Lecture 2

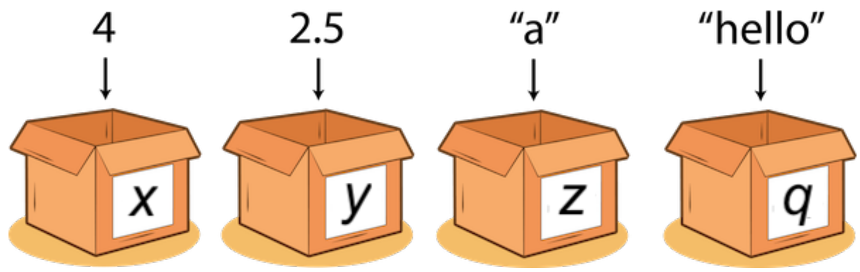
Python Fundamentals

Byeong-Hak Choe
bchoe@geneseo.edu
SUNY Geneseo

January 23, 2026

Python Basics

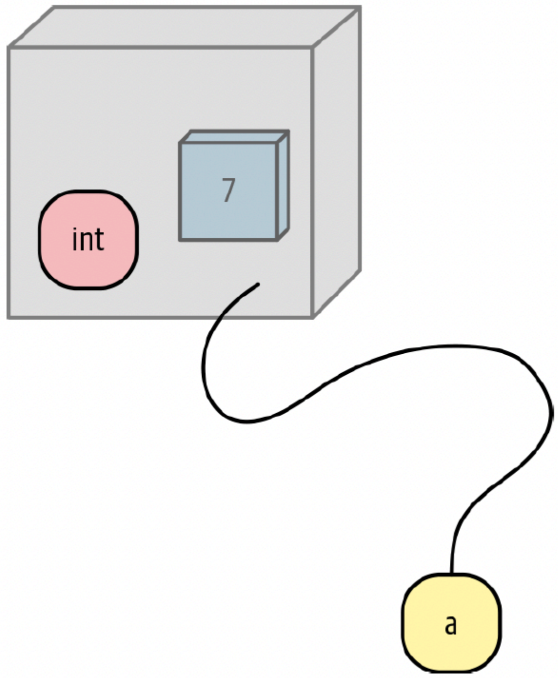
Values, Variables, and Data Types



- A **value** is a *literal* such as a number or text.
- Examples of values:
  - `352.3` → **float**
  - `22` → **int**
  - `"Hello World!"` → **str**

Variables

```python
a = 7
print(a)
```



- A **variable** is a **name** that refers to a value.
- Think of a variable like a *label* attached to a value.
    - A variable is **just a name** (not the value itself).

Objects

- Sometimes you will hear variables referred to as **objects**.
- Everything that is not a literal (like `10`) is an **object** in Python.

Assignment (=)

```python
# Here we assign the integer value 5 to the variable x.
x = 5

# Now we can use the variable x in the next line.
y = x + 12
y
```

- In Python, = means **assignment**:
  - **Right side** is evaluated first
  - The result is assigned to the **left side**

Note

✅ In math, = often means "equal."
✅ In Python, = means "store this value in the variable."

✍💬 Code and comment style

- Two guiding principles:
  - **Make things easy for your future self**
  - **Assume you will forget details later** → write it down now
- In Python, the comment character is #
  - Anything after # is ignored by the interpreter
  - Put comments **right above** the code they describe
- Use Markdown/text cells to explain:
  - What the code cell is doing,
  - Any assumptions/choices,
  - How to interpret output.

⌨️ Most Useful Google Colab Shortcuts

Windows

- **Ctrl + Enter**: Run cell
- **Alt + Enter**: Run cell and add new cell below
- **Ctrl + /**: Comment current line
- **Ctrl + Z**: Undo
- **Ctrl + Shift + Z**: Redo
- **Shift + ⬅️⬆️⬇️➡️**: Select text
- **Shift + Ctrl + ⬅️⬆️⬇️➡️**: Select to the beginning/end of the line

Mac

- **command + return**: Run cell
- **option + return**: Run cell and add new cell below
- **command + /**: Comment current line
- **command + Z**: Undo
- **command + shift + Z**: Redo
- **shift + ⬅️⬆️⬇️➡️**: Select text
- **shift + command + ⬅️⬆️⬇️➡️**: Select to the beginning/end of the line

Types

Name	Type	Mutable?	Examples
Boolean	bool	no	True, False
Integer	int	no	47, 25000, 25_000
Floating point	float	no	3.14, 2.7e5
Complex	complex	no	3j, 5 + 9j
Text string	str	no	'alas', "alack", '''a verse attack'''
List	list	yes	['Winken', 'Blinken', 'Nod']
Tuple	tuple	no	(2, 4, 8)
Bytes	bytes	no	b'ab\xff'
ByteArray	bytearray	yes	bytearray(...)
Set	set	yes	set([3, 5, 7])
Frozen set	frozenset	no	frozenset(['Elsa', 'Otto'])
Dictionary	dict	yes	{'game': 'bingo', 'dog': 'dingo', 'drummer': 'Ringo'}

- The **Type** column shows Python's official type name.
- **Mutable?**
  - ✅ mutable → can be changed after creation
  - ❌ immutable → cannot be changed after creation

One List, Many Types

```python
list_example = [10, 1.23, "like this", True, None]
print(list_example)
type(list_example)
```

- Common built-in types:
  - `int` (e.g., `10`)
  - `float` (e.g., `1.23`)
  - `str` (e.g., `"hello"`)
  - `bool` (e.g., `True`)
  - `NoneType` (e.g., `None`)
- A **list** can contain mixed types.

Square Brackets [ ] in Python

```python
vector = ['a', 'b']
vector[0]
```

- Use [ ] to create a **list**
- Use [ ] to access an element by **index**

Curly Braces {} in Python

```
{'a', 'b'}  # set
{'first_letter': 'a', 'second_letter': 'b'}  # dictionary (key:value pairs)
```

- `{}` is used to denote a **set** or a **dictionary**
- Use `{}` for **sets** and **dictionaries**

Parentheses ( ) in Python

```
num_tup = (1, 2, 3)
sum(num_tup)
```

- Use ( ) for **tuples**
- Use ( ) to pass **arguments** into functions

🧺 Data Containers in Python—List and Tuple

✅ List

- Stores **multiple values** in an ordered sequence
- 🔄 **Mutable**: **You can change it** after creation

```
fruits = ["apple", "banana", "orange"]
fruits.append("grape")
fruits[0] = "pear"
```

✅ Tuple

- Stores **multiple values** in an ordered sequence
- 🔒 **Immutable**: Cannot be changed after creation

```
geneseo_coords = (40.7158, 77.8136)
geneseo_coords[0]             # 👀 reading is OK
# geneseo_coords[0] = 100     # ❌ cannot modify
```

🗃️ Data Containers in Python—Dictionaries

```python
city_to_temp = {
    "Paris": 28,
    "London": 22,
    "New York": 18,
    "Seoul": 29,
    "Rochester": 10
}

city_to_temp["Paris"]           # 🔍 look up a value by key
city_to_temp["London"] = 32     # ✏️ update a value

city_to_temp.keys()         # 🔑 all keys
city_to_temp.values()       # 🌡️ all values
city_to_temp.items()        # 🧾 (key, value) pairs
```

- Stores values as **key→value** pairs
- Keys are used for fast lookup
- Useful when you want to create **associations** (“mapping”)

Running on Empty

```
lst = []
tup = ()
dic = {}
```

- Being able to create empty containers is sometimes useful, especially when using loops (e.g., `for`, `while`).
- **Q.** What is the type of an empty list?

Operators + − × ÷

```
a = 10
b = 3

a + b
a - b
a * b
a ** b
a / b
a // b
a % b
```

- All of the basic operators we see in mathematics are available to use:
  - + add
  - − subtract
  - * multiply
  - ** power
  - / divide
  - // integer divide (floor division)
  - % remainder

Operators Also Work for Lists and Strings

```
string_one = "This is an example "
string_two = "of string concatenation"
string_full = string_one + string_two
print(string_full)

string = "apples, "
print(string * 3)
```

- + **concatenates** (joins) strings
- * **repeats** a string multiple times

```
list_one = ["apples", "oranges"]
list_two = ["pears", "satsumas"]
list_full = list_one + list_two
print(list_full)
```

- + **concatenates** (combines) lists into a longer list

Casting Variables

```python
orig_number = 4.39898498
type(orig_number)
```

```python
mod_number = int(orig_number)
mod_number
type(mod_number)
```

- Casting changes type using built-in functions:
  - `int()`, `float()`, `str()`
  - If we try these, Python will do its best to interpret the input and convert it to the output type we'd like and, if they can't, the code will throw a great big error.
- **Q**. **Classwork 2.1**

✅❓ Booleans, Conditions, and `if` Statements

Booleans

```
10 == 20
10 == '10'
```

- Boolean values are either:
  - `True`
  - `False`

```
int(True)
int(False)
```

- In Python, Booleans can be converted to integers:
  - `int(True)` is 1
  - `int(False)` is 0

Boolean Operators

Operator	Description
x and y	True only if both are True
x or y	True if at least one is True
not x	Flips True ↔ False

Here, both x and y are boolean.

- Existing booleans can be combined by a **boolean operator**, which create a boolean when executed.

⚖️ Comparison Operators

Operator	Description
x == y	equal
x != y	not equal
x > y	greater than
x >= y	greater than or equal to
x < y	less than
x <= y	less than or equal to

Here, both x and y are variables.

〓 The Equality Operator ==

```
boolean_condition1 = 10 == 20
boolean_condition2 = 10 == '10'
```

- The `==` is an operator that compares the objects on either side and returns `True` if they have the same values
- **Q**. What does `not (not True)` evaluate to?
- **Q**. **Classwork 2.2**

Conditions → Boolean Expressions

```python
x = 10

print(x > 5)      # True
print(x == 3)     # False
print(x != 0)     # True
```

- A **condition** is an expression that returns `True` or `False`.

🚦🧾 Condition and `if` Statements

```python
name = "Geneseo"
score = 99

if name == "Geneseo" and score > 90:
    print("Geneseo, you achieved a high score.")

if name == "Geneseo" or score > 90:
    print("You could be called Geneseo or have a high score")

if name != "Geneseo" and score > 90:
    print("You are not called Geneseo and you have a high score")
```

- The real power of conditions comes when we start to use them in more complex examples, such as `if` statements.

The `in` Keyword: Membership Test

```
name_list = ["Lovelace", "Smith", "Hopper", "Babbage"]

"Lovelace" in name_list
"Bob" in name_list
```

- `in` checks whether something exists inside a list, string, etc.
- **Q**. Check if “a” is in the string “Wilson Ice Arena” using `in`. Is “a” in “Anyone”?

`if-else` Chain

```python
score = 98

if score == 100:
    print("Top marks!")
elif score > 90 and score < 100:
    print("High score!")
elif score > 10 and score <= 90:
    pass
else:
    print("Better luck next time.")
```

`if` Statements with `in`

```
fruits = ["apple", "banana", "cherry"]

favorite = "banana"

if favorite in fruits:
    print(f"{favorite} is available!")
else:
    print(f"{favorite} is not in the list.")
```

- The keyword `in` lets you check whether a value is present in a list, string, or other iterable.
- This works seamlessly inside an `if-else` structure.
- Useful for **membership tests** such as:
    - Validating if a company is in a stock list
    - Seeing if a word exists in a sentence

f-Strings (Formatted Strings) in Python

An **f-string** is a convenient way to create strings that include **variable values** directly inside the text.

✅ **Key idea:** Put an `f` before the quotation marks, then use `{}` to insert variables.

```
name = "Ada"
age = 20

message = f"My name is {name} and I am {age} years old."
print(message)
```

Indentation

```python
x = 10

if x > 2:
    print("x is greater than 2")
```

- In Python, indentation is required for code blocks, such as code inside:
    - a **user-defined function** (`def ...`),
    - a **conditional** (`if / elif / else`),
    - a **loop** (`for / while`).
- Indentation is how Python knows which lines belong to a block.
  It tells the interpreter what should run **inside** the block (e.g., inside an `if`) and what should run **after** the block ends.
- Standard Python style is **4 spaces per indentation level**.
    - In Google Colab, you might see **2 spaces**.
- **Q**. **Classwork 2.3**

✂️ Slicing Methods with Strings and Lists

✂️ Slicing Methods

0	1	2	3	4	5	6	7	8	9	10
H	e	l	l	o		W	o	r	l	d
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

- Slicing methods can apply for **strings**, **lists**, and **DataFrames**.
- With slicing methods, we can get a **subset** of the data object.
- Python is:
    - **zero-indexed** (things start counting from 0)
    - **left inclusive**
    - **right exclusive** when we specify a range

🧩 Slicing Patterns

```
letters = "abcdefghij"

letters[:]        # whole string
letters[3:]       # from index 3 to end
letters[:5]       # from start to index
letters[:-4]      # take the last 4 char
letters[2:7]      # index 2 to 6
letters[::2]      # step size 2
letters[::-1]     # reverse
```

- **Slice format**: `[start : end : step]`
  - `start` is **included**
  - `end` is **excluded**
  - `step` controls how many characters to skip

- ✅ **Important (when you “skip” numbers)**
  If you omit `start` or `end`, Python fills them in automatically:
  - If `start` is missing → slicing starts from the **beginning**
  - If `end` is missing → slicing goes to the **end**
  - <u>Example</u>: `letters[::2]` means “from the beginning to the end, taking every 2nd character.”

📏 Length of a String and a List

```python
string = "cheesecake"
len(string)
```

```python
list_of_numbers = [1, 2, 3, 4, 5]
len(list_of_numbers)
```

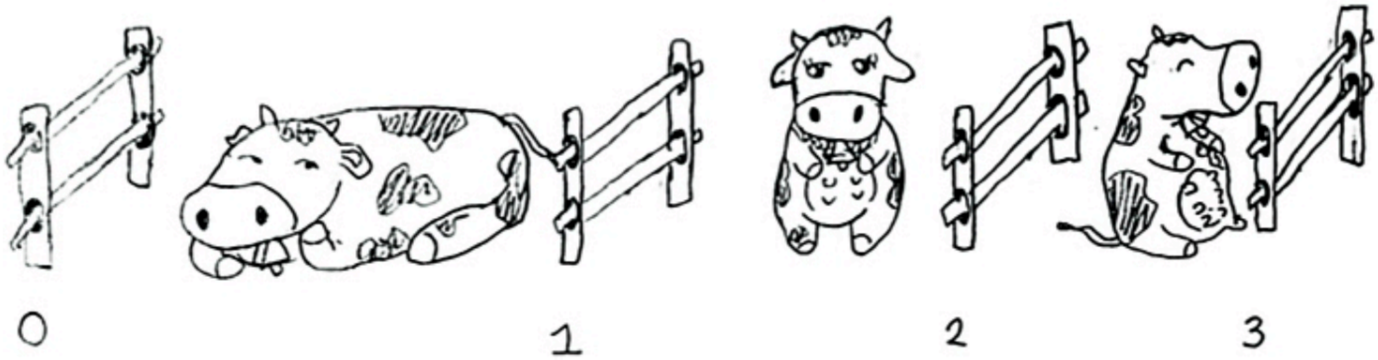
- Both **strings** and **list** objects support `len()`
- `len()` tells you how many items/characters are stored

✂️🧾 Slicing with Lists

```
list_example = ['one', 'two', 'three']
list_example[ 0 : 1 ]
list_example[ 1 : 3 ]
```

- Python is
  - a zero-indexed language (things start counting from zero);
  - left inclusive;
  - right exclusive when we are specifying a range of values.

✂️🧾 Slicing with Lists



- We can think of items in a list-like object as being fenced in.
    - The index represents the fence post.

🔢 Get an Item by `[index]`

```
suny = ['Geneseo', 'Brockport', 'Oswego', 'Binghamton',
        'Stony Brook', 'New Paltz']
```

```
suny[0]
suny[1]
suny[2]
suny[7]
```

```
suny[-1]
suny[-2]
suny[-3]
suny[-7]
```

✂️ Get an Item with a Slice

```
suny = ['Geneseo', 'Brockport', 'Oswego', 'Binghamton',
        'Stony Brook', 'New Paltz']
suny[0:2]    # A slice of a list is also a list.
```

```
suny[ : : 2]
suny[ : : -2]
suny[ : : -1]
```

```
suny[4 : ]
suny[-6 : ]
suny[-6 : -2]
suny[-6 : -4]
```

- **Q**. Classwork 2.4

⚙️ Functions, Arguments, and Parameters

Functions

```python
int("20")
float("14.3")
str(5)
int("xyz")
print("DANL 210")
```

```python
lst = [1,2,3,4]

type(lst)
len(lst)
max(lst)
sum(lst)
```

Common built-in functions you will use often:

- `type()` → data type
- `len()` → length
- `max()` → largest value
- `sum()` → total

- A function can take **inputs** (called **arguments**) and return an **output**.
- Python also lets you define your own functions with the `def` keyword.
- Later, we will use such **user-defined function** together with **pandas**.

⚙️📥 Functions, Arguments, and Parameters

```python
print("Cherry", "Strawberry", "Key Lime")
print("Cherry", "Strawberry", "Key Lime", sep = "!")
print("Cherry", "Strawberry", "Key Lime", sep=" ")
```

- To **call** a function, write its name followed by parentheses:
  - `function_name(...)`
- Inside the parentheses, you provide **arguments** (inputs), separated by commas.
- A **parameter** is the name used in the function definition for an expected input
  - Example: `sep` is a parameter of `print()`.
- A **default argument** is the value used automatically if you do not specify it.
  - For `print()`, the default separator is a space: `sep = " "`.
- **Q. Classwork 2.5**

🔁 Loop with `while` and `for`

➕= Updating a Variable with +=

```python
count = 1
count += 1
print(count)
```

```python
count = 1
count = count + 1
print(count)
```

- `+=` is a **shortcut assignment operator**
- It means: **take the current value and add something to it**
- E.g.,: `count += 1` means the same thing as `count = count + 1`.

🔄⏳ Repeat with `while`

```python
count = 1
while count <= 5:
    print(count)
    count += 1
```

How this loop works

1. Start with `count = 1`.
2. Check the condition: `count <= 5`
   - If it is `True`, run the loop body.
3. Print the current value of count.
4. Update count using `count += 1`.
5. Go back to step 2 and repeat.
6. The loop stops when `count <= 5` becomes `False`.

🙋⌨️ Asking the user for input

```python
stuff = input()
# Type something and press Return/Enter on Python Console
# before running print(stuff)
print(stuff)
```

- `input()` pauses the program and waits for the user to type something.
- Whatever the user types is returned as a **string**.
- This is useful when you want to make your code interactive.

🛑🔁 Cancel an Infinite Loop with `break`

```python
while True:
    user_input = input("Enter 'yes' to continue or 'no' to stop: ")
    if user_input.lower() == 'no':
        print("Exiting the loop. Goodbye!")
        break
    elif user_input.lower() == 'yes':
        print("You chose to continue.")
    else:
        print("Invalid input, please enter 'yes' or 'no'.")
```

- `While` loop is used to execute a block of code repeatedly until given boolean condition evaluated to `False`.
  - `while True` creates an **infinite loop**
- The loop runs forever unless you stop it using `break`
- `break` exits the loop immediately

⏭️🔁 Skip Ahead with `continue`

```
while True:
    value = input("Integer, please [q to quit]: ")
    if value == 'q': # quit
        break
    number = int(value)
    if number % 2 == 0: # an even number
        continue
    print(number, "squared is", number*number)
```

- `continue` skips the rest of the loop body **for the current iteration**
- Then Python jumps back to the top of the loop

🔁🔎 Iterate with `for` and `in`

- Use a `for` loop when you want to go through each item in:
  - a string
  - a list
  - a range (`range()`)
  - or any iterable object

🔁🚶 Repeat with a `for` Loop

`for` loop syntax (the pattern)

```
for <item_name> in <iterable>:
    <indented code block using <item_name>>
```

```
lst_nums = [0, 1, 2, 3, 4]

for num in lst_nums:
    print(num)
```

How this loop works

1. Take the first item in `lst_nums` → set `num = 0` → run `print(num)`
2. Take the next item → set `num = 1` → run `print(num)`
3. Repeat for `2`, `3`, `4`
4. Stop after the last item

Two Ways to Loop Through an Iterable

`while` approach

```python
word = 'thud'
offset = 0
while offset < len(word):
    print(word[offset])
    offset += 1
```

`for` approach

```python
word = 'thud'
for letter in word:
    print(letter)
```

- Which one do you prefer?

- **Q. Classwork 2.6**

🔢 Generate Number Sequences with `range()`

```
list( range(1, 4) )
list( range(0, 4) )
list( range(4) )

for x in range(0, 4):
    print(x)
```

- `range()` creates a sequence of integers without storing a full list
- This is memory-efficient and very common in `for` loops

- Syntax is similar to slicing:
- `range( start, stop, step )`
    - `start` defaults to 0
    - `step` defaults to 1
    - the sequence stops **right before** stop

🔢🧾 Get Index + Value with `enumerate()`

```
fruits = ["apple", "banana", "orange"]

# default: starts counting at 0
for i, fruit in enumerate(fruits):
    print(i, fruit)

# start counting at 1
for i, fruit in enumerate(fruits, start=1):
    print(i, fruit)
```

- `enumerate()` gives you **two things** while looping:
  - the **index** (`i`)
  - the **value** (`fruit`)

- Very handy when you want to **label**, **number**, or **track positions**.
- Syntax: `enumerate(iterable, start=0)`
  - `iterable` can be a list, tuple, string, etc.
  - `start` controls the first index (default is 0)

🛑🔂 Cancel a `for` Loop with `break`

```
word = 'thud'
for letter in word:
    if letter == 'u':
        break
    print(letter)
```

- `break` exits the loop immediately

⏭️🔂 Skip in a `for` Loop with `continue`

```python
word = 'thud'
for letter in word:
    if letter == 'u':
        continue
    print(letter)
```

- `continue` skips the current iteration and moves to the next one

🔄 Loop Control: `continue`, `pass`, `break`

```
for num in range(1, 6):

    if num == 2:
        continue   # skip printing 2

    if num == 3:
        pass       # do nothing, move on

    if num == 4:
        break      # exit the loop

    print(num)
```

- **`continue`** → skips to the next iteration
- **`pass`** → does nothing (useful as a placeholder)
- **`break`** → exits the loop completely
- **Q**. **Classwork 2.7**

📋📘⚡ List and Dictionary Comprehensions

❓📋 What is List Comprehension?

- A concise way to create or modify lists.
- Syntax: `[expression for item in iterable if condition]`

1. **Creating a List of Squares:**

```python
squares = [x**2 for x in range(5)]
```

2. **Filtering Items:**

```python
numbers = [1, 2, 3, 4, 5, 6]
evens = [x for x in numbers if x != 2]
```

❓📘 What is Dictionary Comprehension?

- A concise way to create or modify dictionaries.
- Syntax: `{key_expression: value_expression for item in iterable if condition}`

1. **Creating a Dictionary of Squares:**

```
squares_dict = {x: x**2 for x in range(5)}
```

2. **Filtering Dictionary Items:**

```
my_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
filtered_dict = {k: v for k, v in my_dict.items() if v != 2}
```

3. **Swapping Keys and Values:**

```
original_dict = {'a': 1, 'b': 2, 'c': 3}
swapped_dict = {v: k for k, v in original_dict.items()}
```

🛠️ Modifying Lists and Dictionaries

🔧 Adding an Item to a List

- **append()**: Adds an item to the end of the list.

```
my_list = [1, 2, 3]
my_list.append(4)
```

🔧 Deleting Items in a List

- **`remove()`**: Deletes the first occurrence of value in the list.

```
my_list = [1, 2, 3, 4, 2]
my_list.remove(2)
```

- **List Comprehension**: Removes items based on a condition.

```
my_list = [1, 2, 3, 4, 2]
my_list = [x for x in my_list if x != 2]
```

- **`del` statement**: Deletes an item by index or a slice of items.

```
my_list = [1, 2, 3, 4]
del my_list[1]
del my_list[1:3]
```

🔧 Adding/Updating Items in a Dictionary

- **`update()`**: Adds new key-value pairs or updates existing ones.

```python
my_dict = {'a': 1, 'b': 2}
my_dict.update({'c': 3})
my_dict.update({'a': 10})
```

🔧 Deleting Items in a Dictionary

- **Dictionary Comprehension**: Removes items based on a condition.

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
my_dict = {k: v for k, v in my_dict.items() if v != 2}
```

- **`del` statement**: Deletes an item by key.

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
del my_dict['b']
```

⚠️🛡️ Handle Errors with `try` and `except`

⚠️ Errors

```
short_list = [1, 2, 3]
positions = [0, 1, 5, 2]   # 5 is out of range

for i in positions:
    print(short_list[i])
```

- If we don't write our own **exception handler**, Python will:
  - print an error message (a *traceback*) explaining what went wrong, and
  - stop the program.

Exception Handlers (Why we need them)

- In Python, when something goes wrong, an **exception** is raised.
- If we're running code that *might* fail, we can add an **exception handler** so the program can respond nicely instead of crashing.
- Common examples:
  - Using an index that's out of range for a list/tuple
  - Looking up a key that doesn't exist in a dictionary

⚠️🛡️ Handle Errors with `try` and `except`

```python
short_list = [1, 2, 3]
positions = [0, 1, 5, 2]    # 5 is out of range

for i in positions:
    try:
        print(short_list[i])
    except:
        print("Index error:", i, "is not between 0 and", len(short_list) - 1)
```

- Use `try` to run code that **might fail**, and `except` to **handle the error gracefully**.
  - If an error occurs, Python **raises an exception** and runs the `except` block.
  - If no error occurs, Python **skips** the `except` block.
- **Q**. **Classwork 2.8**

📦⬇️ Importing and Installing Modules, Packages, and Libraries

📦⬇️ Importing Modules, Packages, and Libraries

- Python is a general-purpose programming language and is not specialized for numerical or statistical computation.
- The core libraries that enable Python to store and analyze data efficiently are:
  - `pandas`
  - `numpy`

🐼 pandas

- `pandas` provides `Series` and `DataFrames` which are used to store data in an easy-to-use format.

🔢 `numpy`

- `numpy`, numerical Python, provides the array block (`np.array()`) for doing fast and efficient computations;

📥🧾 `import` statement

- A **module** is basically a bunch of related codes saved in a file with the extension `.py`.
- A **package** is basically a directory of a collection of modules.
- A **library** is a collection of packages
- We refer to code of other module/package/library by using the Python `import` statement.

```
import LIBRARY_NAME
```

- This makes the code and variables in the imported module available to our programming codes.

📥🏷️ `import` statement with `as` or `from`

Keyword `as`

- We can use the `as` *keyword* when importing the module/package/library using its canonical names.

```
import LIBRARY as SOMETHING_SHORT
```

Keyword `from`

- We can use the `from` *keyword* when specifying Python module/package/library from which we want to `import` something.

```
from LIBRARY import FUNCTION, PACKAGE,
```

🛠️📦 `pip` tool

- To install a library `LIBRARY` on your Google Colab, run:

```
!pip install LIBRARY
```

- To install a library `LIBRARY` on your Anaconda Python, open your Spyder IDE, Anaconda Prompt, or Terminal and run:

```
pip install LIBRARY
```

- **Q. Classwork 2.9**