Chapter 2: Getting Started with Python

Contents

Chapter 2: Getting Started with Python 1
How Does Typing in a Code Tutorial Help with Learning? 3
Practice, Practice, and More Practice - Neural Plasticity 3
How Popular is Python? 4
The Story Behind the Name 4
Python Goals 4
Why Python? 4
Interpreted vs Compiled 5
Free and Open Source 5
Portable 6
Object-Oriented Programming 6
The Zen of Python 6
  Keyboard Shortcuts 7
Assignment 1.0: Organization and Folders 7
Tutorial 2.1: Hello World! 8
Tutorial 2.2: Band Name Creator 9
  Requirements 9
    Algorithm 9
    Pseudocode 9
    TODO 9
Variables 11
Tutorial 2.3: Inches to Centimeters 12
  Input - Process - Output 13
    Input 13
    Process 14
    Output 14
Tutorial 2.3: Average 14
    Requirements 14

Comments 15
Title Block 15
Get Input 15
Calculate 15
Display Output 16
The Concept of Type 16
Strongly Typed Languages 16
Python is Not Strongly Typed 16
Declaration of Variables 16
Don't Use the Same Name for Two Variables 17
A More Subtle Danger 17
Variable Names and Keywords 17
snake_case Variable Names 17
Literal Constants 18
Constants 19
Math Operators and Operands 19
Expressions 20
Order of Operations 21
Remainder or Modulus Operator 21
Strings 22
Change Case 23
Escape Sequences 24
The Newline (\n) Character 24
Tutorial 2.4: Savings Account 24
Requirements 24
Comments 25
Choosing Mnemonic (Easy to Remember) Variable Names 26
Debugging 27
My Code Isn’t Working 29
Assignment 1.1: Susie’s Square Calculator 29
Glossary 30
Assignment Submission 31

How Does Typing in a Code Tutorial Help with Learning?

Typing in a code tutorial can significantly enhance learning in several ways:

- **Active Engagement**: Typing the code yourself forces you to actively engage with the material, rather than passively reading or watching. This active participation helps reinforce the concepts being taught.
- **Muscle Memory**: Repeatedly typing code helps build muscle memory, making it easier to recall syntax and structure when you write code independently.
- **Error Handling**: When you type code, you're likely to make mistakes. Debugging these errors helps you understand common pitfalls and how to resolve them, which is a crucial skill for any programmer.
- **Understanding**: Typing out code allows you to see how different parts of the code interact. This deeper understanding can help you apply similar concepts to different problems.
- **Retention**: Studies have shown that actively doing something helps with retention. By typing out the code, you're more likely to remember the concepts and techniques.

 **Red light: No AI**

Time required: 120 minutes

NOTE: Please complete all tutorials and assignments.

Practice, Practice, and More Practice - Neural Plasticity

Practice makes permanent.

We are going to provide you with a lot of information. You will need to put your brain in gear and write a lot of programs to retain the information. Very few people become expert athletes or musicians without a lot of practice. Similarly, very few people become expert programmers without a lot of practice.

A little bit of programming each day will strengthen the neural connections in your brain. Consistent daily practice literally re wires your brain. This is called **neural plasticity**.

If you want to learn to program, plan to spend a lot of time in front of your computer, not just reading, but programming as well.

Neural pathways are **strengthened** into habits through the repetition and practice of thinking, feeling, and acting.

PRACTICE: Start your morning passionately declaring aloud your goals for the day. Declarations send the power of your subconscious mind on a mission to find solutions to fulfill your goals.

How Popular is Python?

https://pypl.github.io/PYPL.html

https://www.tiobe.com/tiobe-index/

The Story Behind the Name

Guido van Rossum, the creator of the Python language, named the language after the BBC show "Monty Python's Flying Circus". He doesn't particularly like snakes that kill animals for food by winding their long bodies around them and crushing them.

Python Goals

In 1999, Guido van Rossum defined his goals for Python:

- an easy and intuitive language just as powerful as those of the major competitors;
- open source, so anyone can contribute to its development;
- code that is as understandable as plain English;
- suitable for everyday tasks, allowing for short development times.

Why Python?

The Python programming language in and of itself is not a particularly interesting programming language. To a programmer accustomed to compiled, strongly typed programming languages such as C, C++, and Java, the Python programming language seems to be a little "sloppy", “loosey goosey” and fraught with pitfalls. In the grand scheme of things, Python is an **extremely important** programming language.

The importance of Python derives not from the language itself, but from the hundreds of independent open-source Python libraries that have been developed by others in such areas as image manipulation, networking, plotting and graphics, engineering and scientific programming, web development, gaming, cryptography, database, geographic information systems (GIS), audio, music, embedded devices, robotics, CyberSecurity, presentation, XML processing, etc. It is hard to come up with a programming application area where someone has not already supplemented the basic Python programming language with an open-source library designed for use in that application area.

Many of those libraries are Python wrappers for compiled C code providing speed and efficiency not normally associated with interpreted languages such as Python.

Interpreted vs Compiled

A program written in a compiled language like C or C++ is converted from the source language i.e. C or C++ into a language that is spoken by your computer (binary code i.e. 0's and 1's) using a compiler with various flags and options. When you run the program, the linker/loader software copies the program from hard disk to memory and starts running it.

Python does not need compilation to binary. You run the program directly from the source code. Internally, Python converts the source code into an intermediate form called bytecode, translates this into the native language of your computer and then runs it. All this makes using Python much easier since you don't have to worry about compiling the program, making sure that the proper libraries are linked and loaded, etc. This makes your Python programs much more portable. You can just copy your Python program onto another computer, and it just works!

A compiled program built in C++ or something similar runs faster, as it is designed for the architecture of the device. The big advantage of Python is that it is very portable to hundreds of different devices.

Free and Open Source

Python is an example of a FLOSS (Free/Libré and Open-Source Software). You can freely distribute copies of this software, read its source code, make changes to it, and use pieces of it in new free programs. FLOSS is based on the concept of a community which shares knowledge. This is one of the reasons why Python is so good - it has been created and is constantly improved by a community who just want to see a better Python.

Portable

Due to its open-source nature, Python has been ported to (i.e. changed to make it work on) many platforms. All your Python programs can work on any of these platforms without requiring any changes at all if you are careful enough to avoid any system-dependent features.

You can use Python on GNU/Linux, Windows, FreeBSD, Macintosh, Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE and PocketPC!

You can even use a platform like Kivy to create games for your computer and for iPhone, iPad, and Android.

Object-Oriented Programming

Python supports procedure-oriented programming as well as object-oriented programming. In procedure-oriented languages, the program is built around procedures or functions which are reusable pieces of programs. In object-oriented languages, the program is built around objects which combine data and functionality. Python has a very powerful but simplistic way of doing OOP, especially when compared to big languages like C++ or Java.

The Zen of Python

Run the following code to find out about the Python mission. This includes good program design and coding principles.

At a Python prompt, type the following code. You should see the following The Zen of Python.

```
# Display the Zen of Python
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
```

```
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Keyboard Shortcuts

The following keystrokes work in most Windows programs and can really speed up your work.

The most important keyboard shortcut is **CTRL+S** which saves the file. Get used to doing that on a very regular basis. Save early, Save often.

Keystroke	Result
CTRL+S	Save the file
CTRL+C	Copy selected text
CTRL+X	Cut selected text
CTRL+V	Paste
CTRL+Z	Undo the last keystroke or group of keystrokes
CTRL+SHIFT+Z	Redo the last keystroke or group of keystrokes
F5	Run module

Assignment 1.0: Organization and Folders

As we create programs, we want to be able to find them.

1. Create a folder named Spring 25.
2. Create a folder named Python.
3. Create a folder for each week under your Python folder.

Take a screenshot of your folder structure to include when submitting this assignment.

Tutorial 2.1: Hello World!

In this and the following tutorials you type in and run sample programs. Learning to program requires typing in code. You are not expected to understand everything yet. The purpose of these tutorials is to give you a taste of what a Python program looks like.

Hello World is the traditional first program to create in any programming language. It confirms that your programming environment is setup properly.

When you open a folder with VSCode, a project is automatically created. This will become more important when we start working with more than one file.

1. In File Explorer → Right Click on the on your Week 1 folder →
    a. Windows 11: Show more options → Open with Code
    b. Windows 10: Open with Code.
2. In **Visual Studio Code** → **File** → **New File** → Python file to create a new Python program file.
3. To save a Python program file, go to **File** → **Save As** → name the file **hello_world.py**
4. Click **Save**.

We are going to add on to the Hello World program we created when we installed Python. We will create this program in the style we will use for this class.

5. Type in the following code.

```
"""
    Name: hello_world.py
    Author:
    Created:
    Purpose: My first Python program
"""
```

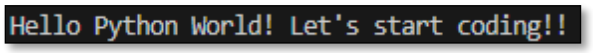
This is a header for the program. It is a multiline comment letting whoever is reading the code know what the program is about. Python ignores comments.

```
# Print the literal string Hello World!
print("Hello Python World! Let's start coding!!")
```

The first line is a comment about what the second line does.

The Python program prints the contents of the variables to the console.

Example run:



Tutorial 2.2: Band Name Creator

Let's create a program that gets input from the user. Who doesn't want a cool band name?

Requirements

We are going to go through a development process to create a program that takes two string inputs from the user and assembles those inputs into a Band Name.

Algorithm

Every program you write solves some type of program. An algorithm is the method of solving that problem.

Pseudocode

Pseudocode is one way of planning out your program in simple natural language.

Pseudocode is the first step of translating the algorithm to the programming language. The more you figure out before you code, the easier it is to code an efficient and elegant program. An elegant, simple program is like a work of art.

```
Create a greeting for your program.
Ask the user for the city that they grew up in.
Ask the user for the name of a pet.
Combine the name of their city and pet.
Display their band name.
```

TODO

A **TODO** list is another way to plan out your program.

Copy and paste the following code to get started with this program.

```
# Anything after the # sign is a comment. Python ignores this line.
# TODO: Create a greeting for your program.

# TODO: Ask the user for the city that they grew up in.

# TODO: Ask the user for the name of a pet.

# TODO: Combine the name of their city and pet.

# TODO: Display the band name.
```

1. Create a Python program named: **band_name_generator.py**
2. Type in the following code.

```
"""
    Name: band_name_generator.py
    Author:
    Created:
    Purpose: Get string inputs from user, concatenate them together
"""
```

This is a multiline comment that describes the program.

```
# TODO: Ask the user for the city that they grew up in.
city = input("What's the name of a city you grew up in: ")

# TODO: Ask the user for the name of a pet.
pet = input("What's your pet's name: ")
```

Get two string inputs into two string variables from the user. A variable is a named storage location in dynamic memory or RAM.

```
# TODO: Combine the name of their city and pet.
band_name = city + pet
```

String concatenation refers to the process of combining multiple strings into a single string. This can be achieved using the **+** operator.

```
# TODO: Display the their band name.
print("Your band name is: " + city + " " + pet)
```

Example run:

```
What's the name of a city you grew up in: Welcome
What's your pet's name: Goldie
Your band name is: Welcome Goldie
```

Variables

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 stored in a temporary memory location.

```
print("  Variable ")
print(" +---------+")
print(" | Address |")
print(" +---------+")
print(" |  Value  |")
print(" +---------+")
```

```
 Variable
+---------+
| Address |
+---------+
|  Value  |
+---------+
```

You can also think of a variable is a pigeonhole in memory, which has a nickname, where you can store values.

You can later retrieve the values that you have stored there by referring to the pigeonhole by its nickname (identifier). You can also store a different value in the pigeonhole later if you desire.

Most of the time we will use these three variable types.

- **int** - an integer is a whole number: **2**
- **float** - is a floating-point or decimal number: **2.23**
- **string** - is a sequence of characters, just a bunch of words.

Tutorial 2.3: Inches to Centimeters

Getting numeric input from the user involves another method of input. All input from the keyboard comes in as a string. To convert the string to a number we use the **float()** or **int()** function.

The **float()** function in Python converts a number or a string that represents a floating-point number into a floating-point value. It returns a floating-point number, allowing numeric operations with decimal points.

```
# Get string input from the user
inches = input("Enter inches: ")
# Convert a string to a float number like 3.114
inches = float(inches)
```

In Python, we typically combine both of those operations in a single line.

```
# Get a string from the user, store a decimal number in the inches variable
inches = float(input("Enter inches: "))
```

1. Create a new Python program named **inches_to_cm.py**.
2. Copy and paste the following TODO code to get started with this program.

```
"""
    Name: inches_to_centimeters.py
    Author:
    Created:
    Purpose: Convert Inches to Centimeters
"""

# TODO: Print a nice title for our program

# TODO: Get inches input from user, cast to float

# TODO: Convert inches to centimeters

# TODO: Display the centimeter result
```

3. Type in the following code.

```
"""
    Name: inches_to_centimeters.py
    Author:
    Created:
    Purpose: Convert Inches to Centimeters
"""
```

This is a multiline comment that gives information about the program. It is helpful to have a descriptive header in a program.

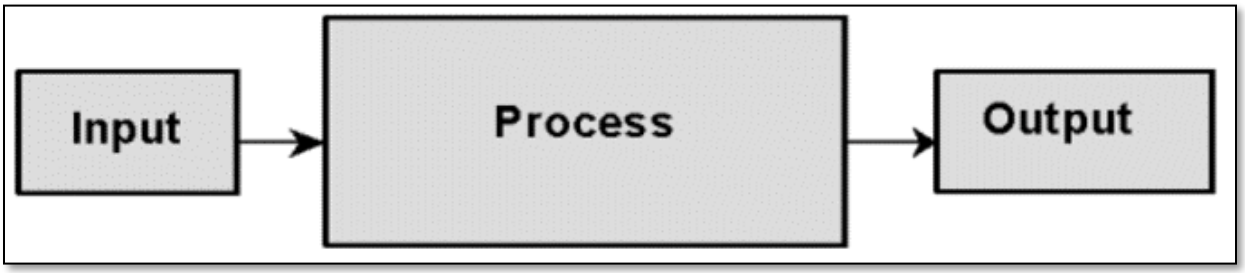
```
# TODO: Print a nice title for our program
print("-----------------------------------------")
print("|     Inches to Centimeters Converter     |")
print("-----------------------------------------")
```

The first line is a single comment to let another programmer know what is going on.

The next lines print the program title to the console.

Input - Process - Output

All programs have three basic components.



Input

```
# TODO: Get inches input from user, cast to float
inches = float(input("Enter inches: "))
```

The first line is a comment describing what happens in the second line.

The next line is an input function. Its job is to ask the user to type something in and to capture what the user types. The part in quotes is the prompt that the user sees. It is called a string and will appear to the program's user exactly as it appears in the code itself.

We use the **float()** function to convert the incoming string to a float. All input from a keyboard is a string.

We use the assignment operator **=** to assign the float value to the **inches** variable.

Process

The process section is where the program does a calculation of some kind.

```
16   # TODO: Convert inches to centimeters
17   centimeters = inches * 2.54
```

This line uses the standard math formula to convert centimeters to inches. Notice that the = is different in programming than in math. It is called the assignment statement. The calculations or values on the right are assigned to the variable on the left.

Output

```
19   # TODO: Display the centimeter result
20   print(inches, "inches is equal to", centimeters, "centimeters.")
```

The last line uses the print function to print out the conversion result. Note the , (comma) sign between the variable names and the strings.

1. Run the program by pressing the F5 key or Click the run button.
2. The program will ask you to Enter inches. Type in 24 and press enter.

Example run:

```
-----------------------------------------
|     Inches to Centimeters Converter     |
-----------------------------------------
Enter inches: 24
24.0 inches is equal to 60.96 centimeters.
```

This program may seem too short and simple to be of much use. There are many websites that do similar conversions. Their code is not much more complicated than the code here.

Tutorial 2.3: Average

Requirements

This program computes the average of two numbers that the user enters.

Create a new Python program and save it as **average.py**.

Comments

Notice the lines that start with #. These are comments to help someone else understand your code. It will also help you remember what you were thinking when you wrote the code. Commenting your code is a good programming practice.

Title Block

```
"""
    Filename: average.py
    Author:
    Date:
    Purpose: Average two numbers input by user
"""
```

The Title Block in a program is part of documenting or commenting a program to make it more understandable. Please use this in each of your programs.

```
print("Calculate the average of two numbers.")
```

This is a description of the programs purposes for the end user.

Get Input

```
# Get input from user, convert string input to float
number1 = float(input("Enter the first number: "))
number2 = float(input("Enter the second number: "))
```

We need to get two numbers from the user. We get the numbers one at a time and assign each number to a variable.

Calculate

```
# Calculate average
average = (number1 + number2) / 2
```

Note the parentheses in the average calculation. This changes the order of operations. All multiplication and division are performed before any addition and subtraction. We use parentheses to get Python to do the addition first. Python follows the PEMDAS rule of operations.

Display Output

```
17    # Display average
18    print(f"The average of {number1} and {number2} is: {average}")
```

The output is displayed to the user. We are using a string formatting called fstrings. We will use fstrings to format output from this point on in this course.

fstrings allow us to control how the data is displayed.

Example run:

```
Calculate the average of two numbers.
Enter the first number: 5
Enter the second number: 2
The average of 5.0 and 2.0 is: 3.5
```

The Concept of Type

Strongly Typed Languages

One of the main differences between Python and programming languages such as Java and C++ is the concept of data type.

In strongly typed languages like Java and C++, variables not only have a name, they have a data type which must be declared before the variable can be used. The data type determines the kind of data that you can store in the pigeonhole.

It is probably more correct to say that the data type determines the values that you can store there and the operations (addition, subtraction, etc.) that you can perform on those values.

Python is Not Strongly Typed

One of the characteristics that makes Python easier to use than Java is that with Python you don't have to be concerned about the type of a variable. Python takes care of type issues for you behind the scenes. That ease of use comes with some costs attached. Performance is one of them.

Declaration of Variables

With Java or C++, you must declare variables before you can use them. Declaration of variables is not required with Python.

With Python, if you need a variable, you come up with a name and start using it as a variable.

Don't Use the Same Name for Two Variables

With Python, if you unintentionally use the same name for two or more variables, the first will be overwritten by the second. This can lead to program bugs that are difficult to find and fix.

A More Subtle Danger

A more subtle danger is that you create a variable that you intend to use more than once and you spell it incorrectly in one of those uses. This can be an extremely difficult problem to find and fix.

Variable Names and Keywords

Programmers generally choose names for their variables that are meaningful and document what the variable is used for. Although you can make up your own names for variables, you must follow these rules:

- You cannot use one of Python's keywords as a variable name.
- A variable name cannot contain spaces.
- The first character must be one of the letters a through z, A through Z, or an underscore character (_).
- After the first character you may use the letters a through z or A through Z, the digits 0 through 9, or underscores.
- Uppercase and lowercase characters are distinct (case sensitive). This means the variable name Number is not the same as number.

Variable names can be long. They can contain both letters and numbers, but they cannot start with a number.

snake_case Variable Names

Variables are typically combinations of more than one word. A common convention among programmers in Python is to use snake_case.

- The variable name is all lowercase letters.
- An underscore is placed between each word.

Here are some examples of snake_case.

```
gross_pay
hours_worked
total_hours_worked
```

If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = 'big parade' SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy' SyntaxError: invalid syntax
```

76trombones is illegal because it begins with a number. more@ is illegal because it contains an illegal character, @. But what's wrong with class?

It turns out that class is one of Python's keywords. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

Python reserves 35 keywords:

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

You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

Literal Constants

An example of a literal constant is a number like **5**, **1.23**, or a string like **'This is a string'** or **"It's a string!"**.

It is called a literal because it is literal - you use its value literally. The number 2 always represents itself and nothing else - it is a constant because its value cannot be changed. These are referred to as literal constants.

Using literal constants for numbers is called using "magic numbers". There is no way to easily tell what that number means. The number 2 could be 2 cats, 2 percent, etc. It is not a good practice to use literal constants for numbers.

Constants

A constant is a type of variable whose value cannot be changed. You can think of constants as a bag to store some books which cannot be replaced once placed inside the bag.

Constants are not enforced in Python. It is what is called a convention. This means that programmers agree to label and use constants in Python.

Constants are used to replace "magic numbers". A magic number is an unexplained value that appears in a program's code. 2 + 2 = 4 are all magic numbers as they have no meaning.

Constants may be used in many parts of the program. Instead of having to search through the program to replace a magic number in many different places, you change the constant once.

Constants use the following naming convention. All characters are capitalized with an underscore (_) in between.

```
A_SIMPLE_CONSTANT
ANOTHER_CONSTANT
SALES_TAX
```

Math Operators and Operands

Symbol	Operation	Example	Description
+	Addition	a + b	Adds two numbers
−	Subtraction	a − b	Subtracts one number from another
-	Negation	- a	Change sign of operand
*	Multiplication	a * b	Multiplies one number by another
/	Division	a / b	Divides one number by another and gives the result as a floating-point number

//	Integer division	a // b	Divides one number by another and gives the result as a whole number
%	Remainder or Modulus	a % b	Divides one number by another and gives the remainder
**	Exponent	a ** b	Raises a number to a power

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called operands.

The operators +, -, *, /, and ** perform addition, subtraction, multiplication, division, and exponentiation, as shown in the following examples.

```
20 + 32
hour - 1
hour * 60 + minute
minute / 60
5**2 (5 + 9) * (15 - 7)
```

The result of any number division in Python is a floating-point result as shown below.

```
minute = 59
minute / 60
# Output: 0.9833333333333333
```

In the following example of integer division, the value is truncated, the fractional part is left out.

```
5 // 2
# Output: 2
```

Expressions

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 (assuming that the variable x has been assigned a value):

```
x = 10
17 * x
x + 17
```

In a program, an expression all by itself doesn't do anything! This is a common source of confusion for beginners.

Order of Operations

When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. For mathematical operators, Python follows mathematical convention. The acronym **PEMDAS** is a useful way to remember the rules:

- **P**arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 * (3 - 1) is 4, and (1 + 1) ** (5 - 2) is 8. You can also use parentheses to make an expression easier to read, as in (minute * 100) / 60, even if it doesn't change the result.
- **E**xponentiation has the next highest precedence, so 2 ** 1 + 1 is 3, not 4, and 3 * 1 ** 3 is 3, not 27.
- **M**ultiplication and **D**ivision have the same precedence, which is higher than **A**ddition and **S**ubtraction, which also have the same precedence. 2 * 3 - 1 is 5, not 4, and 6 + 4 / 2 is 8, not 5.
- **O**perators with the same precedence are evaluated from left to right. The expression 5 – 3 - 1 is 1, not 3, because the 5 - 3 happens first and then 1 is subtracted from 2.

BIG NOTE: When in doubt, always put parentheses in your expressions to make sure the computations are performed in the order you intend.

Remainder or Modulus Operator

The modulus operator works on integers and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators:

```
quotient = 7 // 3
print(quotient)
# Output: 2

remainder = 7 % 3
print(remainder)
# Output: 1
```

7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another: if x % y is zero, then x is divisible by y.

You can also extract the right-most digit or digits from a number. For example, x % 10 yields the right-most digit of x (in base 10). Similarly, x % 00 yields the last two digits.

Strings

A variable that holds a sequence of characters text is called a string. Here is an example of strings and string literals.

```
a_string = "Python is Grand"
```

'Python is Grand!' is called a string literal. String literals must be enclosed by either a pair of single quotes **' '** or a pair of double quotes " ".

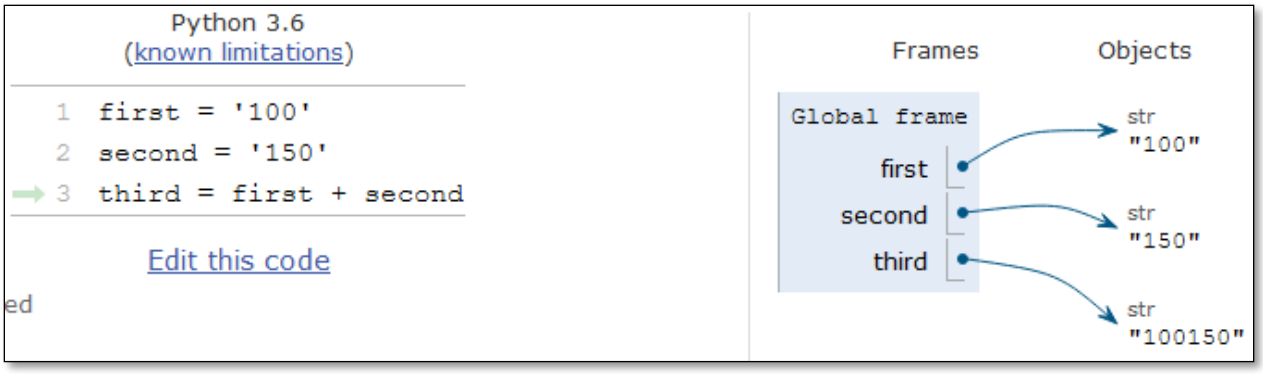
str is a variable type that stores a string. In the statement above, the string literal is assigned to the string variable **a_string**

If you want to print a single quote in the string literal, use double quotes to surround the string literal. You would do the opposite if you wanted to use double quotes in the string literal.

```
print("I'm in love with Python!")
# Output: I'm in love with Python!
```

The + operator works with strings, but it is not addition in the mathematical sense. It performs concatenation, which means joining the strings by linking them end to end. For example:

```
first = 10
second = 15
print(first + second)
# Output: 25

first = '100'
second = '150'
print(first + second)
# Output: 100150

# Notice the space after very
print('very ' + 'hot')
# Output: very hot
```



This example shows what happens when strings are concatenated.

The variable named **third** points to a third object of type **str** containing a string that was produced by using the "+" operator to concatenate the contents of two existing objects of type str. It is important to note that the contents of the third object contain the concatenation of copies of the contents of the first two objects. It doesn't simply contain pointers to the other two objects.

The * operator also works with strings by multiplying the content of a string by an integer. For example:

```
first = "Test "
second = 3
print(first * second)
# Output: Test Test Test
```

Change Case

There are 3 string operators that allow Python to change the case of a string. They can be used to change the case and assign to the same or another string. They can be used in a print statement which doesn’t change the original string.

```
message = "United States"
message = message.upper()
print(message)
message = message.lower()
print(message)
print(message.title())
```

Example run:

```
UNITED STATES
united states
United States
```

Escape Sequences

Escape sequences are special sequences of characters used to represent other characters that:

- cannot be entered directly into a string
- would cause a problem if entered directly into a string

The Newline (\n) Character

\n is an "escape character" representation of the *newline* character. It appeared in the output at the point representing the end of the first line of input. This indicates that the interpreter knows and remembers that the input string was split across two lines.

As the name implies, a newline character is a character that means, "Go to the beginning of the next line."

The newline character is sort of like the wind. You can't see the wind, but you can see the result of the wind blowing through a tree.

Similarly, you can't normally see a newline character, but you can see what it does. We must represent it by something else, like **\n** if we want to be able to see where it appears within a string.

```
print("Dick\nBaldwin")
# Output: Dick
# Output: Baldwin
```

We entered the newline escape sequence between my first and last names when we constructed the string. When the string was printed, the cursor advanced to a new line following my first name and printed my last name on the new line.

Tutorial 2.4: Savings Account

Requirements

Create a program that calculates a balance using a constant.

1. Save the program as **savings_account.py**.

```python
"""
    Name: savings_sccount.py
    Author:
    Date:
    Purpose: Demonstrate the use of constants
"""

# TODO: Declare interest rate constant
INTEREST_RATE = .045

# TODO: Get floating point input from user
balance = float(input("\nEnter your current balance: "))

# TODO: Calculate new balance
new_balance = (balance * INTEREST_RATE) + balance

# TODO Display your current interest, start by echoing user input
print(f"\nYour current balance is: ${balance:,.2f}")
print(f"Your current interest rate is: {INTEREST_RATE * 100}%")
print(f"Your new account balance with interest is: ${new_balance:,.2f}")
```

Example run:

```
Enter your current balance: 248

Your current balance is: $248.00
Your current interest rate is: 4.5%
Your new account balance with interest is: $259.16
```

Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called comments, and in Python they start with the # symbol:

```
# Compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
percentage = (minute * 100) / 60     # Percentage of an hour
```

Everything from the # to the end of the line is ignored; it has no effect on the program.

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 much more useful to explain why.

This comment is redundant with the code and useless:

```
# Assign 5 to v
v = 5
```

This comment contains useful information that is not in the code along with a better variable name.

```
# Velocity in meters/second
velocity = 5
```

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a trade-off.

Choosing Mnemonic (Easy to Remember) Variable Names

Variable names can contain letters, numbers, and the underscore.

- Variable names cannot contain spaces.
- Variable names cannot start with a number.
- Case matters—for instance, temp and Temp and TEMP are different.

If you follow the simple rules of variable naming, and avoid reserved words, you have a lot of choice when you name your variables. In the beginning, this choice can be confusing both when you read a program and when you write your own programs. For example, the following three programs are identical in terms of what they accomplish, but very different when you read them and try to understand them.

```
a = 35.0
b = 12.50
c = a * b
print(c)

hours = 35.0
rate = 12.50
pay = hours * rate
print(pay)

x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

The Python interpreter sees all three of these programs as the same, but humans see and understand these programs quite differently. Humans will most quickly understand the intent of the second program because the programmer has chosen variable names that reflect their intent regarding what data will be stored in each variable.

We call these wisely chosen variable names "mnemonic variable names". The word mnemonic means "memory aid". We choose mnemonic variable names to help us remember why we created the variable in the first place.

Debugging

At this point, the syntax error you are most likely to make is an illegal variable name, like class and yield, which are keywords, or odd~job and US$, which contain illegal characters.

If you put a space in a variable name, Python thinks it is two operands without an operator:

```
>>> bad name = 5
SyntaxError: invalid syntax
>>> month = 09
  File "<stdin>", line 1
     month = 09
             ^
SyntaxError: invalid token
```

For syntax errors, the error messages don't help much. The most common messages are SyntaxError: invalid syntax and SyntaxError: invalid token, neither of which is very informative.

The runtime error you are most likely to make is a "use before def;" that is, trying to use a variable before you have assigned a value. This can happen if you spell a variable name wrong:

```
principal = 327.68
interest = principle * rate
NameError: name 'principle' is not defined
```

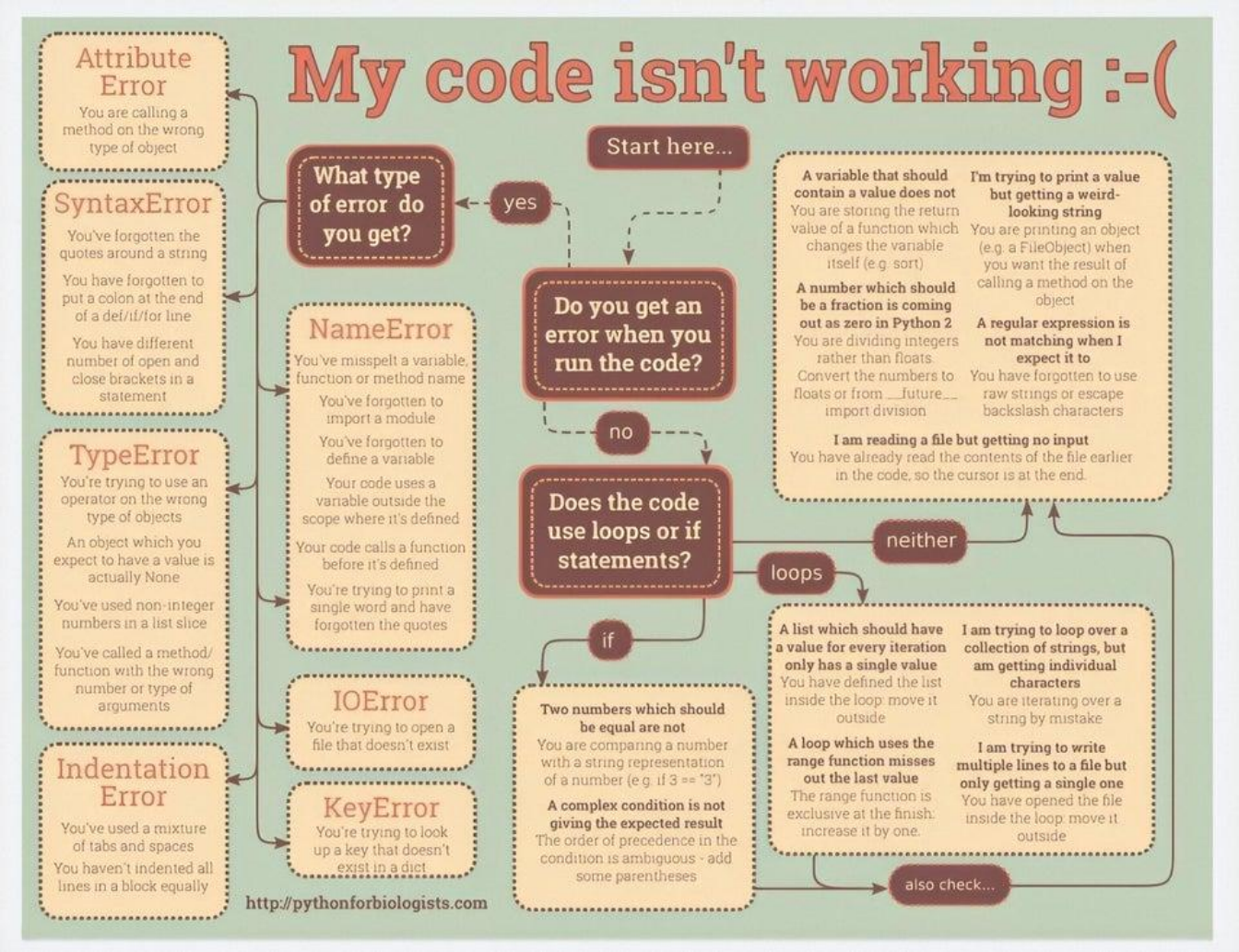
Variables names are case sensitive, so LaTeX is not the same as latex.

At this point, the most likely cause of a semantic error is the order of operations. For example, to evaluate 1/2, you might be tempted to write

```
1.0 / 2.0 * pi
```

The division happens first, so you would get /2, which is not the same thing! There is no way for Python to know what you meant to write, so in this case you don't get an error message; you just get the wrong answer.

My Code Isn't Working ☹️



Assignment 1.1: Susie's Square Calculator

Let's finish off this chapter by calculating the area and perimeter of a square.

- Formula for the area of a square:

  $$A = s^2$$

  where **A** is the area and **s** is the length of a side.

- Formula for the perimeter of a square:

  $$P = 4s$$

  where **P** is the perimeter and **s** is the length of a side.

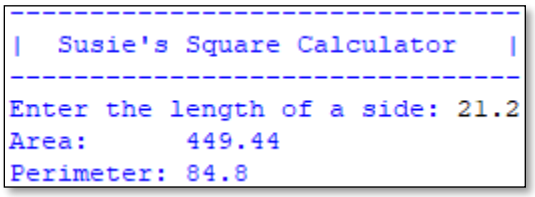
Create a Python program named **square_calculator.py**

You can pseudocode your code by using the **TODO** list shown below. This allows you to plan and start commenting your code.

1. Copy and paste the following to start your program.

```
"""
    Name: square_calculator.py
    Author: William A Loring
    Created: 07/08/22
    Purpose: Calculate the area and perimeter of a square
"""

# TODO: Print a creative title for our program

# TODO: Get length of a side from the user, convert to float

# TODO: Calculate area

# TODO: Calculate perimeter

# TODO: Display results
```

Example run:



Glossary

assignment A statement that assigns a value to a variable.

concatenate To join two operands end to end. Typically used with strings and string literals.

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.

evaluate To simplify an expression by performing the operations in order to yield a single value.

expression A combination of variables, operators, and values that represents a single result value.

floating point A type that represents numbers with fractional parts.

integer A type that represents whole numbers.

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.

mnemonic A memory aid. We often give variables mnemonic names to help us remember what is stored in the variable.

modulus operator An operator, denoted with a percent sign (%), that works on integers and yields the remainder when one number is divided by another.

operand One of the values on which an operator operates.

operator A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

rules of precedence The set of rules governing the order in which expressions involving multiple operators and operands are evaluated.

statement A section of code that represents a command or action. So far, the statements we have seen are assignments and print expression statement.

string A type that represents sequences of characters.

type A category of values. The types we have seen so far are integers (type int), floating-point numbers (type float), and strings (type str).

value One of the basic units of data, like a number or string, that a program manipulates.

variable A name that refers to a value.

Assignment Submission

1. Attach all tutorials and assignment code files.
2. Attach screenshots showing the successful operation of each tutorial program.
3. Attach a screenshot showing your folder structure with your Python programs in week 1.
4. Submit in Blackboard.