### Workshop Details

# Intermediate Workshop to Python Programming Building the Foundation for Coding Success

Ricardo Chin

January 10, 2024



#### **Presentation Overview**

### Data Types: Deeper Dive

Previously in Python Programming —



- int: Whole numbers without decimal points
- float: Numbers with decimal points
- bool: Represents the truth values True or False
- NoneType (None): Represents absence of a value (or null)
- string: Ordered sequence of characters

#### **Collections**

- list: Ordered and mutable sequence of elements
- tuple: Ordered and immutable sequence of elements
- dict: Unordered collection of key-value pairs



### NoneType

#### NoneType

- None is a Singleton there is only ever a single instance of it inside a running Python program
- Multiple variables may refer to that same instance

#### Comparisons using Keyword "is"

Keyword is checks whether two names refer to the same object

```
1 a = [1, 2]

2 b = a

3 x = [1, 2]

4

5 a == b # True

6 a is b # True

7 a == x # True

8 a is x # False
```

As None is a singleton, we can check for it via is None

```
if a is None:
    print("a is None")
```

#### BoolType

#### BoolType

- The bool type is a built-in data type representing truth values
- It has two possible values: True and False

Booleans are a subset of integers (subclass of int) where True behaves as 1 and False as 0 in numerical contexts

False + True # 1

#### **Numbers**

#### **Operations with Numbers**

- Integer Division: 10 // 3 = 3
- Remainder: 10 % 3 = 1
- Exponentiation: 2 \*\* 3 = 8

#### **Numbers**

#### **Operations with Numbers**

- Integer Division: 10 // 3 = 3
- Remainder: 10 % 3 = 1
- Exponentiation: 2 \*\* 3 = 8



### **Underscores in Numeric Literals for Enhanced Readibility**

- Revenue: 1000000000
- Revenue: 1\_000\_000\_000



### Integer Type Representations

#### **Integers**

- 1 Python supports integers of arbitrary size, allowing representation of very large numbers
- 2 It also supports different numeral systems
  - Decimal a = 42
  - Binary b = 0b101010
  - Octal c = 0.052
  - Hexadecimal d = 0x2a
  - Conversion from a string in binary to an integer e = int ('101010', 2)

#### **Tip** — Maximum Size of Integers on the Current System

```
import sys
```

```
print(sys.maxsize) # Maximum size
```

### Float Type Representations

#### **Integers**

- 1 Floating-point numbers in Python use 64 bits
  - Numbering a = .12 or b = 2.55
  - Scientific Notation c = 6e23
  - Special Values d = float('nan') or e = float('inf')

type	range	signi- ficant digits*	exponent fraction (10 bit)	type	composed of
float16	±(6.0*10 <sup>-8</sup> 65504)	3	1bit 5bit 10bit	_	_
float32	±(1.4*10 <sup>-45</sup> 3.4*10 <sup>38</sup> )	6	1bit 8bit 23bit	complex64	two float32's
float64	±(4.9*10 <sup>-324</sup> 1.8*10 <sup>308</sup> )	15	1bit 11bit 52bit	complex128	two float64's
float128**	±(3.7*10 <sup>-4951</sup> 1.1*10 <sup>4932</sup> )	18	1bit 15bit 64bit	complex256	two float128's

Figure: Based on IEEE 754 — Standardized Floating-point Arithmetic

### Float Type Representations

### **Warning!**

Floating-point numbers, while versatile, can't perfectly represent all real numbers. This limitation leads to rounding errors, causing some numbers to be approximations rather than precise representations

### Float Type Representations

### Warning!

Floating-point numbers, while versatile, can't perfectly represent all real numbers. This limitation leads to rounding errors, causing some numbers to be approximations rather than precise representations



#### In the decimal system:

- 1 Fractions like 1/3 and 1/7 can't be represented exactly
- **2** Constant like  $\pi$  isn't fully representable without approximation In binary floats:
  - 1 Decimals like 1/2 and 1/10 can't be precisely represented
  - 2 Fractions like 1/3 and even suffer from approximation

### Float Rounding Errors

#### **Warning!**

As a consequence to not being able to perfectly represent all real numbers, float numbers will lead to rounding error mismatches

#### **Example 1 — Precision Limitations**

1 Computing  $\pi$  +  $\pi$  might yield 6.2 when using decimal numbers with a precision of 2, whereas a more precise result would be 6.3

#### Example 1 — Arithmetic Precision

1 Simple addition like 0.1 + 0.2 might oddly evaluate to  $\approx$  0.300000000000000004 due to limitations in 64-bit floats

```
1 0.1 + 0.2 == 0.3 # Returns False if float assigned
2 3 import math # tolerance = 1e-09
4 math.isclose(0.1 + 0.2, 0.3) # Returns True
```

### Complex Type and Augmented Assignment

A complex number is a numerical type used to represent numbers that have both a real part and an imaginary part: a = 1 + 2j

Let us increment the real part (1) of the variable a

$$a = a + 1$$
 or  $a + = 1$ 

#### **Augmented Assignment**

This operation means "add 1 to the current value of *a* and assign the result back to *a* 

Calculation:

$$a = 1 + 2j + 1$$

Result:

$$a = 2 + 2i$$

Other operations include: -=,  $\star=$ , ...

### **Character Encodings**

Character encodings are used to represent characters in a form that computers can understand and manipulate — mapping characters to bit sequences

### **Types of Character Encodings**

- 1 ASCII (American Standard Code for Information Interchange)
  - Encodes the first 128 Unicode characters using 7 bits, covering basic English characters, digits, and symbols
  - Represents characters like 'A', '!', '\$', space, and line breaks
- 2 Latin1 (ISO 8859-1)
  - Extends ASCII to encode the first 256 Unicode characters using 8 bits
  - Adds additional characters like 'ä', 'á', 'β', '§', etc
- 3 UTF-8, UTF-16, UTF-32
  - Encode the entire Unicode character set
  - UTF-8, a popular encoding, uses variable-width encoding

### **Character Encodings**

Examples in ASCII / Latin1 / UTF-8:

Character	Byte Representation	
!	00100001	
A	01000001	
Line Feed — Line Break — "\n"	00001010	

#### Examples in Latin1:

Character	Byte Representation	
Ä	11000100	

#### Examples in UTF-8:

Character	Byte Representation
Ä	11000011 10100100
Ü	11110000 10011111 10011001 10000010

### Strings

Strings represent sequences of Unicode characters, allowing the manipulation and representation of text data



- 1 String Literals Representations of strings in Python
  - Single quotes: a = 'test'
  - Double quotes: b = "test"
- Multi-line String Literals Multi-line representation

```
a = """this
is a multi-line
string literal """
```

Sequences — a = "He said:\n\"Hi!\""
\n for line feed or line break!



### Strings

If there is no need to use any escape sequences in a string

```
path = r"C:\documents\course\news.txt"
```

Handy when writing directory paths and regular expressions

### **Useful String Methods**

- .lower() and .upper()
- .startswith(...) and .endswith(".xlsx")
- .center (10) centered in 10 chars
- .ljust(10) left justified or .rjust(10) right justified
- .strip() removes leading and trailing spaces
- .split (' ') splits a string into a list of substrings
- ' '.join(list) join a list of strings into a single string

### String Exercises

#### **Exercises**



1 Later

### String Formatting

String formatting allows for the inclusion of values within strings

```
name = "Ricardo"

# Concatenation

greeting = "Hello, " + name + "!"

# f-string (formatted string literals)

greeting = f"Hello, {name}!"
```

There are other formatting ways which are currently a bit obsolete

```
city, temperature = 'Graz', 5.7
veather in %s: %f°C' % (city, temperature)
'weather in {0}: {1}°C'.format(city, temperature)
'weather in {}: {}°C'.format(city, temperature)
'weather in {c}: {t}°C'.format(c=city, t=temperature)
f'weather in {city}: {temperature}°C' # fstring pref
```

### Format Specifications

If we want to specify the format value itself — ie, .4g or .4f

```
# Four decimal places after the decimal point
print(f"Pi is {math.pi:.4f}") # Output: Pi is 3.1416

# Four significant digits
print(f"Pi is {math.pi:.4g}") # Output: Pi is 3.142
```

If we want to specify the sentence alignment

```
first_name, last_name = "Ricardo", "Chin"

# Right-aligned (total width 8 characters)
print(f"{first_name:>8}") # Output: " Ricardo"
print(f"{last_name:>8}") # Output: " Chin"
```

String Formatting Reference — Hyperlink

### Format Specifications

#### **Exercise**

 Create a program that formats a set of names and associated floating-point numbers representing current spare money, finds longest name, returns the names aligned to the right (longest name) and the spare money with 1 floating point

### Format Specifications

#### **Exercise**

 Create a program that formats a set of names and associated floating-point numbers representing current spare money, finds longest name, returns the names aligned to the right (longest name) and the spare money with 1 floating point

```
# Find the length of the longest name
longest_name = max(len(name) for name, _ in data)

# Aligned to longest name and spare money with .1f
for name, value in data:
    print(f"{name:>{longest_name}}{value:.1f}")
```

### Lists

### **Interesting References**

#### **Books**

- Automate the Boring Stuff with Python by Al Sweigart
- Think Python, 2nd Edition by Allen B. Downey
- Python for Everybody by Dr. Charles Severance

#### **Online Courses and Tutorials**

- String Formatting
- Codecademy Beginner Course
- Learn X in Y Minutes Python
- Python Cheat Sheets by Eric Matthes

## Thank you!