AIBridge

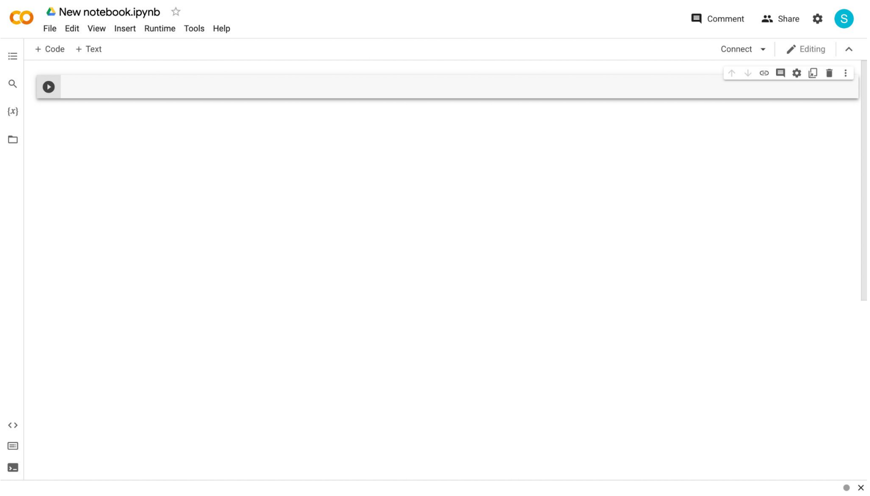
Lectures 1

Lecture Outline

- Google Colab
- General Python Syntax
- Variables
- Logic
- Control Flows
- I/O
- List manipulation
- OOP

Google Colab

- https://colab.research.google.com/
- Stores everything on Google Drive (no setup)
- Can be shared with others
- Run code within "cells"
- Code execution from top to bottom



Follow along as we work through the Python language

Lecture Outline

- Google Colab
- Getting Started
- Variables
- Logic
- Control Flows
- I/O
- List manipulation
- OOP

Getting Started

- Comments allow sections of the code to be more readable
    - Anything after a “#” is a comment
    - `# I am a comment!`
- Indents are required, serving the function of curly brackets (use tab key)

Lecture Outline

- Google Colab
- General Python Syntax
- Variables
- Logic
- Control Flows
- I/O
- List manipulation
- OOP

Variables - Overview

- A variable is a reserved place in memory given to a value
- Creating variables: `variable_name = value`
- Can be used anywhere after its assignment, but never before
- Can re-assign values as needed
- 7 types: Integer, Floating-point, String, Boolean, List, Tuple, and Dictionary

Variables - Names

- Cannot start with a number (`"3rd_variable"` will not work)
- Cannot include spaces (`"my variable"` will not work)
- Case sensitive (`"my_variable"` is different from `"mY_vArIaBle"`)
- Should be descriptive
- * Cannot be a keyword: https://www.w3schools.com/python/python_ref_keywords.asp
- * Good practice: all lowercase with underscores for spacing

Good: `number_of_datapoints, petal_widths, ...`

Invalid: `number of cases, 1status, ...`

Self-Test

What does the following code output?

```
variable_a = 25

varaible_b = 70

variable_a = 40

variable_b = variable_a

print(variable_b)
```

A. 70 ⇒ because the value of `variable_b` is set to be 70 in the second line

B. 40 ⇒ because the value of `variable_b` is set to be the same as `variable_a` which is 40

C. 25 ⇒ because the value of `variable_b` is set to be the same as `variable_a` which is 25

Self-Test

What does the following code output?

```
variable_a = 25

varaible_b = 70

variable_a = 40

variable_b = variable_a

print(variable_b)
```

A. **70 ⇒ because the value of `variable_b` is set to be 70 in the second line**

B. **40 ⇒ because the value of `variable_b` is set to be the same as `variable_a` which is 40**

C. **25 ⇒ because the value of `variable_b` is set to be the same as `variable_a` which is 25**

Variables - Integer

- Whole number
- + or -

```
my_first_number = 1
my_second_number = 5
my_third_number = -3
```

Variables - Floating-Point

- Can be a decimal
- Accurate within 2^{-55}

```
pi = 3.14159265358
petal_length = -3.5
```

Not this:

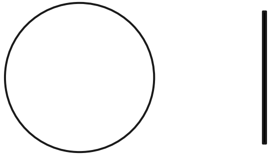
Variables - String

- A string of characters
- **Put in quotations `"  "` or `'  '`**
    - `Cannot mismatch these quotations`
- * Block string (multi-line string): three quotation marks
- * Special character (new line): `'\n'`

```
my_first_string = 's'
my_second_string = "string 2"
my_second_string = 'another string'
```

Variables - Boolean

- True or False (capitalize in Python)
- 1 or 0



```
my_first_boolean = True
my_second_boolean = False
```

Variables - List

[a, b, c, d, e]
0 1 2 3 4
-5 -4 -3 -2 -1

- A list of values
    - `my_list = [object_1, object_2, ...]`
    - Can include multiple different data types
    - `my_second_list = ["hello world", True, 5]`
- For a specific value in the list: `my_list[index]`
    - The index of the 1st item is 0,
    - `a_value = my_second_list[2] # gets the THIRD value in the list`
    - * The index for the last number -1 if using negative index

Self-Test

What does the following code output?

```
my_list = [21, 22, 23, 24, 25]

value = my_list[2]

print(value)
```

A. **22 ⇒ because value is set to the second item in the list**

B. **23 ⇒ because value is set to the third item in the list**

Self-Test

What does the following code output?

```
my_list = [21, 22, 23, 24, 25]

value = my_list[2]

print(value)
```

A. **22 ⇒ because value is set to the second item in the list**

B. **23 ⇒ because value is set to the third item in the list**

* Variables - Tuple

- Works the same as a list, but can’t be changed
- Can contain multiple different data types

```
my_first_tuple = (object_1, object_2, ...)

my_second_tuple = (22, "hello!", True, 3.1415)

a_value = my_second_tuple[2] # gets the THIRD value in the tuple
```

* Variables - Dictionary

- A list of values with custom keys that are indices, like a list but indices are keys and not positions

```
my_dictionary={'apple':'fruit', 'banana':'fruit', 'cabbage':'vegetable',
'dragonfruit':'fruit','eggplant':'vegetable'}

print(my_dictionary['cabbage'])
```

Variable Type Conversion

- Types are named: int, float, str, bool, list, tuple
- Convert types of variables to other types

```
my_float = float(my_object) #gives object in float form if possible
```

- Compatible types:
    - int-float (float to int rounds down)
    - str → int/float
    - * list-tuple
    - * boolean-int/float (0 -> False, anything else -> True)
    - * str-list/tuple (only converts str to list/tuple of single characters)

Lecture Outline

- Google Colab
- General Python Syntax
- Variables
- Logic
- Control Flows
- I/O
- List manipulation
- OOP

Logic - Basic Arithmetic Operations

+	-	*	**	/	//	%
Addition	Subtraction	Multiplication	Exponentiation	Division (turns int to float)	Floor Division (rounds down the quotient)	Modulus (returns the remainder)
`x + y` `1 + 2 = 3`	`x - y` `2 - 1 = 1`	`x * y` `2 * 3 = 6`	`x ** y` `2 ** 3 = 8`	`x / y` `8 / 2 = 4.0`	`x // y` `9 // 4 = 2`	`x % y` `10 % 4 = 2`

```
y = x + 1
```

Logic - if, elif, and else

```
if statement_1:
  Code segment 1
elif statement_2: # elif means else if
  Code segment 2
else:
  Code segment 3
```

Logic example code

```python
x = 3
y = 4
if x == y:
  print('x is equal to y')
elif x > y:
  print('x is greater than y')
else:
  print('x is less than y')
```

Logic Operations - $==, !=, <, >, <=, >=$

== != < > <= >=

== Gives True if the two sides are exactly the same (1 == 1, True)

!= gives True if the two sides are NOT the same (2 != 1, True)

```
print(3 == 3) # True
print(3 == 4) # False

print(3 < 3) # False
```

Logic Operations - not, and, or

- not - negates expression `not 9 + 10 == 21` is True
- and - combines expressions, only true if both are `1==1 and 1==2` is False
- or - if at least one of them are true `1==1 or 1==2` is True

```
x = 1
y = 1
if x < y or x == y:
  print("x is less than or equal to y")
```

Self-Test

Which of these conditions are successfully passed?

```
petal_width = 1.8
petal_length = 3.5

if petal_width < 3 or petal_length < 3:
 print("condition 1 passed")

if petal_width < 3 and petal_length < 3:
 print("condition 2 passed")

if petal_width < 3:
 if petal_length < 3:
   print("condition 3 passed")
```

Self-Test

Which of these conditions are successfully passed?

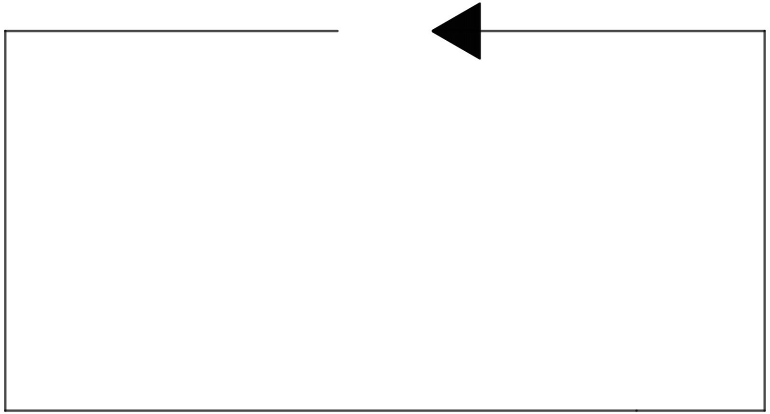
```
petal_width = 1.8
petal_length = 3.5

if petal_width < 3 or petal_length < 3:
 print("condition 1 passed")

if petal_width < 3 and petal_length < 3:
 print("condition 2 passed")

if petal_width < 3:
 if petal_length < 3:
   print("condition 3 passed")
```

Lecture Outline

- Google Colab
- General Python Syntax
- Variables
- Logic
- Control Flows
- I/O
- List manipulation
- OOP

Control flows

- Very important
- Two types: for and while



Control flows - Hypothetical Scenario

We have this very large list of 11 words:

```
words = ["Lorem", "ipsum", "dolor", "sit", "amet", "fusce",
"rhoncus", "mi", "viverra", "velit", "mattis"]
```

How do we access and print out every word?

Control flows - Hypothetical Scenario

```
word_list = ["Lorem", "ipsum", "dolor", "sit", "amet", "fusce", "rhoncus", "mi", "viverra", "velit", "mattis"]

print(word_list[0])
print(word_list[1])
print(word_list[2])
print(word_list[3])
print(word_list[4])
print(word_list[5])
print(word_list[6])
print(word_list[7])
print(word_list[8])
print(word_list[9])
print(word_list[10])
```

Horribly inefficient

A lot of tedious manual coding

Completely unscalable (what if there were 70 words)

Control flows - For

- How to use: `for object in iterable:`
    - String, list, range, etc.
    - Need indentation

```
for number in range(0, 11): #range goes through 0, 1, 2, … 10
  #this loop repeats 11 times and number changes to each number
  print(word_list[number])
```

Control flows - For

```
word_list = ["Lorem", "ipsum", "dolor", "sit", "amet", "fusce", "rhoncus", "mi", "viverra", "velit", "mattis"]

for number in range(0, 11): #range goes through 0, 1, 2, …, 10
  #this loop repeats 11 times and number changes to each number
  print(word_list[number])

for word in word_list:
  #this loop does the exact same thing but with less typing
  print(word)
```

Self-Test

```
big_list = ["Lorem", "Ipsum", "Dolor", "Sit", "Amet",
"Consectetur", "Adipiscing", "Elit", "Sed"]
```

Which of the following code blocks will print out everything in the list?

```
a.

for word in big_list:
 print(word)
```

```
b.

for i in range(9):
 print(big_list[i])
```

```
c.

for word in big_list:
 print(big_list[word])
```

Self-Test

```
big_list = ["Lorem", "Ipsum", "Dolor", "Sit", "Amet",
"Consectetur", "Adipiscing", "Elit", "Sed"]
```

Which of the following code blocks will print out everything in the list?

```
a.

for word in big_list:
 print(word)
```

```
b.

for i in range(9):
 print(big_list[i])
```

```
c.

for word in big_list:
 print(big_list[word])
```

Control flows - Indentation

```python
a_list = [3, 22, 1, 73, 40, 3, 19]
sum = 0

for i in range(0, 7):
    sum = sum + a_list[i]
    sum /= 2.4
    sum *= -1

    print(a_list[i])

print(sum)
```

Inside loop because of indentation

Control flows - While

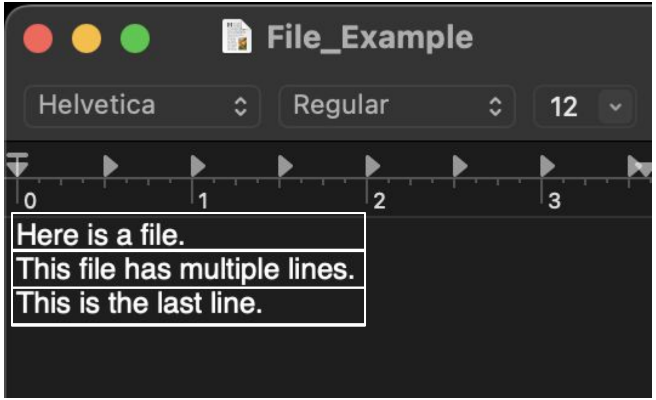
- How to use: `while` *`statement`*`:`
    - The loop repeats as *`statement`* is true
    - Needs indentation

```
my_number = 0
while my_number < 6:
  print(my_number)
  my_number = my_number + 1
```

Lecture Outline

- Google Colab
- Getting Started
- Variables
- Logic
- Control Flows
- I/O
- List manipulation
- Functions and Modules

I/O
Standard Input



Here is a file.
This file has multiple lines.
This is the last line.

- Input from console: `input('prompt')`
- Open file: `file_object=open(file, mode)`
  - `'r'` is read and `'w'` is write for the mode
  - `read(), readline(), readlines()`
- **Always** close file: `file_object.close()`

```
"""Here is a file.
This file has multiple lines.
This is the last line."""
```

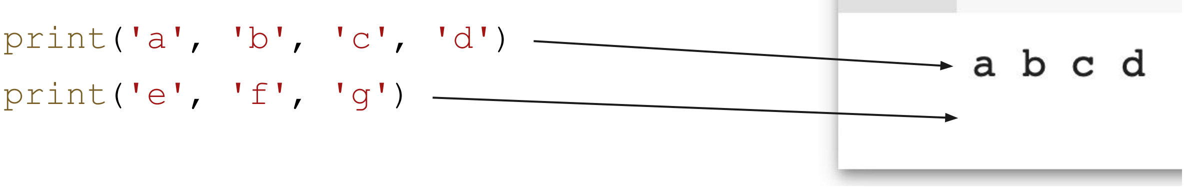
```
"Here is a file."
"This file has multiple lines."
"This is the last line."
```

```
["Here is a file.",
"This file has multiple lines.",
"This is the last line."]
```

I/O

Standard Output

- Output to Console: `print(object1, object2, ...)`



```
print('a', 'b', 'c', 'd')
print('e', 'f', 'g')
```

```
a b c d
```

- Open file: `file_object=open(file, mode)`
- `write()`
- Always close file

Note: This removes any existing file with that name

Lecture Outline

- Google Colab
- Getting Started
- Variables
- Logic
- Control Flows
- I/O
- List Manipulation
- Functions and Modules

List Manipulation

- Indexing
- List operations
- String/list interop
- Multidimensional lists

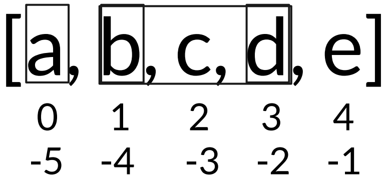
List Manipulation Indexing

- Single indexing

```
list_name[0]

list_name[-2]
```

- List slicing

```
list_name[1:4]
```



[a, b, c, d, e]

0 1 2 3 4

-5 -4 -3 -2 -1

Self-Test

What does the following code output?

```
arr = [4, 5, 6, 101, 102, 103, 104, 105]

new_arr = arr[2:6]
print(new_arr)
```

A. **[5, 6, 7, 101, 102, 103, 104, 105]**

B. **[6, 7, 101, 102, 103, 104, 105]**

C. **[6, 101, 102, 103, 104]**

D. **[6, 101, 102, 103]**

Self-Test

What does the following code output?

```
arr = [4, 5, 6, 101, 102, 103, 104, 105]

new_arr = arr[2:6]
print(new_arr)
```

A. **[5, 6, 7, 101, 102, 103, 104, 105]**

B. **[6, 7, 101, 102, 103, 104, 105]**

C. **[6, 101, 102, 103, 104]**

D. **[6, 101, 102, 103]**

List Manipulation - List operations

- https://docs.python.org/3/tutorial/datastructures.html
- `my_list.append(object) #adds object to the end of my_list`
- `my_list.remove(object) #removes the first occurence of object`
- `my_list.insert(i, object) #adds object to index i in my_list`
- `my_list.pop(i) #removes the object at index i`
- `list_1 + list_2 #adds list_2 to the end of list_1`
- `my_list.count(object) #gives you the number of times object occurs`
- `my_list.sort() #sorts list in ascending order`
- `len(my_list) #gives you the length of my_list`
- `min(my_list), max(my_list) #gives smallest and largest value inmy_list`

* Multidimensional lists

- Lists can contain other lists

```
my_list=[[1,2,3], [4,5,6], [7,8,9]] #list nested twice, so 2 dimensional list
print(my_list[0])
print(my_list[0][0]) #here, my_list[0] is a list, so we can index it
print(my_list[-2][0:3])
my_list_2=[[[[1,2],[3,4]],[[5,6],[7,8]]],[[[9,10],[11,12]],[[13,14],[15,16]]]]
#list nested four times, so 4 dimensional list
print(my_list_2[0][1][-2][0])
print(my_list_2[1][-1][1][0])
```

List Manipulation - String/List Interop

- Strings also have indexing (same as if it’s a list of all single chars)

```
''.join(my_list) #joins all objects (must be strings) in my_list
print('a string'[0])
print('a string'[1])
print('a string'[-1])
my_string.split(substring) #at each point where substring occurs, splits
my_string, returns list
print('this is a string'.split(' '))
```

Lecture Outline

- Google Colab
- Getting Started
- Variables
- Logic
- Control Flows
- I/O
- List manipulation
- Functions and Modules

Functions

- What is a function?
    - Reusable block of code with optional inputs and outputs
    - Like a factory
- `print(___)` is a function
- Built-in functions
- Imported functions
- Custom functions

Functions - Create Functions

```
def function_name(param_1, param_2, ...):
    ...
    return(value)

function_name(p1,p2,e...)
```

Functions - Create Functions

```
def factorial(input_int):
  total = 1
  for n in range(input_int):
    total = total * (n + 1)
  return(total)
  print('factorial computed')

print(factorial(5))
```

Functions - Built-in Functions

- Python already has these functions
- Full list at https://docs.python.org/3/library/functions.html
- For example: `print`, `len`, `range`, etc.

Modules

- Import third-party modules containing functions, etc.

```
import module_name
from module_name import function_name

# This imports a module as a nickname (alias)
import sklearn as skl
```