooks.

Notebooks are excellent place to experiment with small code snippets and create visualizations, but every programmer needs to be able to edit and save script files to run programs again and again.

JupyerLab and Deepnote both include a **file editor**, which gives you the ability to write and execute Python programs using script mode. The built-in file editor also includes several features, like code completion and automatic indentation, that will speed up your coding workflow.

5. **Open the file editor in your project by clicking on the + beside *Files* in the left-hand panel. Provde the file with the name `helloWorld2.py`. A blank window will appear for you to type your program's code into.**

Note that the `.py` file extension is for python script files and you must use this for all of your python scripts in order for them to run.


Computer Studies ...
Teaghan's Project
NOTEBOOKS
My First Notebook
INTEGRATIONS
Connect an integration
To view its schema and query it with SQL
FILES
Import a file
Upload and query CSVs or import .ipynb files
TERMINALS
Upload file
Upload folder
Upload from URL
Connect to Google Drive
Connect to Amazon S3
Connect to Google GCS
Connect a GitHub repository
New Folder
New File

6. **Enter this program into the new file editor window.**

```
# This program says hello and asks for my name.
print('Hello world!')
print('What is your name?')
myName = input()
print('It is nice to meet you, ')
print(myName)
```

Notice that the editor will write different types of instructions using different colours. Colours can be a useful debugging tool.

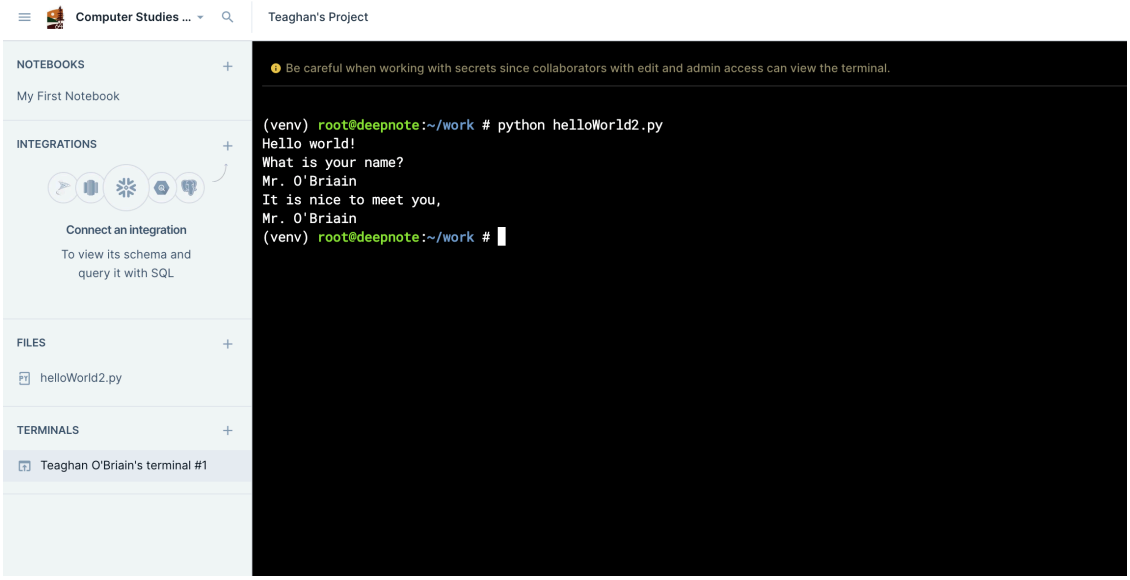
7. **Once you've entered your code, Deepnote will save it automatically.**

Now it's time to run your program. Just like on your own computer, you can open a *Terminal* to interact with the system in Deepnote. This is where you will run your scripts from.

8. **Open a Terminal from the left-hand panel:**

9. **Run your program by typing `python helloWorld2.py` followed by pressing `Enter`**

Your program will run in the shell. Enter your name when the program asks for it. When you type your name and press `Enter`, the program will greet you by name.



If you get an error message instead, go back and check your code against mine.

2.1 Comments

The first line of the helloWorld2 program is a comment:

```
# This program says hello and asks for my name.
```

A comment in Python starts with the hash character, `#`, and extends to the end of the physical line.

Comments are for programmers. They describe and explain your code to other programmers and can remind you what certain parts of your code do.

Python ignores everything after the hash mark and up to the end of the line. You can insert them anywhere in your code, even inline with other code:

[4]:
```
print('This will run.') # This won't run
```

```
This will run.
```

Comments should be short, sweet, and to the point.

Most programmers write so many programs, that they forget what some of them do. It's a programming convention to start each program with a comment that describes what the program will do.

It's also a good idea to add comments throughout your code to explain in natural language what the code is doing.

For example:

[5]:
```
minute = 30

# Compute the percentage of the hour that has elapsed
```

```
percentage = (minute * 100) / 60
print(percentage)
```

```
50.0
```

Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out **what** the code does; it is more useful to explain **why**.

This comment is redundant and useless:

[6]:
```
v = 5 # assign 5 to v
```

This comment contains useful information that is not in the code:

[7]:
```
v = 5 # velocity in meters/second
```

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read.

2.2 Keyboard Input

The fourth line of the helloWorld2 program uses the **input** function:

```
myName = input()
```

In this program, when `input()` is called, the program waits for the user to enter text. The text string that the user enters is then stored in the variable `myName`.

Python provides a built-in function called `input` that stops the program and waits for the user to type something. When the user presses Return or Enter, the program resumes, and the input function returns what the user typed as a string.

Before getting input from the user, it's a good idea to print a prompt telling the user what to type like you did in the line:

```
print('What is your name?')
```

The input function can also take a prompt as an argument. You'll learn more about arguments and functions a little later in the unit, for now it's enough to know that whatever you put inside the parentheses after input will be output to the screen as a prompt.

This means that these 2 lines of code:

```
print('What is your name?')
myName = input()
```

Can be replaced with this line of code:

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

This one line of code will display `What is your name?` on the screen, move the cursor to the next line, wait for the user to type something, when the user presses Enter, it will save their answer as a string in the variable myName.

Remember that the sequence `\n` at the end of the prompt represents a newline, which is a special character that causes a line break. It makes the user's input appear below the prompt.

10. **Edit your `helloWorld2.py` program and replace those 2 lines of code with 1 line.**

```
# This program says hello and asks for my name.
print('Hello world!')
myName = input('What is your name?')
print('It is nice to meet you, ')
print(myName)
```

2.3 Order of Operations

In the last lesson, you learned about the Arithmetic Operators `+`, `-`, `*`, `/`. Just like in mathematics, in Python when an expression contains more than one operator, the order of the evaluation depends on the **order of operations**.

PEMDAS is a useful way to remember the order:

- Parentheses
- Exponentiation
- Multiplication and Division
- Addition and Subtractions

For example:

```
[8]: 2 + 3 * 4
```

```
[8]: 14
```

```
[9]: (2 + 3) * 4
```

```
[9]: 20
```

When in doubt, use brackets to make the order obvious.

Python uses two multiplication symbols to represent exponents: `4**2` is the same as 4^2

```
[10]: 3**2
```

```
[10]: 9
```

```
[11]: 3**3
```

```
[11]: 27
```

```
[12]: 10**6
```

```
[12]: 1000000
```

2.4 String Operators

In general, you can't perform mathematical operations on strings, even if the strings look like numbers, so the following are illegal: `'2' - '1' 'eggs' / 'easy' 'third' * 'a charm'`

But there are two exceptions: `+` and `*`.

The `+` operator performs string concatenation, which means it joins the strings by linking them end-to-end. For example:

[13]:
```
first = 'throat'
second = 'warbler'
first + second
```

[13]:
```
'throatwarbler'
```

The `*` operator also works on strings; it performs repetition. For example:

[14]:
```
'Spam' * 3
```

[14]:
```
'SpamSpamSpam'
```

The use of + and * makes sense by analogy with addition and multiplication. Just as `4 * 3` is equivalent to `4 + 4 + 4`, we expect `'Spam' * 3` to be the same as `'Spam' + 'Spam' + 'Spam'`, and it is.

2.5 Debugging

Three kinds of errors can occur in a program:

- Syntax errors
- Runtime errors
- Semantic errors

It is useful to distinguish between them in order to track them down more quickly.

Syntax Error: Syntax refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so `(1 + 2)` is legal, but `8)` is a syntax error.

If there is a syntax error anywhere in your program, Python displays an error message and quits, and you will not be able to run the program. During the first few weeks of your programming career, you might spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.

Runtime error: The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you will see in the first few lessons, so it might be a while before you encounter one.

Semantic error: The third type of error is "semantic", which means related to meaning. If there is a semantic error in your program, it ill run without generating error messages, but it will not do the right thing. It will do something else. Specifically, it ill do what you told it to do.

Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

2.6 Glossary

This list may help you to understand the terms used in this lesson. Please read it carefully. - **assignment:** A statement that assigns a value to a variable. - **concatenate:** To join two operands end-to-end. - **comment:** Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program. - **expression:** A combination of variables, operators, and values that represents a single result value. - **evaluate:** To simplify an expression by performing the operations in order to yield a single value. - **keyword:** A reserved word that is used by the compiler to parse a program; you cannot use keywords like if, def, and while as variable names. - **operator**: A special symbol that represents a simple computation like addition, multiplication, or string concatenation. - **operand:** One of the values on which an operator operates. - **Order of operations:** The set of rules governing the order in which expressions involving multiple operators and operands are evaluated. - **state diagram: A graphical representation of a set of variables and the values they refer to. -** statement:** A section of code that represents a command or action. So far, most of the statements we have seen are assignment statements. - **variable:** A name that refers to a value.

Next you should do Challenge 24.

[ ]: