Learn Python in Under 3 Hours

Variables, For Loops, Web Scraping & Full Project — Course Notes

SECTION 1: Setting Up Your Python Environment

1.1 Why Learn Python?

Python is one of the most in-demand programming skills in the world. Once intimidating to beginners coming from tools like Excel and SQL, Python becomes highly accessible with practice and is used across data analysis, automation, web scraping, machine learning, and more.

- Used heavily in data analytics, data science, and automation roles
- Complements Excel and SQL skills — making you a much more well-rounded analyst
- Instructor has been using Python for 7+ years and considers it one of the most valuable tools they know

1.2 Installing Anaconda

Anaconda is a free, open-source distribution of Python (and R) that comes pre-packaged with Jupiter Notebooks and many essential libraries. It is the recommended way to get started.

- Download from anaconda.com — available for Windows, Mac, and Linux
- On Windows, check if your system is 32-bit or 64-bit (Settings > About) before downloading
- Installation takes approximately 3.5 GB of disk space
- During install: Do NOT check 'Add Anaconda to PATH' — this can interfere with other Python environments
- Do check 'Register Anaconda as my default Python' to keep settings consistent

1.3 Navigating the Anaconda Navigator

- After install, search for 'Anaconda Navigator' and open it
- The Navigator is a graphical launcher for different Python tools: Jupiter Notebook, VS Code, Spyder, RStudio, and more
- For this course, click Launch next to Jupiter Notebook
- Jupiter opens in your web browser and shows your file system where you can create, open, upload, and organize notebooks

1.4 Using Jupiter Notebook

Jupiter Notebook is where 99% of the course work happens. Key concepts:

- Cells — the individual blocks where you write and run code. Run a cell with Shift+Enter
- Markdown cells — for notes and headings (not code). Use # for a large heading
- Code cells — for actual Python code; output appears directly below
- Kernel — the Python engine running behind the scenes. If it freezes, you can interrupt, restart, or restart and re-run all cells from the Kernel menu
- Save: File > Save, or Ctrl+S. Rename: click the title at the top

Toolbar buttons (from left to right):

- Save — saves the current notebook
- Insert cell below — adds a new cell
- Cut / Copy / Paste selected cells
- Move cell up / Move cell down
- Run cell — runs the currently selected cell
- Interrupt kernel — stops a running cell
- Restart kernel — resets the Python session

SECTION 2: Variables

2.1 What is a Variable?

A variable is a container for storing a data value. You assign a value to a variable using the = sign. Python automatically detects the data type — you do not need to declare it manually.

```
x = 22          # x is now an integer
y = 'mint choc' # y is now a string
print(x)        # outputs: 22
type(x)         # outputs: <class 'int'>
```

2.2 Assigning Variables

- Single assignment: x = 22
- Multiple values to multiple variables: x, y, z = 'chocolate', 'vanilla', 'rocky road'
- Multiple variables to one value: x = y = z = 'root beer float'
- Variables are case-sensitive: x and X are different variables
- You can overwrite a variable by reassigning it: y = 'new value'

2.3 Variable Naming Conventions

Three common styles:

- Camel Case: testVariableCase — first word lowercase, subsequent words capitalized
- Pascal Case: TestVariableCase — every word capitalized
- Snake Case: test_variable_case — all lowercase, words separated by underscores

Snake case is recommended in Python for readability. Best practices:

- Good: test_var, _test_var, testVar2
- Bad: 2test_var (cannot start with a number)
- Bad: test-var (dashes not allowed)
- Bad: test var (spaces not allowed), no special symbols like . / = allowed

2.4 String Concatenation in Variables

- You can combine strings using + : 'ice cream' + '.' gives 'ice cream.'
- You cannot add a string and an integer with +: 'ice cream' + 2 causes a Type Error
- To combine a string and an integer in a print statement, use a comma: print('ice cream', 2)
- Integers can be added together: x = 3 + 2 stores 5

SECTION 3: Data Types

3.1 Numeric Data Types

There are three numeric types in Python:

- Integer (int) — whole numbers, positive or negative. Example: 12, -2
- Float — numbers with decimals. Example: 12 + 10.25 = 22.25 (float)
- Complex — imaginary numbers using 'j'. Example: 12 + 3j. Rarely used in practice

```
type(12)        # <class 'int'>
type(22.25)     # <class 'float'>
type(12 + 3j)   # <class 'complex'>
```

3.2 Boolean

- Only two values: True or False (capital first letter)
- Produced by comparison operations: 1 > 5 returns False, 1 == 1 returns True
- Useful in conditions (if statements, while loops)

3.3 Strings

- A string is a sequence of characters enclosed in single, double, or triple quotes
- Single quotes: 'hello' | Double quotes: "hello" | Triple quotes: '"multi-line text"'
- Use double quotes when your string contains an apostrophe: "I've always wanted..."
- Use triple quotes when your string contains both single and double quotes
- Strings are indexed starting at 0. Access characters with brackets: a[0], a[2:5], a[-1]
- Multiply strings: 'hello' * 3 produces 'hellohellohello'
- Concatenate strings: 'hello' + ' world' produces 'hello world'

3.4 Lists

- Lists store multiple values in a single variable, enclosed in square brackets [ ]
- Can hold mixed data types: ['vanilla', 3, ['scoops', 'spoon'], True]
- Lists are ordered and indexed starting at 0
- Lists are mutable — you can add, remove, and change items after creation
  - Add item: ice_cream.append('salted caramel')
  - Change item: ice_cream[0] = 'butter pecan'
  - Nested lists: access with double indexing ice_cream[2][1]

3.5 Tuples

- Tuples are ordered sequences stored in parentheses ( )
- Tuples are immutable — they cannot be modified, appended to, or deleted from after creation
- Trying to.append() to a tuple raises an Attribute Error
- Best used for data that should never change: city names, coordinates, fixed configurations
- Indexed the same way as lists: tuple_scoops[0]

3.6 Sets

- Sets are created with curly braces { } and store only unique values — no duplicates
- Sets are unordered, so you cannot access items by index
- Use case: comparing two collections to find overlap, differences, or unique values
  - | (pipe) — union: all unique values from both sets combined
  - & (ampersand) — intersection: values present in both sets
  - - (minus) — difference: values in one set but not the other
  - ^ (caret) — symmetric difference: values unique to each set (not shared)

3.7 Dictionaries

- Dictionaries store data in key-value pairs, enclosed in curly braces { }
- Each pair separated by a comma; key and value separated by a colon: {'name': 'Alex', 'age': 28}
- Access a value by its key (not by index): dict_cream['name'] returns 'Alex'
- Dictionaries can hold lists as values: {'favorites': ['mint choc chip', 'chocolate']}
- Methods: .values() — all values | .keys() — all keys | .items() — all key-value pairs
- Update a value: dict_cream['name'] = 'Christine'
- Add/update multiple pairs: dict_cream.update({'weight': '300lbs'})
- Delete a key-value pair: del dict_cream['weight']

SECTION 4: Data Type Conversion

4.1 Converting Numeric Types

Python cannot perform arithmetic between a string and an integer even if the string contains a number. You must convert the type first.

```
num_string = '7'          # this is a string
num_int = 7               # this is an integer
# num_int + num_string    # ERROR: can't add str + int
converted = int(num_string)  # converts '7' to 7
num_int + converted          # now works: 14
```

4.2 Converting Between Collection Types

- List to Tuple: tuple(my_list) — creates an immutable copy of the list
- List to Set: set(my_list) — removes duplicates and makes it unordered. CAUTION: changes the data
- Dictionary keys to list: list(my_dict.keys())
- Dictionary values to list: list(my_dict.values())
- String to list: list('hello world') — splits every character including spaces into a list element
- String to set: set('hello world') — gives only unique characters

Important: Converting to a set may fundamentally change your data by removing duplicates. Always be aware of side effects.

SECTION 5: Operators

5.1 Comparison Operators

Used to compare two values. Always return True or False (Boolean).

Operator	Meaning	Example
==	Equal to	`10 == 10 → True`
!=	Not equal to	`10 != 50 → True`
>	Greater than	`50 > 10 → True`
<	Less than	`10 < 50 → True`
>=	Greater than or equal to	`10 >= 10 → True`
<=	Less than or equal to	`10 <= 50 → True`

5.2 Logical Operators

- and — returns True only if BOTH conditions are True: 70 > 50 and 50 > 10 → True
- or — returns True if AT LEAST ONE condition is True: 10 > 50 or 50 > 10 → True
- not — reverses the result: not (50 > 10) → False

Logical operators can be combined with comparison operators and used with strings (Python compares alphabetically based on ASCII values).

5.3 Membership Operators

- in — returns True if a value is found within a sequence
  - 'love' in 'I love chocolate ice cream' → True
  - 2 in [1, 2, 3, 4, 5] → True
- not in — returns True if a value is NOT found in the sequence
  - 6 not in [1, 2, 3, 4, 5] → True

SECTION 6: If / Elif / Else Statements

6.1 Basic Structure

The if/elif/else statement lets your code make decisions based on conditions. Only one branch executes — the first one whose condition is True.

```
if 25 > 10:
    print('it worked')    # condition is True, this runs
elif 25 < 30:
    print('elif worked')  # only checked if the if above was False
else:
    print('none matched') # only runs if ALL conditions above were False
```

6.2 Rules

- You can have exactly ONE if statement and ONE else statement
- You can have as many elif (else-if) statements as you need between them
- Once a True condition is found, all remaining elif and else blocks are SKIPPED
- Code inside each block MUST be indented (Python uses indentation instead of braces)
- Conditions can use comparison operators, logical operators (and, or, not), and membership operators (in, not in)

6.3 One-Line If/Else (Ternary)

```
print('it worked') if 10 > 30 else print('it did not work')
```

This compact form is useful for simple conditions. The if and else are written on a single line.

6.4 Nested If Statements

You can place if statements inside the body of another if statement. This creates layered logic that can become very complex in real-world code.

```
if 25 > 10:
    print('outer if is true')
    if 10 > 5:
        print('nested if is also true')
```

SECTION 7: For Loops

7.1 How a For Loop Works

A for loop iterates over every item in a sequence (list, tuple, string, dictionary, etc.) one by one. For each iteration, a temporary variable holds the current item and the body of code executes.

```
integers = [1, 2, 3, 4, 5]
for number in integers:
    print(number)   # prints 1, then 2, then 3, etc.
```

- The temporary variable (number) can be named anything
- The body (indented code block) can perform any operations
- The loop body does not have to use the temporary variable — it can be used as a simple counter

7.2 Looping Over Dictionaries

Dictionaries require specifying what you want to iterate over:

```
# Loop over values only:
for cream in ice_cream_dict.values():
    print(cream)

# Loop over keys and values:
for key, value in ice_cream_dict.items():
    print(key, '-->', value)
```

7.3 Nested For Loops

A for loop inside another for loop. The inner loop completes ALL its iterations for each single iteration of the outer loop.

```
flavors = ['vanilla', 'chocolate', 'cookie dough']
toppings = ['fudge', 'oreos', 'marshmallows']

for f1 in flavors:
    for f2 in toppings:
        print(f1, 'topped with', f2)
```

Output: vanilla topped with fudge, vanilla topped with oreos, vanilla topped with marshmallows, then chocolate topped with fudge ... and so on.

SECTION 8: While Loops

8.1 How a While Loop Works

A while loop repeats a block of code FOR AS LONG AS a condition remains True. Unlike a for loop, a while loop does not iterate over a set sequence — it keeps going until told to stop.

```
number = 0
while number < 5:
    print(number)
    number = number + 1  # counter — without this, infinite loop!
```

- Always include a counter or mechanism to eventually make the condition False
- Forgetting to increment the counter creates an infinite loop

8.2 Break Statement

break immediately exits the while loop, even if the condition is still True.

```
while number < 5:
    if number == 3:
        break       # exits loop when number reaches 3
    number += 1
```

Output: 0, 1, 2 — the loop stops before printing 3.

8.3 Continue Statement

continue skips the rest of the current iteration and goes back to the top of the loop to check the condition again. Unlike break, it does not exit the loop.

```
while number < 5:
    number += 1
    if number == 3:
        continue  # skips printing 3
    print(number)
```

Output: 1, 2, 4 — 3 is skipped.

CAUTION: If continue is placed before the counter increment, it creates an infinite loop because the number never changes.

8.4 Else with While Loop

An else block attached to a while loop runs once when the condition becomes False naturally (i.e., not triggered by a break statement).

```
while number < 5:
    number += 1
else:
    print('no longer less than 5')
```

- If the loop exits via break, the else block does NOT run
- If the loop exits because the condition became False, the else block DOES run

SECTION 9: Functions

9.1 Defining and Calling Functions

A function is a reusable block of code that only runs when it is called. Functions help avoid repetition and make code easier to read and maintain.

```
def first_function():
    print('we did it')

first_function()   # calling the function — this runs it
```

- def is the keyword to define a function
- The function name follows def, then parentheses and a colon
- The body is indented
- Defining a function does NOT run it — you must call it explicitly

9.2 Arguments (Parameters)

Arguments let you pass data INTO a function so it can work with different values each time it's called.

```
def number_squared(number):
    print(number ** 2)

number_squared(5)    # outputs 25
number_squared(9)    # outputs 81
```

- ** is the exponent / power operator
- Multiple arguments are separated by commas: def func(number, power):
- When calling a function with multiple arguments, you MUST provide all of them

9.3 Arbitrary Arguments (*args)

Use *args when you don't know in advance how many arguments will be passed. The arguments are received as a tuple.

```
def number_args(*number):
    print(number[0] * number[1])

number_args(5, 6, 128)   # outputs 30 (5 * 6)
```

- The * tells Python to accept any number of positional arguments
- To pass a pre-built tuple to an *args function, prefix it with * when calling: number_args(*my_tuple)

9.4 Keyword Arguments

Keyword arguments allow you to pass arguments by name rather than position, giving you more control and readability.

```
def number_squared_custom(number, power):
    print(number ** power)

# Positional: number=5, power=3
number_squared_custom(5, 3)
```

```
# Keyword: order doesn't matter
number_squared_custom(power=3, number=5)
```

9.5 Arbitrary Keyword Arguments (**kwargs)

Use **kwargs when you don't know how many keyword arguments will be passed. Arguments are received as a dictionary.

```
def number_kw(**number):
    print('my number is ' + number['integer'])

number_kw(integer='2309', integer2='456')
```

- ** (double star) before the parameter name indicates arbitrary keyword arguments
- Inside the function body, access values by key: number['integer']

SECTION 10: Mini-Project — BMI Calculator

10.1 Project Goal

Build a command-line BMI calculator that takes user input (name, weight in pounds, height in inches), calculates BMI, and outputs a personalised message about the user's BMI category.

BMI Formula:

```
bmi = (weight * 703) / (height * height)
```

10.2 Getting User Input

The input() function displays a prompt and waits for the user to type something. By default, all input is stored as a string.

```
weight = int(input('Enter your weight in pounds: '))
height = int(input('Enter your height in inches: '))
```

- Wrap input() with int() to convert the string to an integer for arithmetic
- Without conversion, trying to multiply height * height raises a TypeError

10.3 BMI Classification Logic

```
bmi = (weight * 703) / (height * height)

if bmi < 18.5:
    print(name + ', you are underweight')
elif bmi <= 24.9:
    print(name + ', you are normal weight')
elif bmi <= 29.9:
    print(name + ', you are overweight')
elif bmi <= 34.9:
    print(name + ', you are obese')
elif bmi <= 39.9:
    print(name + ', you are severely obese')
else:
    print(name + ', you are morbidly obese')
```

- The elif chain checks each range in order — first True condition wins and the rest are skipped
- Using the user's name variable makes the output personalised and friendly

SECTION 11: Web Scraping with BeautifulSoup & Requests

11.1 What is Web Scraping?

Web scraping is the process of automatically extracting data from websites. Instead of manually copying and pasting, you write a Python script that reads the raw HTML of a webpage and pulls out exactly the data you want.

- Used for: price monitoring, research, data journalism, building datasets
- Libraries used in this course: requests (HTTP requests) and BeautifulSoup from bs4 (HTML parsing)
- More advanced libraries exist (e.g. Selenium for dynamic pages) but BeautifulSoup is the best starting point

11.2 Importing Libraries

```
from bs4 import BeautifulSoup
import requests
```

- If these don't import, install them: pip install bs4 requests in your terminal
- With Anaconda/Jupyter, bs4 and requests are typically pre-installed

11.3 Sending a GET Request

```
url =
'https://en.wikipedia.org/wiki/List_of_largest_companies_in_the_United_States_by
_revenue'
page = requests.get(url)
print(page.status_code)   # 200 means success
```

HTTP response codes:

- 200 — Success
- 204 — No content
- 400 — Bad request (invalid)
- 401 — Unauthorized
- 404 — Not found

11.4 Parsing HTML with BeautifulSoup

```
soup = BeautifulSoup(page.text, 'html.parser')
```

- page.text contains the raw HTML of the page as a string
- BeautifulSoup parses this HTML into a searchable Python object called 'soup'
- 'html.parser' is Python's built-in HTML parser — no extra install required

11.5 Finding HTML Elements

BeautifulSoup provides two main methods for finding elements:

- .find('tag') — returns the FIRST matching element
- .find_all('tag') — returns ALL matching elements as a list

```
# Find all tables
tables = soup.find_all('table')
table = tables[1]   # select the second table (index 1)

# Or find by class attribute
table = soup.find('table', class_='wikitable sortable')
```

- Real pages often have multiple tables — use find_all() and indexing to pick the right one
- Inspecting the page (right-click > Inspect in Chrome) shows you the HTML structure and class names

11.6 Extracting Table Headers

```
# <th> tags are table header cells
world_titles = table.find_all('th')
world_table_titles = [title.text.strip() for title in world_titles]
```

- List comprehension loops through every <th> element and extracts its text
- .text gets the text content of the tag
- .strip() removes leading/trailing whitespace and newline characters
- IMPORTANT: Use table.find_all() NOT soup.find_all() — otherwise you pull headers from ALL tables on the page

11.7 Extracting Table Row Data

```
# <tr> tags are table rows; <td> tags are data cells within rows
import pandas as pd
df = pd.DataFrame(columns=world_table_titles)

column_data = table.find_all('tr')

for row in column_data[1:]:   # [1:] skips the header row
    row_data = row.find_all('td')
    individual_row_data = [data.text.strip() for data in row_data]

    length = len(df)
    df.loc[length] = individual_row_data
```

- column_data[1:] — slicing skips the first row (which contains <th> headers, not data)
- .find_all('td') finds all data cells within each row
- df.loc[length] = row — appends each row to the DataFrame at the next available index

11.8 Exporting to CSV

```
df.to_csv(r'C:\path\to\output\companies.csv', index=False)
```

- df.to_csv() writes the DataFrame to a CSV file at the specified path
- index=False — prevents pandas from writing the DataFrame's row numbers as an extra column in the CSV
- Use r before the path string to treat backslashes as literal characters (raw string)

11.9 Full Web Scraping Workflow Summary

- 1. Import libraries: BeautifulSoup, requests, pandas
- 2. Assign URL to a variable
- 3. requests.get(url) — fetch the page HTML
- 4. BeautifulSoup(page.text, 'html.parser') — parse the HTML
- 5. Identify the correct table using find_all() + indexing or find() + class name
- 6. Extract headers using find_all('th') with list comprehension and .text.strip()
- 7. Create a blank pandas DataFrame with those headers as columns
- 8. Loop through rows find_all('tr')[1:], find_all('td') within each row
- 9. Append each row to the DataFrame using df.loc[len(df)]
- 10. Export to CSV using df.to_csv('path/filename.csv', index=False)

— End of Course Notes —