

# Lecture 2

## *Python Fundamentals*

Byeong-Hak Choe

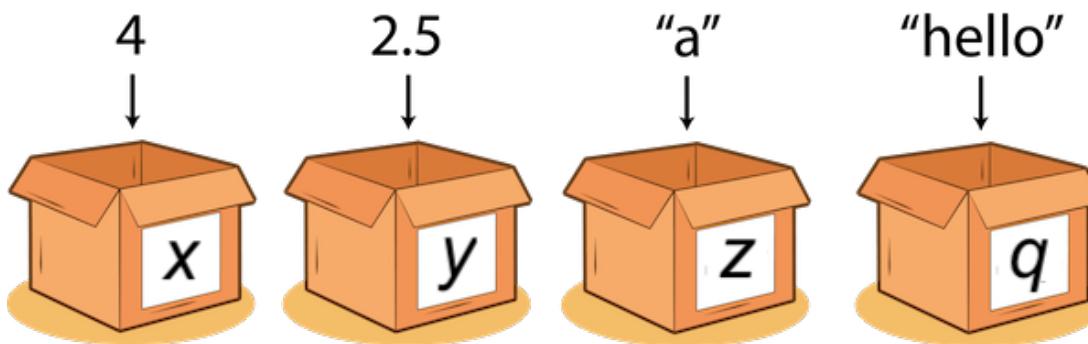
*bchoe@geneseo.edu*

SUNY Geneseo

January 23, 2026

# Python Basics

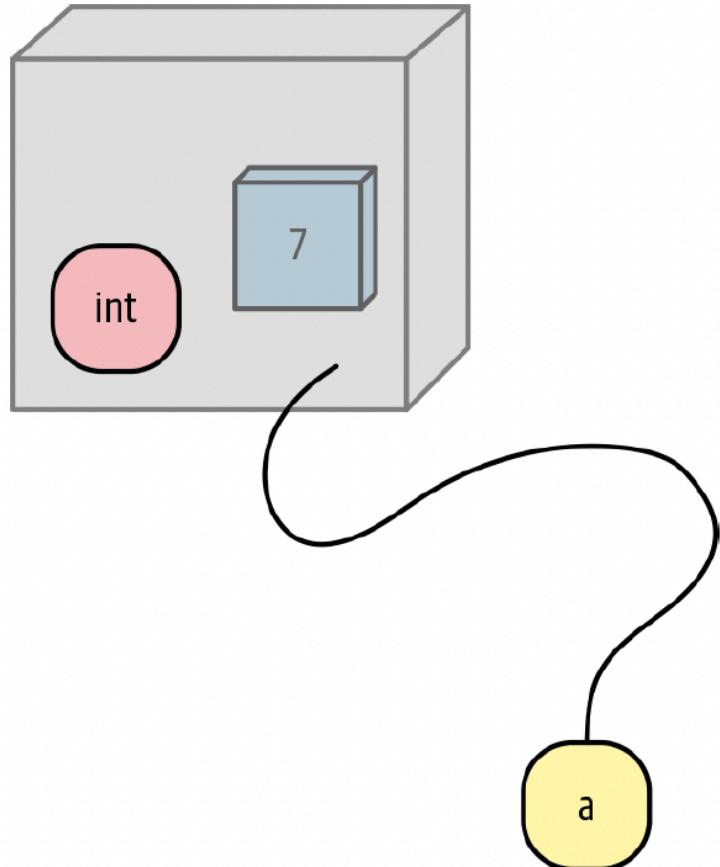
# Values, Variables, and Data Types



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

# Variables

```
1 a = 7  
2 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 ( = )

```
1 # Here we assign the integer value 5 to the variable x.  
2 x = 5  
3  
4 # Now we can use the variable x in the next line.  
5 y = x + 12  
6 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

```
1 list_example = [10, 1.23, "like this", True, None]
2 print(list_example)
3 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

```
1 vector = [ 'a', 'b' ]
2 vector[0]
```

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

# Curly Braces {} in Python

```
1 {'a', 'b'} # set  
2 {'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

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

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

## ✓ Tuple

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

```
1 geneseo_coords = (40.7158, 77.8136)
2 geneseo_coords[0]           # 🚫 reading is OK
3 # geneseo_coords[0] = 100  # ❌ cannot modify
```



# Data Containers in Python—Dictionaries

```
1 city_to_temp = {  
2     "Paris": 28,  
3     "London": 22,  
4     "New York": 18,  
5     "Seoul": 29,  
6     "Rochester": 10  
7 }  
8  
9 city_to_temp["Paris"]          # 🔎 look up a value by key  
10 city_to_temp["London"] = 32    # 🖊 update a value  
11  
12 city_to_temp.keys()          # 🔑 all keys  
13 city_to_temp.values()        # 💰 all values  
14 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

```
1 lst = []
2 tup = ()
3 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 + - × ÷

```
1 a = 10
2 b = 3
3
4 a + b
5 a - b
6 a * b
7 a ** b
8 a / b
9 a // b
10 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

```
1 string_one = "This is an example "
2 string_two = "of string concatenation"
3 string_full = string_one + string_two
4 print(string_full)
5
6 string = "apples, "
7 print(string * 3)
```

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

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

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

# Casting Variables

```
1 orig_number = 4.39898498  
2 type(orig_number)
```

```
1 mod_number = int(orig_number)  
2 mod_number  
3 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

```
1 10 == 20  
2 10 == '10'
```

```
1 int(True)  
2 int(False)
```

- Boolean values are either:
  - `True`
  - `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 $\leftrightarrow$ 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
<code>x == y</code>	equal
<code>x != y</code>	not equal
<code>x &gt; y</code>	greater than
<code>x &gt;= y</code>	greater than or equal to
<code>x &lt; y</code>	less than
<code>x &lt;= y</code>	less than or equal to

Here, both `x` and `y` are variables.

# = The Equality Operator ==

```
1 boolean_condition1 = 10 == 20  
2 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

```
1 x = 10
2
3 print(x > 5)      # True
4 print(x == 3)      # False
5 print(x != 0)      # True
```

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



# Condition and if Statements

```
1 name = "Geneseo"
2 score = 99
3
4 if name == "Geneseo" and score > 90:
    print("Geneseo, you achieved a high score.")
5
6
7 if name == "Geneseo" or score > 90:
8     print("You could be called Geneseo or have a high score")
9
10 if name != "Geneseo" and score > 90:
11     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

```
1 name_list = ["Lovelace", "Smith", "Hopper", "Babbage"]  
2  
3 "Lovelace" in name_list  
4 "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

```
1 score = 98
2
3 if score == 100:
4     print("Top marks!")
5 elif score > 90 and score < 100:
6     print("High score!")
7 elif score > 10 and score <= 90:
8     pass
9 else:
10    print("Better luck next time.")
```

# if Statements with in

```
1 fruits = ["apple", "banana", "cherry"]
2
3 favorite = "banana"
4
5 if favorite in fruits:
6     print(f"{favorite} is available!")
7 else:
8     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.

```
1 name = "Ada"
2 age = 20
3
4 message = f"My name is {name} and I am {age} years old."
5 print(message)
```

# Indentation

```
1 x = 10
2
3 if x > 2:
4     print("x is greater than 2")
```

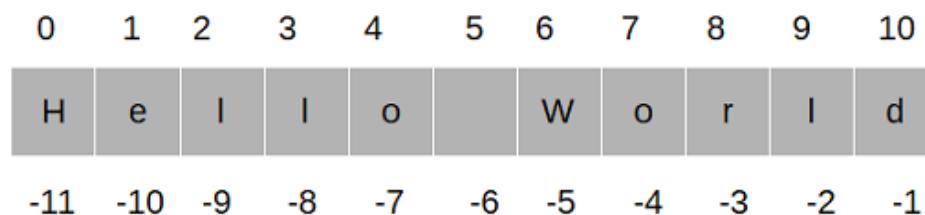
- In Python, indentation is required for code blocks, such as code inside:
  - a **user-defined function** (`def ...`),
  - a **conditional** (`if / elif / else`),
  - a **loop** (`for / while`).
- Indentation is how Python knows which lines belong to a block.  
It tells the interpreter what should run **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



- 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

```
1 letters = "abcdefghijklmnopqrstuvwxyz"  
2  
3 letters[:]      # whole string  
4 letters[3:]     # from index 3 to end  
5 letters[:5]     # from start to index  
6 letters[:-4]    # take the last 4 characters  
7 letters[2:7]     # index 2 to 6  
8 letters[::-2]    # step size 2  
9 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**
- Example: **letters[::-2]** means “from the beginning to the end, taking every 2nd character.”



# Length of a String and a List

```
1 string = "cheesecake"  
2 len(string)
```

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

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



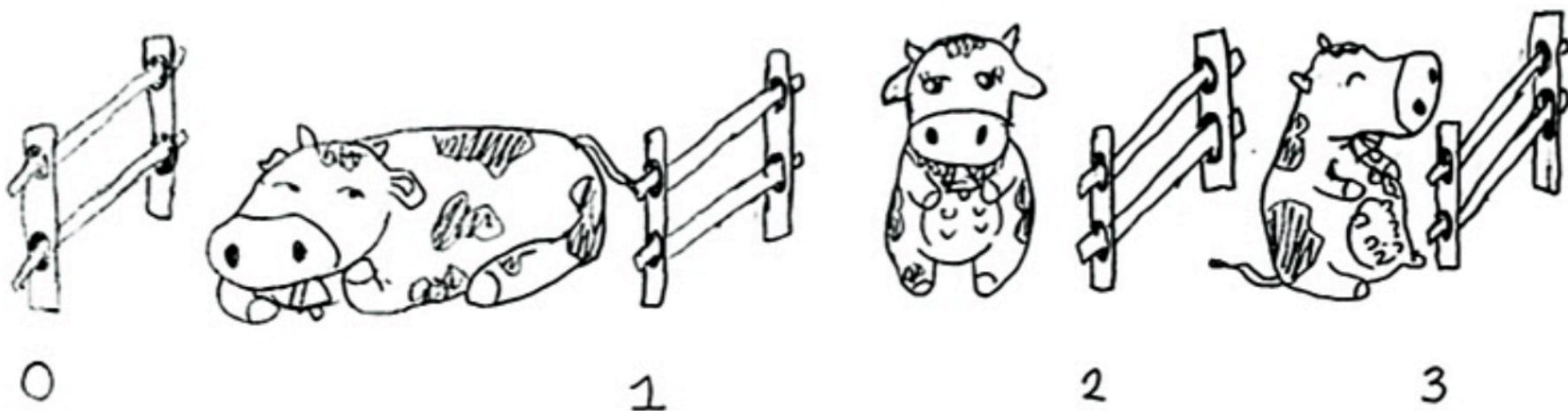
# Slicing with Lists

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

```
1 sunny = [ 'Geneseo', 'Brockport', 'Oswego', 'Binghamton',  
2           'Stony Brook', 'New Paltz' ]
```

```
1 sunny[ 0 ]  
2 sunny[ 1 ]  
3 sunny[ 2 ]  
4 sunny[ 7 ]
```

```
1 sunny[ -1 ]  
2 sunny[ -2 ]  
3 sunny[ -3 ]  
4 sunny[ -7 ]
```



# Get an Item with a Slice

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

```
1 sunny[ : : 2]  
2 sunny[ : : -2]  
3 sunny[ : : -1]
```

```
1 sunny[4 : ]  
2 sunny[-6 : ]  
3 sunny[-6 : -2]  
4 sunny[-6 : -4]
```

- Q. Classwork 2.4



# Functions, Arguments, and Parameters



# Functions

```
1 int("20")
2 float("14.3")
3 str(5)
4 int("xyz")
5 print("DANL 210")
```

```
1 lst = [1,2,3,4]
2
3 type(lst)
4 len(lst)
5 max(lst)
6 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

```
1 print("Cherry", "Strawberry", "Key Lime")
2 print("Cherry", "Strawberry", "Key Lime", sep = "!")
3 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 +=**

```
1 count = 1  
2 count += 1  
3 print(count)
```

```
1 count = 1  
2 count = count + 1  
3 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`

```
1 count = 1
2 while count <= 5:
3     print(count)
4     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

```
1 stuff = input()  
2 # Type something and press Return/Enter on Python Console  
3 # before running print(stuff)  
4 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`

```
1 while True:  
2     user_input = input("Enter 'yes' to continue or 'no' to stop: ")  
3     if user_input.lower() == 'no':  
4         print("Exiting the loop. Goodbye!")  
5         break  
6     elif user_input.lower() == 'yes':  
7         print("You chose to continue.")  
8     else:  
9         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`

```
1 while True:  
2     value = input("Integer, please [q to quit]: ")  
3     if value == 'q': # quit  
4         break  
5     number = int(value)  
6     if number % 2 == 0: # an even number  
7         continue  
8     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)

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

```
1 lst_nums = [0, 1, 2, 3, 4]  
2  
3 for num in lst_nums:  
4     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

```
1 word = 'thud'  
2 offset = 0  
3 while offset < len(word):  
4     print(word[offset])  
5     offset += 1
```

## for approach

```
1 word = 'thud'  
2 for letter in word:  
3     print(letter)
```

- Which one do you prefer?
- Q. **Classwork 2.6**

# Generate Number Sequences with `range()`

```
1 list( range(1, 4) )
2 list( range(0, 4) )
3 list( range(4) )
4 list( range(0, 4, 2) )
5
6 for x in range(0, 4):
7     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()`

```
1 fruits = ["apple", "banana", "orange"]
2
3 # default: starts counting at 0
4 for i, fruit in enumerate(fruits):
5     print(i, fruit)
6
7 # start counting at 1
8 for i, fruit in enumerate(fruits, start=1):
9     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**

```
1 word = 'thud'  
2 for letter in word:  
3     if letter == 'u':  
4         break  
5     print(letter)
```

- **break** exits the loop immediately



# Skip in a for Loop with `continue`

```
1 word = 'thud'  
2 for letter in word:  
3     if letter == 'u':  
4         continue  
5     print(letter)
```

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



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

```
1 for num in range(1, 6):
2
3     if num == 2:
4         continue    # skip printing 2
5
6     if num == 3:
7         pass        # do nothing, move on
8
9     if num == 4:
10        break      # exit the loop
11
12    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:

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

## 2. Filtering Items:

```
1 numbers = [1, 2, 3, 4, 5, 6]
2 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:

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

## 2. Filtering Dictionary Items:

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

## 3. Swapping Keys and Values:

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



# Modifying Lists and Dictionaries

# The . (dot) Operator on an Object

```
1 text = "GeNeSeO"  
2  
3 print(text.lower())    # method on a string object  
4 print(text.upper())    # another string method
```

- In Python, the **dot operator** (`.`) means:  
**“go inside this thing and access one of its members.”**
- Many data types come with **built-in methods** you can call using the dot operator.
- Example: For strings, `.lower()` is a method that returns a lowercase version of the string.
- Here, `text` is a **string object**, and `text.lower()` means:  
“use the **lower() method** that belongs to the **string type**.”



# Adding an Item to a List

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

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



# Deleting Items in a List

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

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

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

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

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

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



# Adding/Updating Items in a Dictionary

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

```
1 my_dict = {'a': 1, 'b': 2}
2 my_dict.update({'c': 3})
3 my_dict.update({'a': 10})
4 my_dict.update({'e': -1, 'f': 0})
```



# Deleting Items in a Dictionary

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

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

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

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

- **Q. Classwork 2.8**

⚠️  Handle Errors with `try`  
and `except`

# ⚠ Errors

```
1 short_list = [1, 2, 3]
2 positions = [0, 1, 5, 2]    # 5 is out of range
3
4 for i in positions:
5     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`

```
1 short_list = [1, 2, 3]
2 positions = [0, 1, 5, 2]    # 5 is out of range
3
4 for i in positions:
5     try:
6         print(short_list[i])
7     except:
8         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 2.9](#)



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



- **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.

```
1 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.

```
1 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.

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



## pip tool

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

```
1 !pip install LIBRARY
```

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

```
1 pip install LIBRARY
```



# The . (dot) Operator on a Library

- In Python, the **dot operator** (`.`) means:  
**“go inside this thing and access one of its members.”**
  - `module.name` → access something **defined inside** the module
  - `object.attribute` or `object.method()` → access a **property** or **function** of an object

# Example 1: `import sys`

```
1 import sys  
2  
3 print(sys.version)      # attribute: Python version info  
4 print(sys.path)         # attribute: module search paths
```

- ✓ Here, `sys` is a **module**, and `sys.version` and `sys.path` are things **inside** the `sys` module.

## Example 2: `import datetime`

```
1 import datetime  
2  
3 now = datetime.datetime.now()      # module.class.method()  
4 today = datetime.date.today()     # module.class.method()
```

- `datetime` (left side) is the **module**
- `datetime.datetime` is a **class inside** the module
- `.now()` is a **method** you can call from that class
- A **class** is a blueprint for creating objects that bundles **attributes** and **methods** together.

# Common Patterns to Remember

- `module.something`
- `module.class_name.method_name()`
- `object.method_name()`

***Read it like:***

“start with the thing on the left → use `.` to reach something inside → then (maybe) call it with `( )`”

- Q. **Classwork 2.10**