Intro to Coding

Class 2

Overview

- Methods to reuse code/data
  - Variables and functions
- Variable data types
  - Type casting
- Creating variables
- Arithmetic and comparison operators
- String concatenation
- `print()` and `input()` functions
- Calling functions
- Libraries
- Comments

Variables vs Functions

What are variables?

- Used for **storing values** in your code
- They are also used so that these values can be **accessed later**
- Examples: keeping track of your score, current coordinates, etc.

What are functions?

- Used for **repeating certain procedures** over and over, rather than continuously copy-pasting the same code
- Examples: drawing a shape, performing certain calculations, etc.

Numerical Data Types

Integers

- Stores **whole number** values
  - Ex: 1, 1234, 90927, -9999999

Floats

- Similar function to integers, but stores **decimals**
  - Ex: 3.33, 1.0, -43.888888

Strings

Stores all kinds of text values

- `fav_food = “Pizza”`
- `birthday = “November 12”`
- `email = “support@codeup.ca”`

They are marked by a set of quotation marks “ ”

Boolean

- `toggled = True`
- `human = False`

- Has only true/false values
- True and False have to be capitalized
- Used for conditional statements (if, else, else if)

Data Types

- Data comes in different types, such as strings, integers, floats boolean, etc.
- Serve different purposes:
  - Strings for text
  - Integers for integer numerical values
  - Floats for numerical values which may not necessarily be integers
  - Boolean for yes or no (but True/False)
- Type casting:
  - Functions used to change a variable's data type
  - **int(), float(), str()**
    - Seen in "Taking Input", to get integer or floating point inputs instead of the default string.

Defining Variables

Luckily, Python doesn't care about what types your variables are when you define them. To call a variable, simply type it out:

```
x = "hello world"  # string
y = 2025  # int
z = 3.14159  # float
x = 1  # changes x to an int
```

When you try to do operations with variables, it is generally recommended to have them match (i.e. don't try to add a number with a string):

```
x = "abc"
y = 123
z = x + y
```

Results in an error

Naming Variables

All variable names must adhere to the following rules:

- No special characters (!, @, #, $, etc.), including space
- Can't begin with numbers (**1num** is bad, but **num1** is fine)
- Must start with a letter or the underscore character (myFavoritePlace, __score__)
- Capitalization matters (ie. **abc** is NOT the same as **Abc**)
- Cannot be any of the Python keywords (string, list...)

It is common practice to use all lowercase variable names with underscores in place of spaces (ie. player_score). This is called **snake case** convention

Arithmetic Operators

These are common mathematical operators used to change variables

- Addition
  - Sidenote: we can add **strings** together this way. This is called string concatenation
  - **+**
    - `X + Y`
- Subtraction
  - **-**
    - `X - Y`
- Multiplication
  - *****
    - `X * Y`
- Division
  - **/**
    - `X / Y`
  - Note: The result of the / operation **is always a float**, even if the answer can be represented as a whole number (ie. 4.0 / 2 = 2.0)

Arithmetic Operators

- Modulus
  - Returns the remainder
  - %
    - `X % Y`
- Exponentiation
  - **
    - `X ** Y` ⇒ (X^y)
- Floor Division
  - Returns division result (quotient) **rounded down**
  - //
    - `X // Y`
  - `10 // 3` returns `3` but `-10 // 3` returns `-4`

Assignment Operators

- As the name suggestions, *Assignment Operators* assign a value to a variables
- Form is arithmetic operator + a “=”
- I.e +=, -=
  - `X += 3 → X = X + 3`
  - `X -= 3 → X = X - 3`
  - `X *= 3 → X = X * 3`
  - `X /= 3 → X = X / 3`
- What do these operations have in common?

Comparison

- These are used to compare two variables. They return a Boolean result (True/False)
- Equal
  - X == Y
- Not Equal
  - X != Y
- Greater Than
  - X > Y
- Less than
  - X < Y
- Greater Than or equal to
  - X >= Y
- Less Than or equal to
  - X <= Y

Concatenating Strings

- The process of combining strings is called **concatenation**
- String concatenation follows the same format as integer addition
  - `result = st1 + st2`
- If you want to add a string to the end of an existing string, you can do this
  - `st1 += st2` (adds `st2` at the end of `st1`)
  - `st1 += “a”` (adds the character `“a”` at the end of `st1`)

Both solutions lead to the same output →

```
a = "Code"
b = "Up"
c = a + b
print(c)
```

```
a = "Code"
a += "Up"
print(a)
```

```
CodeUp
```

The print() Function Revisited

Aside from printing content in quotation marks, the **print()** function can print content using variables

```
name = "codeup"
print(name)
```

Excluding quotation marks in a print statement causes Python to print the variable indicated inside (ie. this statement prints the **name** variable, not the word “name” in text)

Important: make sure the variables are declared **before** you print, otherwise it will result in an error

Concatenating in a print() Statement

- The print statement supports concatenation
- You can directly concatenate within a print statement without storing it in a variable
- **Remember:** not all modifications need to be stored in an intermediary variable, oftentimes you can just make the modifications within the print statement
    - It's good practice to create only necessary variables for cleaner code

```
name = "CodeUp"
print("Hello, " + name + "!")
```

```
Hello, CodeUp!
```

Printing non-strings

- If you want to **only** print **one** non-string, you can directly print it

```
x = 10
print(x)
```

```
x = True
print(x)
```

- If you want to **concatenate** non-strings in a print statement, they must be **casted** to a **string** first using `str(var_name)`

```
x = 10
print("x = " + str(x))
```

```
x = 10
```

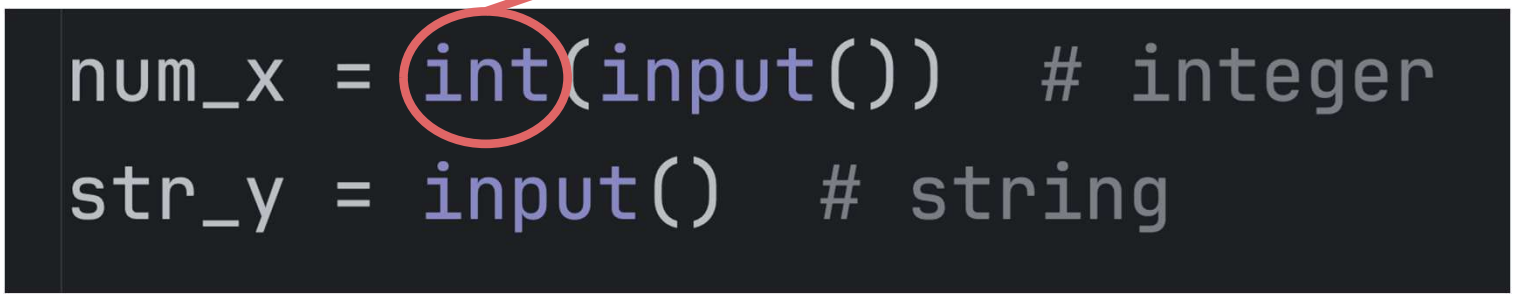
```
x = 10
y = True
print(str(y) + " " + str(x))
```

```
True 10
```

The input() Function

- To get a user input, the most basic function is using `input()`
- This automatically stores the value as a **string** if the data type is not specified
- To store the input to a variable, simply define your variable as equal to the input

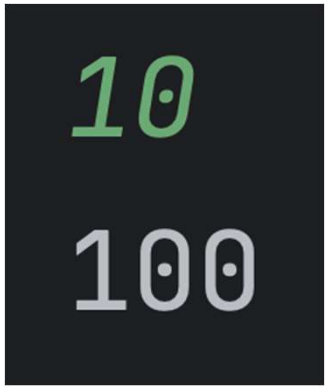
Changes variable type from
a string to an integer

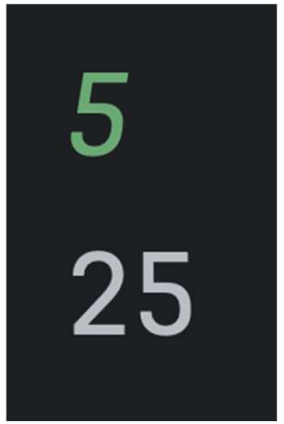


Integer Arithmetic Activity

Write a program that prompts the user to input an integer. The program should print the resulting integer after it has been squared.

Note: green text indicates user input





Solution

- Remember to cast `num` to an integer when taking it as input
  - We cannot perform arithmetic on non-integers
- You do **not** need to cast to string, as you are not using concatenation
- Not[illegible]low

```
num = int(input())
print(num ** 2)
```

Concatenation Activity

Look at the following interaction between the user and the program and figure out the password structure based off the two examples. Then, write a program that does the same function.

Note: string inputs are allowed to have spaces, though it cannot span multiple lines

```
Enter a name: Bob
Enter a city: Toronto
Enter a shape: Square
Your special password is: Toronto!!!Bob@@@Square###
```

```
Enter a name: Code Up
Enter a city: New York City
Enter a shape: Circle
Your special password is: New York City!!!Code Up@@@Circle###
```

Solution

- The password structure is:
  - `city + “!!!” + name + “@@@” + shape + “###”`
- All three variables are taken through regular input, and then printed using string concatenation
- No type casting required as they are all string variables

```
name = input("Enter a name: ")
city = input("Enter a city: ")
shape = input("Enter a shape: ")
print("Your special password is: " + city + "!!!" + name + "@@@" + shape + "###")
```

Live Activity

- We will be writing some code involving various expressions and data types
- It's **YOUR** job to determine what the output will be!

Calling Functions

Functions have to be called to be used:

`function_name(parameters)`

```
max(1, 2) # returns 2, the maximum value
```

Sometimes, functions require more than one parameter:

`list.index(element)` requires a **list**, and the **target element** to find

Calling Functions

Functions that return a value must be stored or printed immediately, or else that value **disappears**.

```
max(1, 2) # 2 is not assigned to anything, so it disappears
minimum = min(1, 2) # 1 is assigned to 'minimum'
print(minimum) # 1 is stored in 'minimum', and thus can be called
```

```
Prints 1
```

Max() and Min()

- Both are functions that take in 2 arguments and either return the *maximum* or *minimum* of the two

- `max(a, b)` → returns the larger value of a and b
  - e.g., `max(3, 4)` returns 4, `max(20, 3)` returns 20
- `min(a, b)` → returns the smaller value of a and b
  - e.g., `min(3, 4)` returns 3, `min(20, 3)` returns 3

- Applications:
  - `health = min(health + 10, 100)` prevents the user's health from exceeding 100
  - `health = max(health - 10, 0)` prevents the user's health from being negative
  - `x_pos = min(x_pos + 10, width)` prevents the user from going out of bounds

Libraries

- Libraries are collections of **modules** containing bundles of code that are used very often
  - These bundles are typically high utility and stored in **functions**, **classes**, and **variables** that can be called quickly and efficiently
  - e.g. `randint` function from random library
- Imported with the import statement at the top of your code by convention
  - Ex: `import random`

Libraries

Frequently Used Libraries

- Random

```
import random
print(random.randint( a: 1, b: 10))
```

- Math

```
import math
print(math.sqrt(16))
```

- pygame

- We can use functions like square root without writing all the code ourselves every single time, making coding easier and faster
- **Note:** if you want to call a specific function from the library, you must type the library name first (as shown in the images to the left)

Random Number Generator

- Import the built-in random library with `import random`
- randint:
  - `random.randint()` function will generate a random number within a given range
    - Note: the first number must be smaller than or equal to the second
  - `random.randint(1,10)`
    - Generates random number from 1 to 10, inclusive

```
1  import random
2
3  num = random.randint(a:1, b:10)
```

Random Number Generator Activity

Write a program that takes in two positive numbers (inputted by the user). Let `lar` represent the larger of the two numbers. The program should print a random integer from `lar` to `2 * lar`, inclusive.

The program generated a random number from 10 to 20, inclusive

Solution

- Import the random library at the top
- Take in input for both integers, then store the larger one in a variable
- Print the randomly generated integer

```
import random

num1 = int(input())
num2 = int(input())
lar = max(num1, num2)
print(random.randint(lar, 2 * lar))
```

Comments

- Comments are snippets of text that the program **does not execute**
  - In other words, comments are treated as *plain text*
- They can be used in many ways:
  - **Explaining** the code
  - Improving the **readability** of the code
  - **Stopping execution** of certain lines when testing

Comments

- Single line comments
  - Add a hashtag(#) and the rest of the line becomes a comment

```
#This is a comment
print("Hello World") #This is a comment too!
```

  - Note: python still prints “Hello World”
- Multiline comments
  - Method 1: Add a hashtag before every line

```
#This comment
#is on 2 lines!
```

  - Method 2: Add three double quotes(") or three single quotes(') before and after the comment (*Python ignores strings that are not assigned to a variable*)

```
"""
This comment
is on 2 lines!
"""
```

Homework

Write a program that takes in two numbers (inputted by the user), then adds the two numbers. The sum should range from 0 to 10, inclusive. If it is not, round to the nearest boundary (ie. -3 rounds to 0, 17 rounds to 10). The program should print the sum before and after rounding.

```
3
-10
Sum before rounding: -7
Sum after rounding: 0
```

```
7
8
Sum before rounding: 15
Sum after rounding: 10
```

```
1
2
Sum before rounding: 3
Sum after rounding: 3
```