Introduction to Python Programming

Building the Foundation for Coding Success

Ricardo Chin

January 10, 2024

Presentation Overview

Use Cases

Simple Syntax - Beginner Friendly

- Basic syntax examples (indentation for blocks, simplicity of loops, etc.)
- Comparisons with other languages to showcase Python's readability

Versatility in Different Domains

- Specific use cases in scientific computing (NumPy, SciPy), web development (Django, Flask), GUI programming (Tkinter, PyQt)
- Real-world examples of companies or projects using Python in these domains

Use Cases

Discussion on the Standard Library and Packages

- Popular third-party packages (e.g., Pandas, Matplotlib, Requests) and their significance in expanding Python's capabilities
- Comparisons with other languages to showcase Python's readability

Community and Support:

- Python's has an active community, diverse user base, and strong support through forums, tutorials, and online resources.

Python Development and Versions

- Python 3: Current version (e.g., 3.12) released annually in October
- Python 2: Support ended in 2019; Despite the end of support, around 10% of developers continued to use Python 2 for various reasons, including legacy codebases and migration challenges.

Four Code Examples

```python
# Python snippet 1
for i in range(5):
    print(i)
```

```python
# Python snippet 2
x = 10
if x > 5:
    print("x is greater")
```

```python
# Python snippet 3
def greet(name):
    print("Hello, " + name)
```

```python
# Python snippet 4
numbers = [1, 2, 3, 4, 5]
squared = [x ** 2 for x in numbers]
print(squared)
```

Installation on Windows

Download from `https://python.org`

During installation:

- Check the option ”Add Python 3.x to PATH”

Verify the installation:

- `python --version` should display the version number
- `pip install requests` should successfully download and install ’requests’

Installation includes:

- Python runtime for executing Python code
- Interactive Python console
- IDLE: simple development environment
- PIP: package manager for installing extensions

Interactive Python Console

Options for Running Python Code:

- Writing programs as files and executing them (e.g., GUI, web apps, data processing)
- Typing code into an interactive console or notebook for quick calculations, experimentation, data exploration, or analysis

Options Available:

- Local installation and usage
- Online notebooks like Jupyter

Launching Python Console:

- Via command prompt: ‘python‘
- Using the Start Menu (e.g., Python 3.12)

Quitting:

- Exiting with ‘exit()‘

Integrated Development Environments (IDEs)

Available Open-source IDEs

- VS Code
- PyCharm
- Spyder

VS Code Setup

- Installation:
  1. Open the extensions view from the left sidebar (fifth icon)
  2. Search and install the extension named ”Python” by Microsoft

- Registering Python Installation with VS Code:
  1. Open the command palette (F1 or Ctrl + Shift + P)
  2. Search for `"Python: Select Interpreter."`
  3. Choose the desired Python version (usually only one available)

Variables

- Naming conventions and rules
  - Written in lowercase with words separated by underscores
  - Consist of letters, digits, and underscores only

- Variable Examples

```
1 birth_year = 2000
2 current_year = 2024
3 age = current_year - birth_year
```

- Overwriting Variables

```
1 birth_year = 1996
2 birth_year = 1997
3 age = 27
4 age = 27 + 1
```

- **Printing using print(`variable_name`)**

Basic Data Types

Primitive Types:

1. **int (Integer):** Represents whole numbers, positive or negative, without any decimal points
2. **float (Floating-point Number):** Represents decimal numbers, including fractions and exponential values
3. **str (String):** Represents a sequence of characters enclosed in single or double quotes
4. **bool (Boolean):** Represents the truth values `True` or `False`
5. **None:** Denotes the absence of a value or lack of data. It's a unique type in Python

Basic Data Types

Integer (int)

- Represents whole numbers, e.g., -5, 0, 100
- Supports arithmetic operations: addition, subtraction, multiplication, division, and exponentiation
- Operations involving integers produce integer results unless divided (/), which produces a float

Floating-point Number (float)

- Represents decimal numbers, e.g., 3.14, -0.002, 7.0
- Supports all arithmetic operations similar to integers
- Division (/) always produces a float
- Precision in floating-point arithmetic might lead to slight rounding errors in calculations

Integer and Float Data Types

Integer Operations

How would you do integer operations with division or exponentiation? What data type do you obtain?

Integer and Float Data Types

Integer Operations

How would you do integer operations with division or exponentiation? What data type do you obtain?

- Integer division using ‘/‘ returns a float, and exponentiation uses the ‘**‘ operator

Integer and Float Data Types

Integer Operations

How would you do integer operations with division or exponentiation? What data type do you obtain?

- Integer division using ‘/‘ returns a float, and exponentiation uses the ‘**‘ operator

What if we’d like to round up or truncate our float value?

Imagine you have worked 27 overhours and you would like to know how much that translates to available days. How do you do it?

Integer and Float Data Types

Integer Operations

How would you do integer operations with division or exponentiation? What data type do you obtain?

- Integer division using '/' returns a float, and exponentiation uses the '**' operator

What if we'd like to round up or truncate our float value?

Imagine you have worked 27 overhours and you would like to know how much that translates to available days. How do you do it?

```
1 days = total_hours // 7.7  # Available days
2 # For total_hours = 27, days = 27 // 7.7 = 3 days
```

```
1 remaining_hours = total_days % 7  # Remaining hours
2 # For total_days = 27, hours = 27 % 7.7 = 3.89 hours
```

Integer Limits

Are there any practical limitations to the size of integers in Python? How does Python handle large integers?

Integer and Float

Integer Limits

Are there any practical limitations to the size of integers in Python? How does Python handle large integers?

- In Python, integers have arbitrary precision, meaning they can grow as large as the available memory allows without practical limitations

Scientific Notation

How does Python handle very large or very small floating-point numbers?

Integer and Float

Integer Limits

Are there any practical limitations to the size of integers in Python? How does Python handle large integers?

- In Python, integers have arbitrary precision, meaning they can grow as large as the available memory allows without practical limitations

Scientific Notation

How does Python handle very large or very small floating-point numbers?

- Python uses scientific notation for very large or very small floating-point numbers $\rightarrow$ 1.5e6 represents $1.5 \cdot 10^6$ (1 500 000)

String and Boolean

String (str)

- Represents sequences of characters enclosed in single (' ') or double (" ") quotes
- Supports various string manipulation operations: concatenation, slicing, formatting, etc $\rightarrow$ in built functions will be seen later
- Immutable - once created, a string cannot be modified in place

Boolean (bool)

- Represents truth values: `True` or `False`.
- Essential for conditional and logical operations: if statements, while loops, etc
- Results of logical expressions (e.g., comparisons) are of type bool

String and Boolean

Questions

1. What is the difference between single, double and triple quotes?
2. String concatenation?
3. How can I do string indexing and slicing?

String and Boolean

Questions

1. What is the difference between single, double and triple quotes?
2. String concatenation?
3. How can I do string indexing and slicing?

Answers

1. No significant difference between single and double quotes, except when the string itself contains one of these characters. Triple quotes are used for multiline strings
2. String concatenation in Python involves using the + operator to combine two strings together (str1 + " " + str2)
3. String indexing allows accessing individual characters within a string using indices. Slicing retrieves substrings by specifying a range of indices. **Indexing in Python starts at index i = 0!**

String and Boolean

String Concatenation

```
s1, s2, name = "Hello", "World", Ricardo
result = s1 + " " + s2 + ". This is " + name
```

F-strings

```
ans = f"{s1} {s2}. This is {name}!"
```

String Indexing and Slicing

→ Notice the index

```
text = "Python"
char = text[2] # Accessing the character 't' at idx 2
substring = text[1:4]  # Slicing to get 'yth'
```

Tip

String Operations: slice(), strip(), reverse(), lower() and upper()

String and Boolean

How do we include characters like ” in a string?

```
1 string_t = "When you drink, you must say: "Prost!""
```

Python treats the sequence \” like a single quote

```
1 string_t = "When you drink, you must say: \"Prost!\""
```

You can also introduce line breaks using *n*

```
1 res = "string_t\nstring_t"
2 print(res)
```

None

None Data Type

- Denotes the absence of a value or lack of data
- Often used to reset variables
- Evaluates to False in boolean contexts

```
1 first_name = "Ricardo"
2 middle_name = None
3 last_name = "Chin"
```

Tip

Best practices include using None as a default value for optional function arguments or variables to signify missing values

Variable Conversions

Operations

- **Type Coercion and Mixed Operations** → When an operation involves both integers and floats, Python automatically promotes integers to floats to maintain precision
- **Casting between Integers and Floats** → Explicit casting can be done using int() and float() functions
  1. Integer to Float: float(5) converts the integer 5 to the float 5.0.
  2. Float to Integer: int(3.7) converts the float 3.7 to the integer 3

```
1 freedom = True and some_function()
2
3 pi = 3.1415
4 pi_int = int(pi)
5 message = "Pi is approximately " + str(pi_int)
```

Functions

Functions are like specialized tools that perform specific tasks

Example of Predefined Functions

- `len()` - Calculates the length of a string or list
- `id()` - Retrieves a unique ID for an object
- `type()` - Identifies the type of an object
- `print()` - Displays information on the screen

A function can be passed the so-called input parameters and produce a result (a return value)

Example of Input / Output Usage

- `len()` - Takes a string or list as input, returns the count
- `print()` - Receives various types of inputs but doesn't specifically return a value

Methods

Methods are functions that belong to specific object types, like strings (`str`)

Example of String Methods

- `first_name.upper()` - Changes a string to uppercase
- `first_name.count("a")` - Counts occurrences of a specific character
- `first_name.replace("a", "@")` - Replaces characters within a string

Composite Data Types and Structures

- Composite data structures in programming encompass various types that allow the combination of multiple elements or data types into a single unit
- Some composite data structures include **lists, sets, tuples and dictionaries**
  1. **Lists** represent collections of elements, allowing duplicates and modification of their content → `[1, 2, 1, 'apple', 'banana']`
  2. **Sets** contain an unordered collection of unique elements, discarding → duplicates **lists, sets, tuples and dictionaries**
  3. **Tuples** resemble lists but are immutable, meaning their elements cannot be altered once defined
  4. **Dictionaries** store key-value pairs, offering a mapping relationship between unique keys and corresponding values

Object Mutation

- Certain objects can undergo direct mutation, such as through methods like `.append()` or `.pop()`
- Mutable objects like lists and dicts fall under this category
- On the flip side, several basic objects, once created, remain immutable. Yet, they can be substituted by other objects as needed. It includes integers, floats, strings, booleans and tuples

Summary

In the following slides, I will describe briefly each of the composite data structures. For more info:

1. `help(list)`
2. `https://docs.python.org/3/library/index.html` (Google: "python library")

Dictionaries

Dictionaries in Python, often referred to as hash maps in other programming languages, are mutable collections that store key-value pairs. They offer a mapping structure where each key is associated with a corresponding value

```
person = {   "first_name": "Ricardo",
             "last_name": "Chin",
             "nationality": "Portuguese",
             "age": 27                    }
```

Dictionary Characteristics

- Key-Value Mapping: Dictionaries map unique keys to specific values. The keys within a dictionary must be unique, but the values can be duplicates
- Mutable and Unordered: Dictionaries are mutable, meaning their contents can be changed after creation. Any data type can be also stored

Dictionaries Operations

- **Creation**
  Dictionaries are created using curly braces and follow the syntax key1: value1, key2: value2, ...
- **Accessing Values**
  Values within a dictionary are accessed by their corresponding keys using square brackets ([]) or the get() method

```
print(person["first_name"])   # Output: Ricardo
print(person.get("age"))   # Output: 27
```

- **Accessing Values**
  Dictionaries are mutable, allowing the addition, modification, or deletion of key-value pairs
- **Iterating Through a Dictionary**
  Python provides ways to iterate through `keys`, `values`, or both simultaneously using loops or methods like `keys()`, `values()` and `items()`

Lists

Lists

Lists or arrays are versatile and mutable collections that store ordered sequences of elements. They are denoted by square brackets [ ] and enable flexible operations for modification, insertion, deletion, and retrieval of elements

```
1 primes = [2, 3, 5, 7, 11]
2 users = ["Ricardo", "Pedro", "Henrique"]
3 products = [          {"name": "iPhone 15 Max Pro",
   ↪  "price": 1699},
4                       {"name": "Fairphone", "price":
                         ↪  599},
5                       {"name": "Pixel 8", "price": 999}
                         ↪  ]
```

List Characteristics

- Ordered Collection: Lists maintain the order of elements as they are added and allow indexing to access specific elements by their position within the list
- Mutable: Elements can be modified, appended, removed, or replaced after creation. Operations like append(), insert() and pop() are common. Any data type can be stored

Lists Operations

- **Accessing Elements**
  1. Elements within a list are accessed using zero-based indexing
  2. primes[0] accesses the first element, primes[-1] accesses the last element, and slicing primes[1:3] retrieves elements from index 1 up to, but not including, index 3
- **Modifying Elements**

  Elements within lists can be modified directly by assignment

```
1 primes[0] = 13
2 primes[-1] = 'Ricardo'
3 primes[1] = primes[0]
```

  Can you guess what happens?
- **Accessing Values**

  Built-in list methods like append(), extend(), insert(), remove(), and pop() to manipulate lists are common. These methods enable various operations like adding elements, removing specific elements, or altering the list's structure

Sets

Sets are unique, unordered collections of elements that **do not allow duplicates**. They're denoted by curly braces or using the set() constructor. Primarily used for membership testing, eliminating duplicates from sequences, and performing set operations like union, intersection, and difference

```
# Creating a set directly
my_set = {1, 2, 3, 4, 5}

# Creating a set using the set() constructor
another_set = set([3, 4, 5, 6, 7])
```

Sets Characteristics

- **Uniqueness**: Sets contain unique elements, ensuring that each element appears only once within the set
- **Hash-based Storage**: Internally, sets utilize hash tables to store elements, allowing for quick membership checks and set operations

Sets Operations

- **Adding Elements**

  Elements can be added to a set using the add() method or by using the union (|) operator

```
my_set = {1, 2, 3}
my_set.add(4) # Adds the element 4 to my_set
```

- **Removing Elements**

  Elements can be removed using the remove() or discard() methods

```
my_set.remove(2) # Removes element 2 from my_set
```

- **Sets Tasks**

  Operations like union (|), intersection (&), difference (-), and symmetric difference (^)

```
union_set = my_set | another_set  # Union
inter_set = my_set & another_set  # Intersection
difference_set = my_set - another_set
# Elements in my_set but not in another_set
symmetric_difference_set = my_set ^ another_set
# Elements in either but not both
```

Tuples

Tuples

Tuples are ordered collections of elements, similar to lists, but they are immutable, meaning their contents cannot be modified after creation. Tuples are denoted by parentheses () and can contain various data types, including integers, strings, floats, other tuples, etc

```
# Creating a tuple
my_tuple = (1, 'apple', 3.5, (4, 5, 6))
```

Tuples Characteristics

- **Ordered Sequence**: Maintain order of elements, allowing access via indexing
- **Immutable Nature**: Elements cannot be changed, added, or removed
- **Flexible Data Types**: Can hold elements of different data types within the same structure, similar to lists

Tuples Operations

- **Accessing Elements**: Elements within a tuple are accessed using zero-based indexing or slicing. For example, my_tuple[0] accesses the first element, my_tuple[-1] accesses the last element, and slicing my_tuple[1:3] retrieves elements from index 1 up to, but not including, index 3

```
# Accessing elements in a tuple
print(my_tuple[1])  # Output: 'apple'
```

- **Tuple Methods**: Tuples have limited methods due to their immutability. However, they have methods like `count()` and `index()` for counting occurrences and finding the index of elements within the tuple

```
# Counts occurrences of 'apple'
print(my_tuple.count('apple'))

# Returns the index of 3.5
print(my_tuple.index(3.5))
```

First Python Program

We'll create a file called `greeting.py`
Our program will ask the user their name and greet them

First Python Program

We'll create a file called `greeting.py`
Our program will ask the user their name and greet them

```
1 print("What is your name?")
2 name = input() # input will always return a string
```

Okay..now we wrote our name, how do we display the result?

First Python Program

We'll create a file called `greeting.py`
Our program will ask the user their name and greet them

```
1 print("What is your name?")
2 name = input() # input will always return a string
```

Okay..now we wrote our name, how do we display the result?

```
1 print("Nice to meet you, " + name)
```

For increasing readability, we might want to add comments to our code. Comments start with and describe code functionality. Typically placed above the code they explain.

```
1 # Determine the length of the name
2 name_length = len(name)
```

To execute the program type the command line `python greeting.py` on the terminal or the green play button in VS Code

Exercises

- Write a program which asks the user for their name. It should respond with the number of letters in the user's name

  For this purpose use the function `len(...)` to determine the length of a string
- Write a program called `age.py` which will ask the user for their birth year and will respond with the user's age in the year 2022
- Write a program called `sum.py` which will ask the user for N numbers and will do the same of numbers in the range from 1 to N
- Write a program called `length_of_concatenation.py` which will ask the user for 2 strings, concatenate them and afterwards will return the length of the concatenated strings

Libraries and More Built-in Functions

Standard Python Libraries

Additional modules that can be imported

```
import random
# Return random number between 1 - 10
random_number = random.randint(1, 10)
print("Random number:", random_number)
```

+ Builtin Functions

```
print(), input()
len(), max()
min(), open()
range(), round()
sorted(), sum()
type(), read()
```

Other Useful Libraries / Use Case Examples

```
random
math
datetime
os (op. system)
collections
shutil
```

```
import random
print(random.randint(1, 6))
print(random.choice(["heads", "tails"]))

file = open("message.txt", "w")
file.write("hello\n")
file.write("world\n")
file.close()
```

Control Structures

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

Control Structures

- if / else statements
- loops
  - while loop
  - do while loop
  - for loop (counting loop)
  - foreach loop

The structure is usually pretty straightforward

- if ... else ...
- loops
  - while
  - for ... in ...
  - for ... in range(...)

Comparisons

To utilize `if` and `while` statements, comparisons between values are necessary

```
a = 1
b = 0

print(a == b)   # Is a equal to b?
print(a != b)   # Is a not equal to b?
print(a < b)    # Is a less than b?
print(a > b)    # Is a greater than b?
print(a <= b)   # Is a less than or equal to b?
print(a >= b)   # Is a greater than or equal to b?
```

Comparisons yield boolean values `True / False`. We can capture this outcome in a variable

```
# Checking if 'age' is 18 or older
is_adult = age >= 18
```

Comparisons

It is possible to combine comparisons using `and`, `or` and `not`

```
if age >= 18 and country == "Austria":
    print("May drink alcohol.")

if temperature < -15 or temperature > 40:
    print("Extreme weather!")

if shopping_basket not None:
    print("There are still items left.")

name = "Alice"
other_name = "Bob

print(name == other_name) # Are names identical?
print("Alli" in name)     # Does it contain "Alli"?
print("Ali" not in name)  # Does it contain "Ali"?
```

If Statements

`if statements` help computers decide what to do based on certain conditions. It's like giving the computer a set of instructions

```
1 number = 27
2
3 if number % 2 == 0:
4     print("The number is even.")
5 else:
6     print("The number is odd.")
```

How does it work?

- Checks if number is even by using the modulus operator `%`
- If the remainder of `number / 2` is `0`, it prints "The number is even."
- If not, it prints "The number is odd."

If, elif and else Statements

For multiple conditions to check: `if, elif and else` are used. The computer goes through each condition, executing the first one that's true

```
1 if number < 0:
2     print("The number is negative.")
3 elif number == 0:
4     print("The number is zero.")
5 elif number % 2 == 0:
6     print("The number is divisible by 2.")
7 else:
8     print("The number isn't in any category.")
```

Code Blocks

- Code block is a group of statements that are executed together when a condition (like an `if statement`) is true
- The colon `:` signifies the start of a code block, and the indented lines following the colon are considered part of that block

If, elif and else Statements

Exercise: Coin Flip

Simulate coin flipping via `random.choice(["heads", "tails"])`
Let the user guess the outcome and tell them if they were right

If, elif and else Statements

Exercise: Coin Flip

Simulate coin flipping via `random.choice(["heads", "tails"])`
Let the user guess the outcome and tell them if they were right

```
1  import random
2
3  coin_outcome = random.choice(["heads", "tails"])
4  user_guess = input("Guess (heads or tails): ")
5
6  if user_guess.lower() == coin_outcome:
7      print("Congratulations! Your guess was correct.")
8  else:
9      print(f"Sorry, landed on {coin_outcome}.")
10     print("Your guess was incorrect.")
```

For Loops

`For` loop is like a helpful assistant that goes through a list of items, one by one, and performs a set of instructions for each item.

```
1 names = ["Alice", "Bob", "Ricardo"]
2
3 for name in names:
4     print("Hello, " + name + "!")
5
```

How does it work?

- The `for` loop starts by saying, "Hey Python, let's go through each name in the `names` list."
- For each name, the loop performs the action inside its block of code. In this case, it says hello to each person by printing "Hello, [name]!"

For Loops

Exercises

Assume the list = [1, 2, 3, 4, 5]

- Exercise 1: Prints numbers of the list in separate lines
- Exercise 2: Calculate the cumulative sum of the list
- Exercise 3: Print only the odd numbers
- Exercise 4: Print a pattern using loops

```
1 *
2 **
3 ***
4 ****
5 *****
```

- Exercise 5: Create a list with the square valued of each number

Counting Loops

Counting loops are used to iterate through a sequence of numbers or perform an action a certain number of times. In Python, `for` loops and the `range()` function are commonly used for counting loops

```
1 # Counting from 0 to 9 using a for loop
2 for i in range(10): # 10 excluded
3     print(i)
```

How does it work?

- `range(10)` generates a sequence of numbers from 0 to 9 ([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
- The `for` loop iterates through each number in this sequence, storing the value in `i`, and executes the code block (in this case, printing `i`)

For counting loops, `i` is commonly used as a variable name, especially when the purpose is straightforward iteration

Counting Loops

Exercises

- Exercise 1: Prints numbers from 0 to 9 in separate lines
- Exercise 2: Calculate the cumulative sum of numbers from 0 to 9
- Exercise 3: Create multiplication table of 5
- Exercise 4: Print a pattern using loops

```
1
12
123
1234
12345
```

- Exercise 5: Print the duplicates of a given list [3, 4, 3, 5, 6, 7, 8, 5]
- Exercise 6: Print the first 5 terms of the Fibonacci Series

While Loops

`while` loops repeatedly execute a block of code as long as a specified condition is true. They are useful when the number of iterations is not fixed beforehand and depends on certain conditions

```
1 while condition:
2 # code block to execute as long as it stays true
3 # the condition is re-evaluated after each iteration
```

Key Points

1. `Condition`: Repeats continuously while statement is `True`
2. `Iteration`: The code inside the loop runs repeatedly until the condition becomes False. If the condition is initially False, the code inside the loop will not execute
3. `Updating Condition`: It's crucial to have a way to change the condition inside the loop to avoid an infinite loop. Usually, a variable involved in the condition is modified within the loop

While Loops

Let's print numbers from 1 to 5 using a `while` loop:

```
num = 1
while num <= 5:
    print(num)

    # Incrementing variable `num`
    num += 1
    # Break out of the loop when num > 5 (condition)
```

Advantages

- `Dynamic Control`: Useful when the number of iterations isn't predetermined and depends on conditions
- `Flexibility`: Allows for a wide range of applications such as user input, iteration based on changing conditions, and more

If the condition never becomes `False` or if there's no logic to change the condition, it causes an infinite loop

While Loops

Exercises

- Exercise 1: Do a countdown from 10 to 0
- Exercise 2: Create a simple guessing game
- Exercise 3: Find the highest number in a list
- Exercise 4: Count and print the number of occurrences of a specific letter ('a') in a string `"apple and banana are fruits"`
- Exercise 5: Create an infinite loop of "Hello world!" prints
- Exercise 6: Create a team list program

```
1 Register a person or "X" to quit:
2 Ricardo
3
4 Your team consists of:
5 ["Anand", "Ricardo", "Simon"]
```

Break Statement

The `break` statement is a control flow statement used to terminate the execution of a loop prematurely

```
while True:
if condition:
    break  # Terminate loop if condition is met
```

When encountered within a loop (such as `for` or `while`), the `break` statement causes the loop to immediately stop iterating

```
while True: # cause infinite loop
    guess = int(input("Enter a number (0 to exit):"))
    if guess == 0:
        print("Exiting the loop.")
        break
    else:
        print(f"You entered: {guess}")
```

`break` is particularly useful when the loop needs to end before its natural completion based on a specific condition

Pass Statement

`Pass` statement is used as a placeholder for an empty code block where Python syntax requires a statement but no action is needed

Empty Codeblocks

```
1 # TODO: warn the user if path doesn't exist
2 if not os.path.exists(my_path):
3     pass
```

`if condition:` `pass` indicates to ”do nothing” when a certain condition is met

Statements Across Lines

Statements can span multiple lines by enclosing them in parentheses `()` without the need for line continuation characters.

Long Assignments

`a = (2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10)` can be written across multiple lines for readability

```
1                    a = (2 + 3 + 4 + 5 + 6 +
2                         7 + 8 + 9 + 10)
```

`()` or alternatively using newline `\`

```
1                    a = 2 + 3 + 4 + 5 + 6 + \
2                        7 + 8 + 9 + 10
```

Positional and Keyword Parameters

1. **Positional Parameters**: Passed based on their position in the function call

   Example: `open("this_file.txt", "w", -1, "utf-8")`, where `"this_file.txt"` is the file name, `"w"` is the mode, `"-1"` might be the buffer size, and `"utf-8"` is the encoding

2. **Keyword Parameters**: Passed with a keyword and value, allowing flexibility and clarity

   Example: `open("this_file.txt", encoding="utf-8", mode="w")`, explicitly defining the encoding and mode

3. **Optional Parameters**: Parameters that don't have to be provided in a function call

   Example: In the previous example, parameters `encoding` and `mode`

Defining Functions

Syntax — def function_name(parameters)

```
1    def average(a, b):
2    m = (a + b) / 2
3    return m
```

Usage — Functions encapsulate reusable blocks of code, performing specific tasks

Exercises

- **Calculating Area**: Create a function to calculate the area of a rectangle or circle
- **Converter**: Develop a function that converts temperature between Fahrenheit and Celsius
- **String Manipulation**: Write a function that reverses a given string
- **List Operations**: Create a function to find the maximum or minimum number in a list

Variable Scope

Function Scope — Variables defined within a function have local scope, separate from variables defined outside the function

```
1 m = "Hello, world"
2
3 def average(a, b):
4     m = (a + b) / 2
5     return m
6
7 x = average(1, 2)
8 print(m) # Prints "Hello, world" as the function
  ↪  creates a local 'm'
```

Read Access — Functions can access variables from the outer scope, but modifications create local variables

Immutable vs. Mutable — Immutable types (like strings) cannot be modified in place; assigning creates a new local variable

Function Exercises

Function Exercises

1. Leap Year Check: Verify if a given year is a leap year based on certain criteria
2. Prime Number Check: Determine whether a number is a prime number or not
3. Finding Prime Numbers: Create a function that returns all prime numbers within a specified interval
4. Fibonacci Number Calculation: Compute Fibonacci numbers within a given range
5. Lottery Numbers Generation: Generate a list of random lottery numbers
6. Yes/No Question Function: Create a function that asks a yes/no question and returns True or False

Modules, Packages and Libraries

Module — A module is a file containing Python code that defines variables, functions, and classes. It can be imported and used in other Python files. **Example:** `math`, `os`, `random`

Package — A package is a directory that contains multiple Python modules, typically accompanied by an __init__.py file to indicate it's a package. **Example:** `numpy` , `pandas` , `matplotlib`

Examples Imports

- **Module Import** — `import urllib.request`
- **Function Import** — `from urllib.request import urlopen`
- **Object Import** — `import sys`

Importing Local Modules

Local modules are Python files created and used within the same project

```python
def greet(name):
return f"Hello, {name}!"
```

```python
import my_module
print(my_module.greet("Alice"))
```

To create a local module, save a Python file `(.py)` and import its contents in another Python file using import

PyPI (Python Package Index) — PyPI is the official repository for Python packages, hosting thousands of installable Python packages/modules

PIP (Python Package Manager) — PIP is a package manager for Python, used to install, manage, and remove Python packages from PyPI

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

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

The End

Thank you!