INTRO TO PROGRAMMING

# Intro to Programming Course

Welcome to the Intro to Programming course! This course is for you if you have never written a single line of code, and you are interested in learning data science and machine learning. (If you do have programming experience and are just new to the Python language, the Python course is a better fit to get started.)

In this course, you will learn how to use code to get a computer to perform certain tasks for you. Python is one of the most popular programming languages for data science, and it's the language you'll learn in this course. Once you complete this course, you'll be fully prepared to progress to the Python course, followed by the Intro to Machine Learning course.

In this tutorial, you'll see several examples of Python code. You'll get a chance to write your own code in the exercise. (If you'd like to preview the exercise, feel free to take a look now. We also provide a link to the exercise at the end of this tutorial.)

## Printing

One of the simplest (and most important!) tasks you can ask a computer to do is to print a message.

In Python, we ask a computer to print a message for us by writing print() and putting the message inside the parentheses and enclosed in quotation marks. Below, we ask the computer to print the message Hello, world!.

print("Hello, world!")

**Output:**

Hello, world!

The code is inside the box (known as a code cell), and the computer's response (called the output of the code) is shown below the box. As you can see, the computer printed the message that we wanted.

## Arithmetic

We can also print the value of some arithmetic operation (such as addition, subtraction, multiplication, or division).

For instance, in the next code cell, the computer adds 2 to 1 and then prints the result, which is 3. Note that unlike when we were simply printing text, we don't use any quotation marks.

print(1 + 2)

**Output:**

3

We can also do subtraction in Python. The next code cell subtracts 5 from 9 and prints the result, which is 4.

print(9 - 5)

**Output:**

4

You can actually do a lot of calculations with Python! See the table below for some examples.

| Operation | Symbol | Example |
| --- | --- | --- |
| Addition | + | 1 + 2 = 3 |
| Subtraction | - | 5 - 4 = 1 |
| Multiplication | \* | 2 \* 4 = 8 |
| Division | / | 6 / 3 = 2 |
| Exponent | \*\* | 3 \*\* 2 = 9 |

You can control the order of operations in long calculations with parentheses.

print(((1 + 3) \* (9 - 2) / 2) \*\* 2)

**Output:**

196.0

In general, Python follows the PEMDAS rule when deciding the order of operations.

## Comments

We use comments to annotate what code is doing. They help other people to understand your code, and they can also be helpful if you haven't looked at your own code in a while. So far, the code that we have written is very short, but annotations become more important when you have written a lot of code.

For instance, in the next code cell, we multiply 3 by 2. We also add a comment (# Multiply 3 by 2) above the code to describe what the code is doing.

# Multiply 3 by 2

print(3 \* 2)

**Output:**

6

To indicate to Python that a line is a comment (and not Python code), you need to write a pound sign (#) as the very first character.

Once Python sees the pound sign and recognizes that the line is a comment, it is completely ignored by the computer. This is important, because just like English or Hindi (or any other language!), Python is a language with very strict rules that need to be followed. Python is stricter than a human listener, though, and will just error if it can't understand the code.

We can see an example of this, in the code cell below. Python errors if we remove the pound sign, because the text in the comment is not valid Python code, so it can't be interpreted properly.

Multiply 3 by 2

**Output:**

File "/tmp/ipykernel\_18/3750420471.py", line 1

Multiply 3 by 2

^

SyntaxError: invalid syntax

## Variables

So far, you have used code to make a calculation and print the result, and the result isn't saved anywhere. However, you can imagine that you might want to save the result to work with it later. For this, you'll need to use variables.

### Creating variables

The next code cell creates a variable named test\_var and assigns it the value that we get when we add 5 to 4.

We then print the value that is assigned to the variable, which is 9.

# Create a variable called test\_var and give it a value of 4+5

test\_var = 4 + 5

# Print the value of test\_var

print(test\_var)

**Output:**

9

In general, to work with a variable, you need to begin by selecting the name you want to use. Variable names are ideally short and descriptive. They also need to satisfy several requirements:

* They can't have spaces (e.g., test var is not allowed)
* They can only include letters, numbers, and underscores (e.g., test\_var! is not allowed)
* They have to start with a letter or underscore (e.g., 1\_var is not allowed)

Then, to create the variable, you need to use = to assign the value that you want it to have.

You can always take a look at the value assigned to the variable by using print() and putting the name of the variable in parentheses.

Over time, you'll learn how to select good names for Python variables. It's completely fine for it to feel uncomfortable now, and the best way to learn is just by viewing a lot of Python code!

### Manipulating variables

You can always change the value assigned to a variable by overriding the previous value.

In the code cell below, we change the value of my\_var from 3 to 100.

# Set the value of a new variable to 3

my\_var = 3

# Print the value assigned to my\_var

print(my\_var)

# Change the value of the variable to 100

my\_var = 100

# Print the new value assigned to my\_var

print(my\_var)

**Output:**

3

100

Note that in general, whenever you define a variable in a code cell, all of the code cells that follow also have access to the variable. For instance, we use the next code cell to access the values of my\_var (from the code cell above) and test\_var (from earlier in this tutorial).

print(my\_var)

print(test\_var)

**Output:**

100

9

The next code cell tells Python to increase the current value of my\_var by 3.

To do this, we still need to use my\_var = like before. And also just like before, the new value we want to assign to the variable is to the right of the = sign.

# Increase the value by 3

my\_var = my\_var + 3

# Print the value assigned to my\_var

print(my\_var)

**Output:**

103

### Using multiple variables

It's common for code to use multiple variables. This is especially useful when we have to do a long calculation with multiple inputs.

In the next code cell, we calculate the number of seconds in four years. This calculation uses five inputs.

# Create variables

num\_years = 4

days\_per\_year = 365

hours\_per\_day = 24

mins\_per\_hour = 60

secs\_per\_min = 60

# Calculate number of seconds in four years

total\_secs = secs\_per\_min \* mins\_per\_hour \* hours\_per\_day \* days\_per\_year \* num\_years

print(total\_secs)

**Output:**

126144000

As calculated above, there are 126144000 seconds in four years.

Note it is possible to do this calculation without variables as just 60 \* 60 \* 24 \* 365 \* 4, but it is much harder to check that the calculation without variables does not have some error, because it is not as readable. When we use variables (such as num\_years, days\_per\_year, etc), we can better keep track of each part of the calculation and more easily check for and correct any mistakes.

Note that it is particularly useful to use variables when the values of the inputs can change. For instance, say we want to slightly improve our estimate by updating the value of the number of days in a year from 365 to 365.25, to account for leap years. Then we can change the value assigned to days\_per\_year without changing any of the other variables and redo the calculation.

# Update to include leap years

days\_per\_year = 365.25

# Calculate number of seconds in four years

total\_secs = secs\_per\_min \* mins\_per\_hour \* hours\_per\_day \* days\_per\_year \* num\_years

print(total\_secs)

**Output:**

126230400.0

**Note:** You might have noticed the .0 added at the end of the number, which might look unnecessary. This is caused by the fact that in the second calculation, we used a number with a fractional part (365.25), whereas the first calculation multiplied just numbers with no fractional part. You'll learn more about this in Lesson 3, when we cover data types.

### Debugging

One common error when working with variables is to accidentally introduce typos. For instance, if we spell hours\_per\_day as hours\_per\_dy, Python will error with message NameError: name 'hours\_per\_dy' is not defined.

print(hours\_per\_dy)

**Output:**

---------------------------------------------------------------------------

NameError Traceback (most recent call last)

/tmp/ipykernel\_18/142450907.py in <module>

----> 1 print(hours\_per\_dy)

NameError: name 'hours\_per\_dy' is not defined

When you see NameError like this, it's an indication that you should check how you have spelled the variable that it references as "not defined". Then, to fix the error, you need only correct the spelling.

print(hours\_per\_day)

**Output:**

24