Workshop Details

Intermediate Workshop to Python Programming

Building the Foundation for Coding Success

Ricardo Chin

February 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

```python
a = [1, 2]
b = a
x = [1, 2]

a == b # True
a is b # True
a == x # True
a is x # False
```

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

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

BoolType

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

```python
a = True
if a:
    print('hello')
```

```python
x, y = 10, 20
is_greater = x > y   # False
if is_greater:
    print("x greater than y")
else:
    print("x not greater than y")
```

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

```python
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 Readability

- 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 = 0o52`
    - 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')`

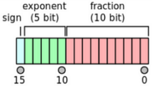
type	range	significant digits*		type	composed of
float16	$\pm(6.0*10^{-8}\ ..\ 65504)$	3	1bit 5bit 10bit	–	–
float32	$\pm(1.4*10^{-45}\ ..\ 3.4*10^{38})$	6	1bit 8bit 23bit	complex64	two float32's
float64	$\pm(4.9*10^{-324}\ ..\ 1.8*10^{308})$	15	1bit 11bit 52bit	complex128	two float64's
float128**	$\pm(3.7*10^{-4951}\ ..\ 1.1*10^{4932})$	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 π 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 $\approx$ 0.30000000000000004 due to limitations in 64-bit floats

```
0.1 + 0.2 == 0.3 # Returns False if float assigned

import math         # tolerance = 1e-09
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*

$$a = a + 1 \quad \texttt{or} \quad a += 1$$

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:

$$a = 2 + 2j$$

Other operations include: `-=, *=, ...`

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
$\ddot{A}$	11000100

Examples in UTF-8:

Character	Byte Representation
$\ddot{A}$	11000011 10100100
$\ddot\smile$	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"`

2. **Multi-line String Literals** — Multi-line representation

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

3. **Escape 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

Ex1 — Remove duplicates from a string (using set)

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
'weather 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`

```python
# 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

```python
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

```python
# Names
data = [("Ricardo",   12.51), ("Anand",       8.75),
        ("Simon",     15.32), ("Khaled",     10.27)]
```

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

```python
# Names
data = [("Ricardo",  12.51), ("Anand",      8.75),
        ("Simon",    15.32), ("Khaled",    10.27)]
```

```python
# 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}")
```

Bytes and Hexadecimal Notation

Bytes

- Sequences of integers (8 bits) in the range of 0 to 255
- Represent various data types, including images, text, and more
- Commonly used with storage media or network responses

Hexadecimal

- Bytes are often written in hexadecimal notation
- Values 0 to 15 represented by digits 0-9 and letters A-F

Decimal	Hexadecimal
1	0x1
9	0x9
10	0xa
15	0xf
16	0x10
17	0x11
31	0x1f
32	0x20

- Python uses the '0x' prefix to denote hexadecimal literals

Bytes and String Encodings

Creating Bytes from Lists

```
a = bytes([0, 64, 112, 160, 255])
b = bytes([0, 0x40, 0x70, 0xa0, 0xff])
print(bytes([0x00, 0x40, 0x70, 0xa0, 0xff])) # 'a'
```

- Illustrates creating bytes from a list of numbers
- Hexadecimal values can also be used directly

Creating Bytes from Byte Literal Strings

```
c = b"\x00\x40\x70\xa0\xff"
```

- 'b' prefix indicates a byte string
- Bytes usually hold encoded text, so we can do: `'ä'.encode('utf-8')` and `b'34'.decode('utf-8')`
- Also possible to represent it with ASCII characters

Lists

Lists

- Dynamic arrays for storing sequences of objects
- Versatile and mutable
- Ideal for homogenous entries of the same type and structure

```
primes = [2, 3, 5, 7, 11]
users = ["Ricardo", "Anand", "Blazhe"]
```

List Operations

- Indexing

```
primes[0] # returns 2
primes[-1] # returns last element of the list -> 11
```

- Accessing multiple elements (sublists)

```
primes[1:4] # returns [3, 5, 7]
```

Lists

- Modifying lists (append, insert, pop)

```
primes.append(13) # add 13 to the list primes
primes.insert(0, "Khaled") # Khaled to beginning
primes.pop() # pops last element of the list
primes.pop(0) # pops element at index 0
```

- Characteristics of the list

```
len(primes) # returns the size of the list
max(primes) # returns the max value of the list
min(primes) # returns the min value of the list
```

- Sorting lists

```
primes.sort() # increasing, alphabet for strings
primes.sort(reverse = True) # sorts decreasingly
primes.sort(key = len) # sorts by length
```

Lists

- Iterating through lists

```
for prime in primes:
    print(prime)
```

- Conditionals in lists

```
if "Ricardo" in users:
    print("Ricardo is here.")
```

Example:

```
users.sort(key = len)

def count_a(s):
    return s.count("a")

users.sort(key=count_a)
```

Tuples

Tuples

- Lightweight and immutable sequences of objects
- Entries separated by commas, typically surrounded by round brackets
- Commonly used for grouping related data

```python
single_value = ('Ricardo', ) # or
single_value = 'Ricardo', # notice the comma
values = ('Ricardo', 'Chin') # or
values = 'Ricardo', 'Chin'
```

- Elements in a tuple can be accessed using indexing
- `values[0]` returns 'Ricardo'

```python
first_name, last_name = two_values # var to tuple
first_name, last_name = last_name, first_name
```

Dictionaries

- Mappings of keys to values
- Example dictionary representing my information

```python
individual = {
    "first_name": "Ricardo",
    "last_name": "Chin",
    "nationality": "Portuguese",
    "birth_year": 1996
}
```

- Elements in a dictionary can be accessed using the keys

```python
individual["first_name"]  # return "Ricardo"
```

Dictionaries

Iterations Over Dictionaries

```python
for key in individual:
    print(key)
```

- Keys: "first_name", "last_name", "nationality", "birth_year"

Tip

Dictionary keys are maintained in insertion order

Iterations Over Key-Value Pairs

```python
for key, value in person.items():
    print(f'{key}, {value}')
```

- Shows how to iterate over key-value pairs in dictionaries
- Utilizes the `items()` method for iteration

Operations on Dictionaries

```python
d = {0: 'Ricardo', 1: 'Anand', 2: 'Blazhe'}
d[2] # value of key 2
d[2] = 'Thomas'
d[3]  # raises KeyError
d.get(3)  # returns None (similar to above, but safe)
d.setdefault(2, 'n')
d.setdefault(3, 'n')

d.keys() # get keys from dictionary
d.items() # get key-value pairs from dictionary
d1.update(d2) # overwrites if key is existing already
```

- Using `get()` and `setdefault()` for safe key retrieval
- Retrieving keys and items (key-value pairs)
- Updating dictionaries using `update(d)` or `extend(value)`

Dictionary Exercises

Exercises

- Exercise 1 — Get the keys of a dictionary as a list
- Exercise 2 — Get the values of a dictionary as a list
- Exercise 3 — Check if a key exists in a dictionary
- Exercise 4 — Count the occurrences of each item
- Exercise 5 — Check if all values are unique
- Exercise 6 — Find the key of the maximum value
- Exercise 7 — Sort a dictionary by keys or values
- Exercise 8 — Remove duplicates from dict values
- Exercise 9 — Get N largest or smallest items
- Exercise 10 — Count the frequency of characters in a string

Object Oriented Programming (OOP)

Introduction to OOP

- Python follows the OOP paradigm
- The mantra: ”Everything is an object.

Examples:

```python
# Example 1: Integer
a = 20

# Example 2: Method Call on Integer
bytes_representation = a.to_bytes(1, "big")

# Example 3: Method Call on String
uppercase_hello = "hello".upper()
```

The integer `a`, the method `to_bytes()`, and the string method `upper()`

Types and Instances

- Everything is an object and each object has a type

```python
message = "hello"
```

- This line creates a variable message and assigns it the value "hello". For terminology reasons, "hello" is an instance of the string type `str`

```python
type(message)
isinstance(message, str)
```

- `type()` returns the type of the object
- `isinstance()` checks if the object is an instance of a particular type
- the outcome is a Boolean returning `True` if message is an instance of a `str` type, otherwise `False`

Classes

- Classes are defined by the keyword `Class`
- The definition of class usually encompasses:
  1. **Attributes** — Properties of a class where data is stored
  2. **Methods** — Specific behaviours or functionalities of that class
  3. **Constructor** — `The __init__` method is a special method. It is called when an object is created and is used to initialize the object's attributes

```python
class BankAccount(object):
    def __init__(self, account_number, balance):
        self.account_number = account_number
        self.balance = balance

    def deposit(self, amount):
        # Method for depositing money

    def withdraw(self, amount):
        # Method for withdrawing money
```

Classes

```python
class BankAccount(object):
    def __init__(self, account_number, balance):
        self.account_number = account_number
        self.balance = balance

    def deposit(self, amount): # Deposit amount
        self.balance += amount
        print(f"New balance: {self.balance}€")

    def withdraw(self, amount): # Withdraw amount
        if amount <= self.balance:
            self.balance -= amount
            print(f"New balance: {self.balance}€")
        else:
            print("Insufficient funds.")

    def check_balance(self): # Current balance
        print(f"Current balance: {self.balance}€")
```

Classes

```python
# Create an instance of the BankAccount class
my_account = BankAccount("100_000", 1000)
# Use the methods of the BankAccount class
my_account.check_balance()  # Current balance: 1000€
my_account.deposit(500)     # New balance: 1500€
my_account.withdraw(200)    # New balance: 1300€
my_account.withdraw(1500)   # Insufficient funds.
my_account.check_balance()  # Current balance: 1300€
```

- The constructor `__init__` initializes an object with attributes

Warning!

- The double underscore in the constructor `__init__` indicates that this method is intended for internal use within the class and should not be accessed or modified directly from outside the class. This is called **mangling**

Classes

- We can instance private attributes and methods

```python
class BankAccount:
    def __init__(self, account_number, balance):
        self.__account_number = account_number
        self.__balance = balance # Private attrivyte

    def __validate_withdrawal(self, amount):
        """Validate if withdrawal is possible."""
        return amount <= self.__balance
```

- Attempting to access these directly from outside the class using name mangling is technically possible but not recommended

```python
print(my_account._BankAccount__account_number)
my_account._BankAccount__validate_withdrawal(200)
```

Exercise

- Class `Rectangle` with attributes length and width. Add methods `calculate_area()` and `calculate_perimeter()` to compute the area and perimeter of the rectangle
- Class `Circle` with attribute radius. Add methods `calculate_area()` and `calculate_circumference()` to compute the area and circumference of the circle
- Class `Student` with attributes name, age, and grades (a list of grades). Add methods `add_grade()` to add a grade to the list and `average_grade()` to calculate and return the average
- Class `Playlist` with attributes name and songs (a list of song objects). Add methods `add_song(song_title)`, `remove_song(song_title)`, and `display_songs()`

Inheritance

What is this fancy word: Inheritance?

It is a mechanism in OOP where a new class is created by inheriting attributes and behaviours from an existing class. The subclass automatically gains the methods and attributes of the superclass

1. **New class** — `superclass/base class`
2. **Existing class** — `subclass/derived class`

```python
class User(object):
    def __init__(self, username):
        self.username = username

class AdminUser(User): # inherited attributes from User
    def __init__(self, username, admin_level):
        super().__init__(username)
        self.admin_level = admin_level
```

Inheritance

Inheritance is commonly used in defining database models — example in a popular web framework `Django`

```python
from django.db import models

class Employee(models.Model):
    name = models.CharField(max_length=50)
    position = models.CharField(max_length=50)
    salary = models.DecimalField(max_digits=10,
     ↪   decimal_places=2)

class RegularEmployee(Employee):
    department = models.CharField(max_length=50)

class Manager(Employee):
    department_mgmt = models.CharField(max_length=50)
    team_size = models.PositiveIntegerField()
```

Inheritance

```python
# Creating instances of RegularEmployee and Manager
...

# Saving instances to the Django database
regular_employee.save()
manager.save()

# Querying the database
all_employees = Employee.objects.all()
regular_employees = RegularEmployee.objects.all()
managers = Manager.objects.all()

# Accessing fields
print(regular_employee.department)
print(manager.department_managed)
print(manager.team_size)
```

Composition

Composition?

Composition is an alternative to inheritance. It involves creating objects from other classes within a class, allowing for more flexibility and avoiding the pitfalls of deep class hierarchies

```python
class TicTacToeGame(object):
    def __init__(self):
        # Game logic implementation

class GameGUI(object):
    def __init__(self):
        self.game = TicTacToeGame()
        # GUI implementation using TicTacToeGame
```

- GameGUI uses composition by having an instance of TicTacToeGame within it. This way, it can access and use the functionality of TicTacToeGame

Composition

Composition?

Composition is an alternative to inheritance. It involves creating objects from other classes within a class, allowing for more flexibility and avoiding the pitfalls of deep class hierarchies

```python
class TicTacToeGame(object):
    def __init__(self):
        # Game logic implementation

class GameGUI(object):
    def __init__(self):
        self.game = TicTacToeGame()
        # GUI implementation using TicTacToeGame
```

- GameGUI uses composition by having an instance of TicTacToeGame within it. This way, it can access and use the functionality of TicTacToeGame

Docstrings

These OOP concepts can drastically decrease readibility — **This is why docstrings are useful:**

1. They provide documentation for functions, classes, and modules describing purpose, usage, and behaviour of the code
2. Placed at the beginning of the function, class, or module

```python
def fib(n):
"""Compute the n-th Fibonacci number.

Parameters:
- n (int): A nonnegative integer.

Returns:
int: The n-th Fibonacci number."""
# Function implementation...
```

Further info: `help()` function or accessing the `__doc__` attribute

Control Structures

Control structures are essential components in programming languages that enable the control flow of a program

Control Structures

- if ... else ...
- loops
  - while
  - for ... in ...
  - for ... in range()
- try ... except ... finally

The structure is usually pretty straightforward

```python
for i in range(5):
print(i)
```

If Statements

Previously we saw conditions for `if` and `while` where we usually used expressions that evaluate to boolean values `(if a == b)`

Any value can be used in a condition as most values are considered ”truthy”

```python
name = input("Enter your name: ")
if name:
    print(f"Hello, {name}!")
else:
    print("Empty.")
```

```python
print(f"Hello, {name}!") if name else print("Empty.")
```

```python
13 <= age and age <= 19
```

```python
13 <= age <= 19
```

For Loops

Recap: Tuple unpacking allows to assign multiple variables at once when iterating over a sequence

- `enumerate()` is a built-in function that returns an iterator of tuples containing indices and values from a list

```
names = ['Ricardo', 'Anand', 'Khaled']

for i, name in enumerate(names):
print(f'{i}: {name}')
```

Enumerate returns a data structure that behaves like this list:

```
0: Ricardo
1: Anand
2: Khaled
```

- `enumerate()` generates pairs of index and value, allowing to iterate over both simultaneously

Nested For Loops

- `os.walk()` is used to iterate over all files and subdirectories in a specified directory. We can also nest `for` loops for going through all the files in the directories

```python
import os

for directory, dirs, files in os.walk("C:\\"): # all
    for file in files: # Each file in the directory
        if not file.endswith(".xlsx"): # Not excel?
            continue # We skip non-excel files

        # Process text files
        file_path = os.path.join(directory, file)
        print(f"Processing: {file_path}")
```

- `continue` statement skips the processing for non-xlsx files

List Comprehensions

List comprehensions provide a concise way to create lists based on existing lists, allowing for transformations and filtering

```python
names = ["Ricardo", "Anand", "Khaled"]

uppercase_names = [name.upper() for name in names]
```

```python
# Results
["RICARDO", "ANAND", "KHALED"]
```

Another example:

```python
amounts = [10, -7, 8, 19, -2]

positive_amounts = [amount for amount in amounts if
    amount > 0]
```

This creates a new list `positive_amounts` containing only the positive values from the original list `amounts`

Dictionary Comprehensions

Dictionary comprehensions are expressions followed by a `for` clause inside curly braces , and optionally, one or more `if` clauses

```
{key_exp: value_exp for item in iterable if
↪   condition}
```

Where an iterable (list, tuple, etc.) contains various items

```
numbers = [1, 2, 3, 4, 5] # Square of numbers
squares = {num: num**2 for num in numbers}
```

```
grades = {'Alice': 90, 'Bob': 85, 'Charlie': 92}
adjusted_grades = {name: grade + 5 for name, grade in
↪   grades.items()}
```

Exercises

- Exercise 1 — Generate a list of numbers from 0 to 10
- Exercise 2 — Generate a list of even numbers from 40 to 100
- Exercise 3 — Generate a list of squared numbers from 1 to 15
- Exercise 4 — Extract the first letter of each word in a sentence
- Exercise 5 — Generate a list of vowels from a string
- Exercise 6 — Generate a list of palindromic nums up to 1000
- Exercise 7 — Generate a list of nums up to 10, squared if even, cubed if odd
- Exercise 8 — Swap keys and values in a dictionary
- Exercise 9 — Create a dict from a string with char frequencies
- Exercise 10 — Generate a dictionary with word length as keys and words as values from a string sentence

Exception Handling

Exception handling allows us to manage errors gracefully

```python
def divide(a, b):
    try:
        if b == 0:
            raise ValueError("Cannot divide by zero")
        result = a / b
        print(f"The result is: {result}")
    except ValueError as ve:
        print(f"Error: {ve}")
    finally:
        print("This will be executed no matter what")

divide(10, 0) # Try other values
```

1. The `try` block attempts to execute the division operation
2. If a `ValueError` is raised (due to division by zero), the `except` block catches it and prints an error message

Module Name and Entrypoint

....

Virtual Environments

....

Arbitrary Number of Arguments (Args / Kwargs)

....

Global and Local Scope

....

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!