Python Review #1

BIPN 162

Objectives for today

- Review the top 10 programming features you need to know
- Establish guidelines for using AI assistants in this course (and beyond)

Top #10 programming features you need to know

1. Variables
2. Data types
3. Lists
4. Dictionaries
5. Conditionals
6. Indentation
7. Functions
8. Loops
9. Files
10. Modules

Creating new variables

- Names are always on the left of the `=`, values are always on the right
- Pick names that describe the data / value that they store
- Make variable names as **descriptive** and **concise** as possible
- We use an equal sign to assign the value to a name, but it's not the same thing as saying they are equal, mathematically.
- Variables cannot be Python keywords:

```
>>> import keyword
>>> print(keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def',
 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lamb
da', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
>>>
```

Built-in simple variable types in Python

Type	Example	Description
`int`	x = 1	integers (i.e., whole numbers)
`float`	x = 1.0	floating-point numbers (i.e., real numbers)
`complex`	x = 1 + 2j	Complex numbers (i.e., numbers with real and imaginary part)
`bool`	x = True	Boolean: True/False values
`str`	x = 'abc'	String: characters or text
`NoneType`	x = None	Special object indicating nulls

Integers, strings, floats

function to convert to integer

- **Integers** (`int`): any whole number
- **Float** (`float`): any number with a decimal point (floating point number)
- **String** (`str`): letters, numbers, symbols, spaces
  - Represented by matching beginning & ending quotes
  - Quotes can be single or double; use single *within* double
  - Use \ to ignore single quote
  - Concatenate strings with +

```
data_point = 8.02
```

→ **single variable** (int, float, string)

```
data = [8.38,3.34,6.35]
```

→ **data structure** (list, tuple, dictionary)

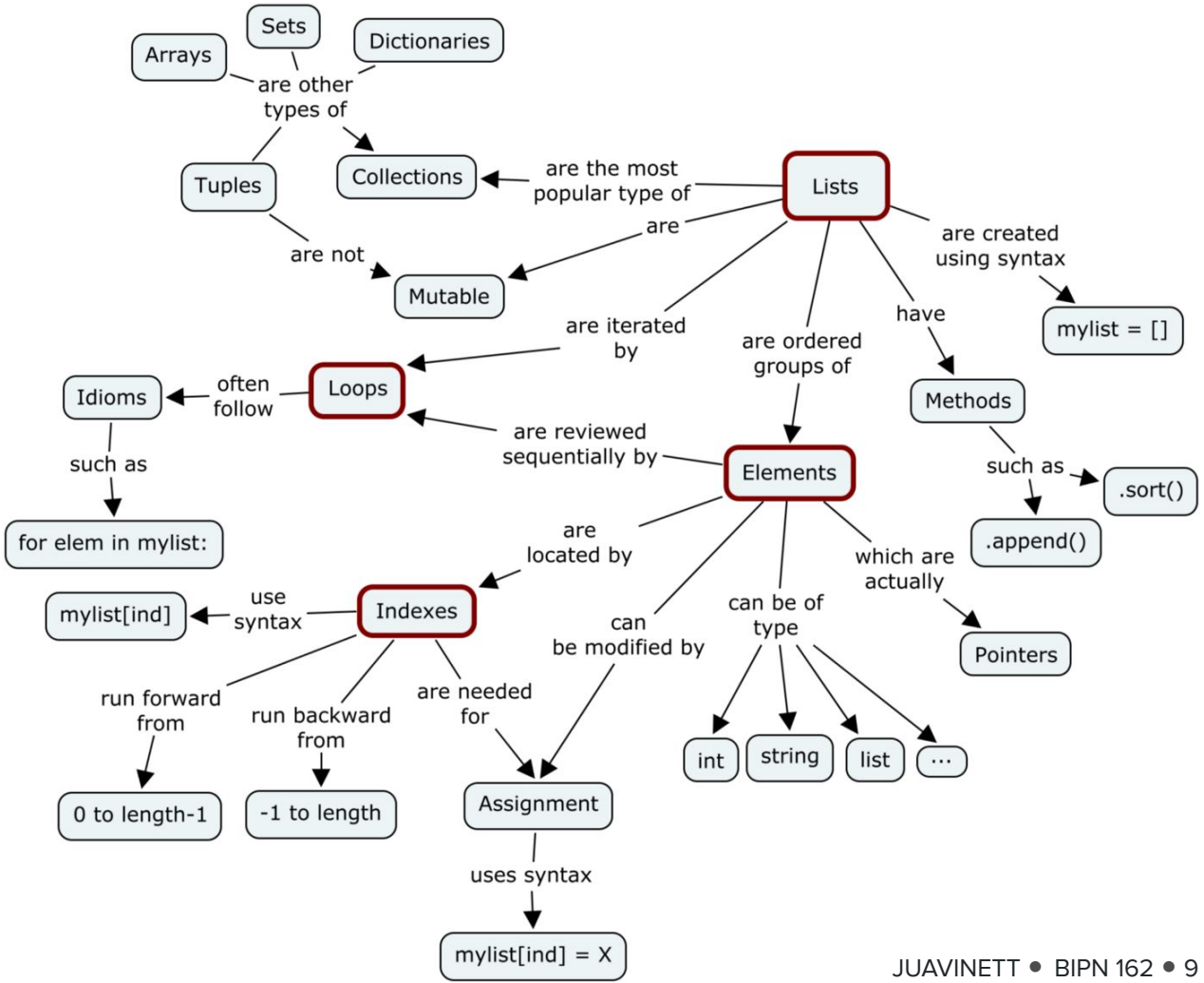
```
big_data = [data_1,data_2,...]
```

→ **array** or **dataframe**

Which objects are immutable?

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

Mutable vs Immutable Objects in Python | by megha mohan | Medium



[Mutable vs Immutable Objects in Python | by megha mohan | Medium]

Lists are flexible & efficient containers for heterogeneous data

- Lists are **mutable**: we can change individual elements of the list
- Denoted by brackets & elements are separated by commas

```
my_list = [‘apples’, ‘bananas’, ‘oranges’]
```

- Check the length of your list by using **`len(my_list)`**
- Use **`my_list.append()`** to add elements to a list
- Remove elements by index using **`del my_list[2]`**
- Remove elements by value by using **`my_list.remove(‘oranges’)`**
- Sort by using **`my_list.sort()`**

Indexing lists

```
my_list = [1,2,5,2,3]

my_list[1] = 2

my_list[-1] = 3

my_list[5] =

    IndexError
```

Index number

Allows you to count from the end (could be -2, etc.)

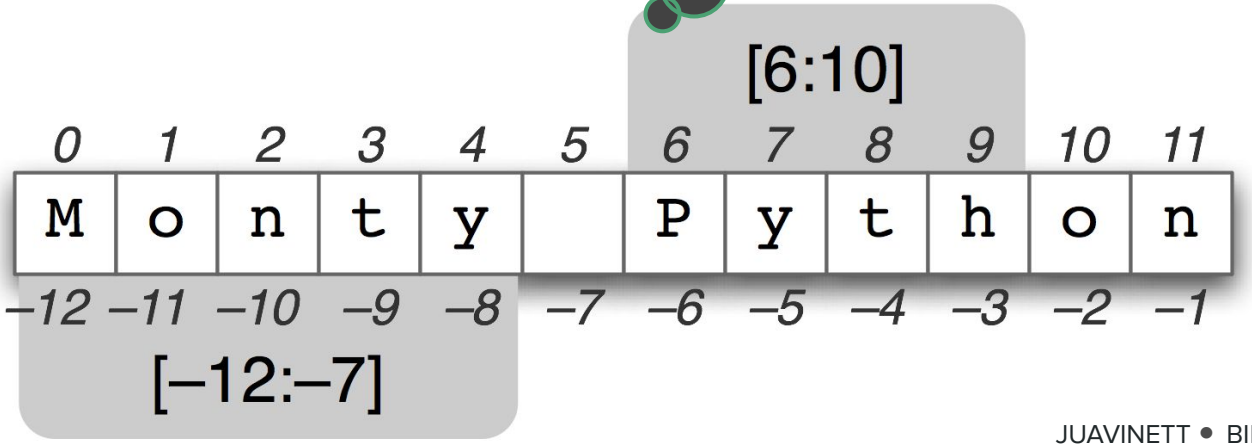
Shown if you try to get an index that doesn't exist

Slicing lists

```
my_list[0:2]

my_list[1:3]

my_list[:3]

my_list[3:]

my_list[:]
```

`[included:excluded]`
It doesn't show you the stop element (it shows you elements with indices 0 & 1)

One way to remember how slices work is to think of the indices as pointing between characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of n characters has index n.



Dictionaries link keys to values

- Denoted by **curly braces** and elements are separated by **commas**. Assignments are done using **colons**.

key

```
>>> capitals =  {'US' : 'DC' , 'Spain':'Madrid',
'Italy:'Rome'}
```

value

```
>>> capitals['US']

>>> 'DC'
```

- You'll get a Key Error if you ask for a key that doesn't exist
    - Use **`'Germany' in capitals`** to check

When dictionaries are useful

1. Flexible & efficient way to associate labels with heterogeneous data
2. Use where data items have, or can be given, labels
3. Appropriate for collecting data of different kinds (e.g., name, addresses, ages)

Top #10 programming features you need to know

1. Variables
2. Data types
3. Lists
4. Dictionaries
5. Conditionals
6. Indentation
7. Functions
8. Loops
9. Files
10. Modules

Operators in Python

Operators are special symbols that carry out arithmetic or logical computation.

Type of operator	Examples
assignment	`a = 6`
arithmetic (math)	`2 * 3`
logic (boolean)	`True and False`
comparison	`a != 6`
identity	`a is 6`
membership	`'a' in 'cat'`

Basic conditional operators in Python

Symbol	Operation	Usage	Outcome
==	Is equal to	`10==5*2`	`True`
!=	Is not equal to	`10 != 5*2`	`False`
>	Is greater than	`10 > 2`	`True`
<	Is less than	`10 < 2`	`False`
>=	Greater than *or* equal to	`10 >= 10`	`True`
<=	Less than *or* equal to	`10 <= 10`	`True`

Boolean variables store `True` (1) or `False` (0) and are the basis of all computer operations.

Syndey Padua:
https://sydneypadua.com/2dgoggles/happy-200th-birthday-george-boole/

`if` statements syntax

```
if condition:
    print('condition met')
    print('nice work.')
print('not in the block')
```

you need a colon here!

indented
by 4 spaces
(or tab)

block

`if/else` statement syntax

```
if condition:
        print('condition met')
        print('nice work.')
else:
        print('condition not met')
```

you need a colon here!

One more conditional: `elif`

- Short for “else if”
- Enables you to check for additional conditions ➔ *necessary if there is more than two outcomes*

```
condition_1 = False
condition_2 = True

if condition_1:
    print('Condition 1 is true.')
elif condition_2:
    print('Condition 2 is true.')
else:
    print('Both Condition 1 and 2 are false.')
```

Functions are pieces of code that are designed to do ***one task***

Functions take in inputs, process those inputs, and *possibly* return an output.

Python has *built-in* functions, but we can also write our own!

`function` syntax

function name

```
def function(value):
    print(value)
```

colon

function body

indented by 4 spaces (or tab)

`function` syntax

input arguments (these can be variables or default arguments)

```
def function(b):

    a = b**2

    return a


a = function(6)
```

`return` to retrieve a variable outside of a function *(what happens in the function stays in the function)*
ALSO ENDS THE FUNCTION!

call to function giving it the argument and saving the returned variable as a

`function` syntax

```
def function(b):
    c = b**2
    a = c * 2
    return a
a = function(6)
print(c)
????
```

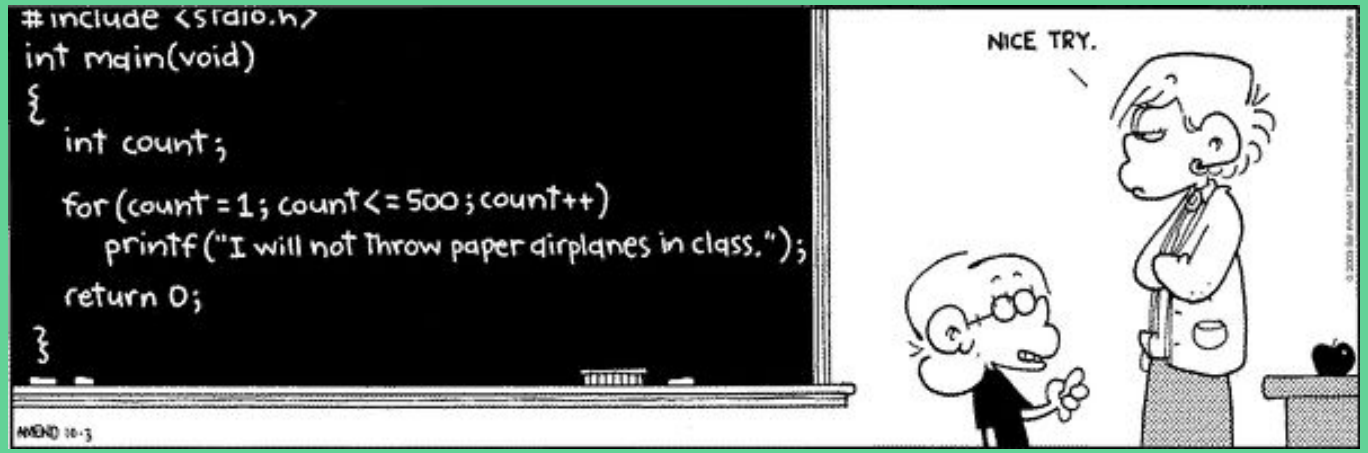
`return` to retrieve a variable outside of a function ***(what happens in the function stays in the function)***

Top #10 programming features you need to know

1. Variables
2. Data types
3. Lists
4. Dictionaries
5. Conditionals
6. Indentation
7. Functions
8. Loops
9. Files
10. Modules

A **loop** is a procedure to repeat a piece of code.
(another way of saying this is that it **iterates** through code)

- Loops enable you to re-run blocks of code for as many times as you need.
- Python has two main ways to run loops: **`for & while`**



Bill Amend, FoxTrot, October 3, 2003

efficiency benefit of `for` loops

Each of these would accomplish the same thing:

Option #1: 2+ lines of code

```
for value in a_list:
    print(value)
```

Option #2: as many lines of code as there are list entries

```
print(a_list[0])
print(a_list[1])
print(a_list[2])
...
```

`for` loop syntax

iteration variable

iter*able* variable

```
for value in a_list:

    print(value)
```

colon

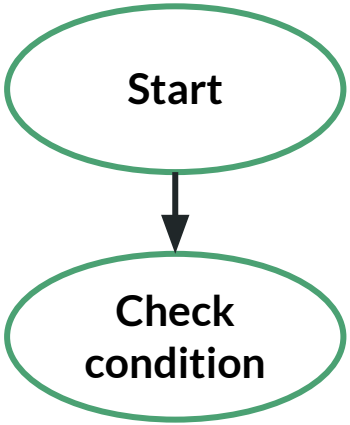
body of the loop

indented by 4 spaces (or tab)

A **for loop** is a procedure to repeat code for every element in a sequence.

`for` loop syntax

```
a_list = [1,2,3]

for value in a_list:

    print(value)
```

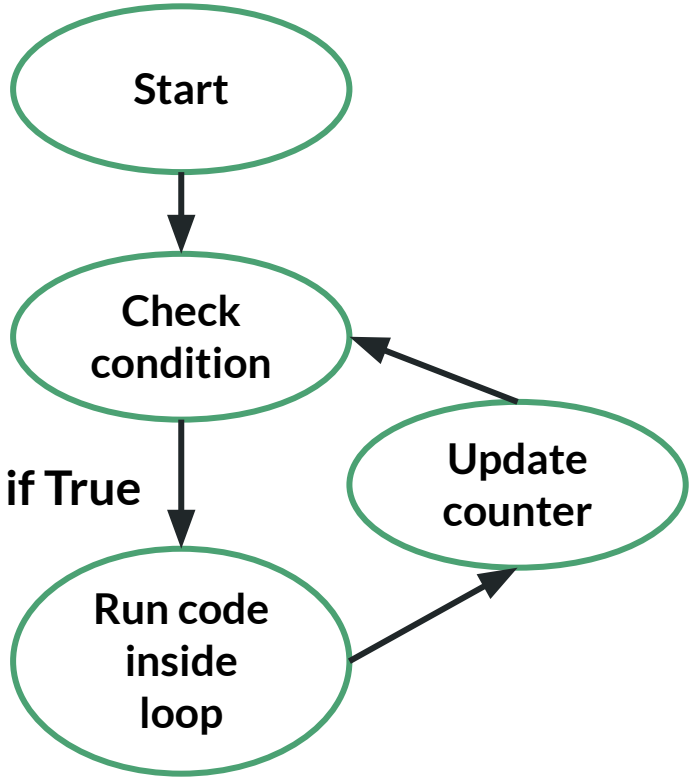


`for` loop syntax

```
a_list = [1,2,3]

for value in a_list:

    print(value)
```

```
1
```

output

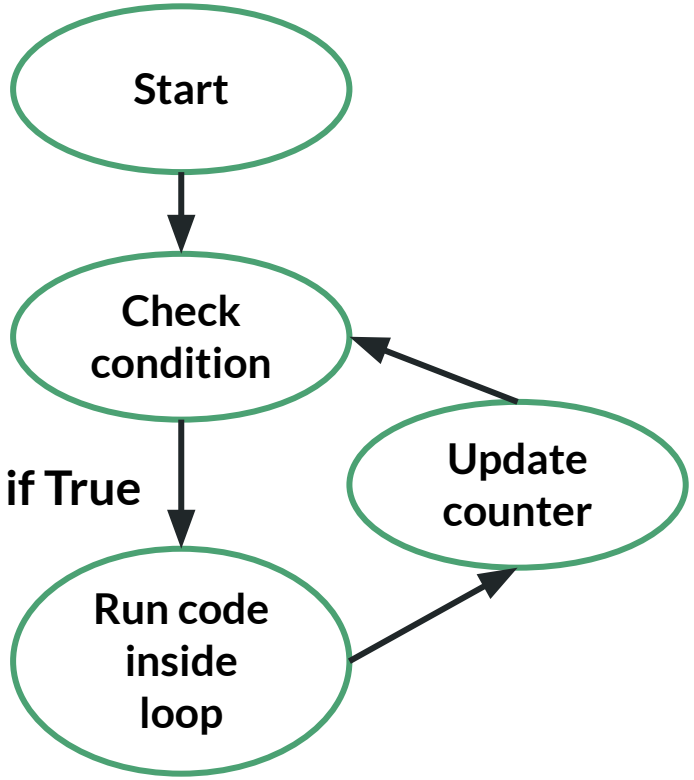


`for` loop syntax

```
a_list = [1,2,3]

for value in a_list:

    print(value)
```

```
1
2
```

output

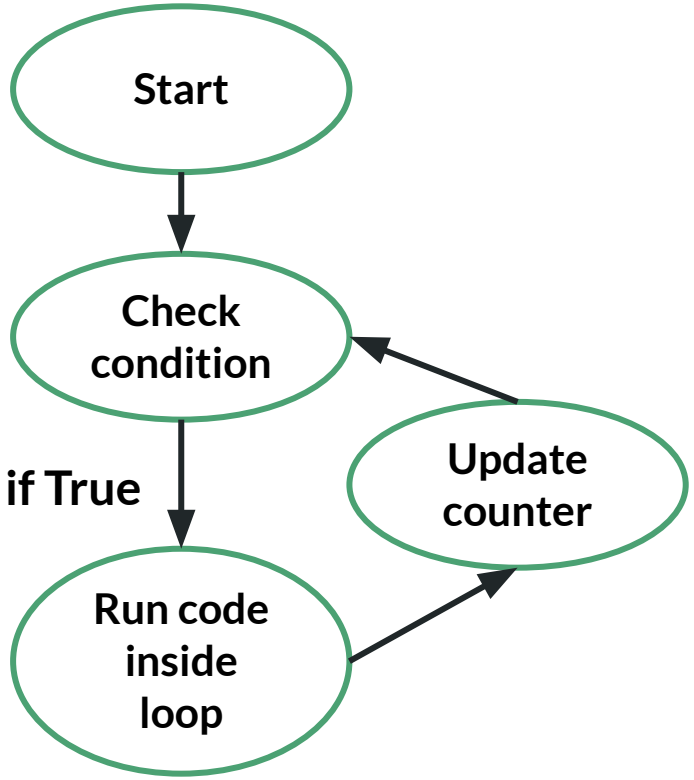


`for` loop syntax

```
a_list = [1,2,3]

for value in a_list:

    print(value)
```

```
1
2
3
```

output

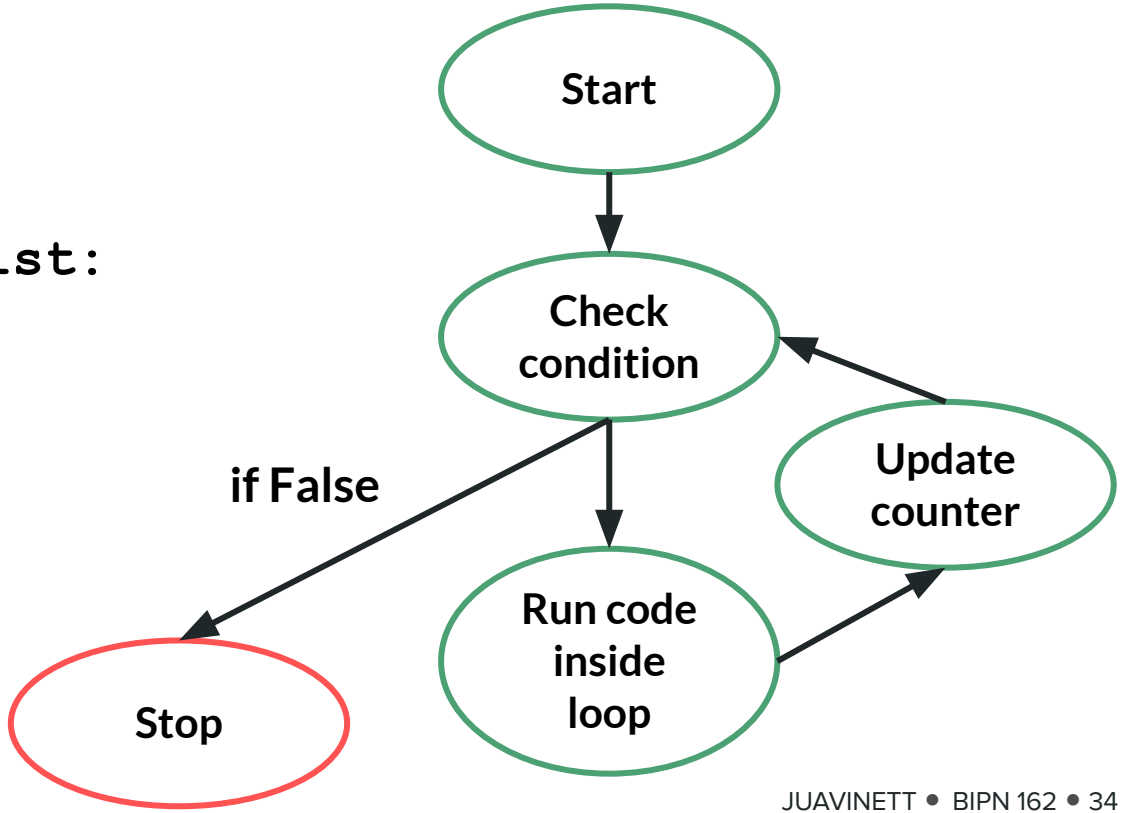


`for` loop syntax

```
a_list = [1,2,3]

for value in a_list:

    print(value)
```

```
1
2
3
```

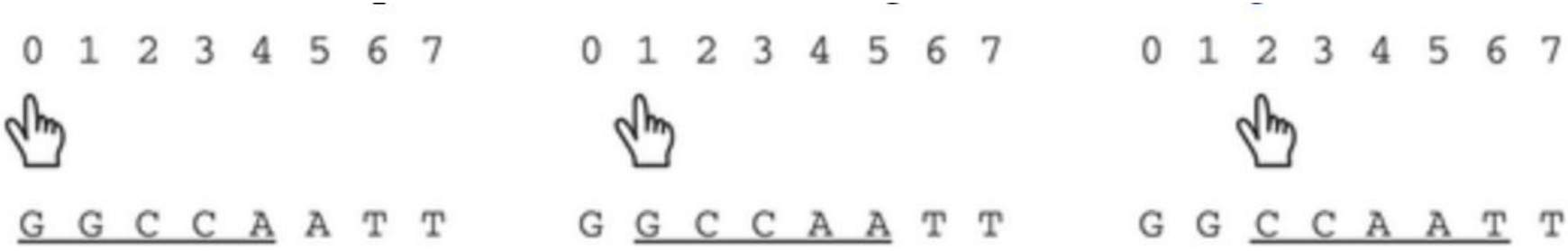
output



We can also loop over a list of indices!

Let's say we want to look for a "CAT" box, a common motif in DNA, with the sequence `CCAATT`

Since we want to look at a **slice** of DNA, rather than looping through individual items in the string, we need the indices.



`while` loop syntax

condition we're checking

colon

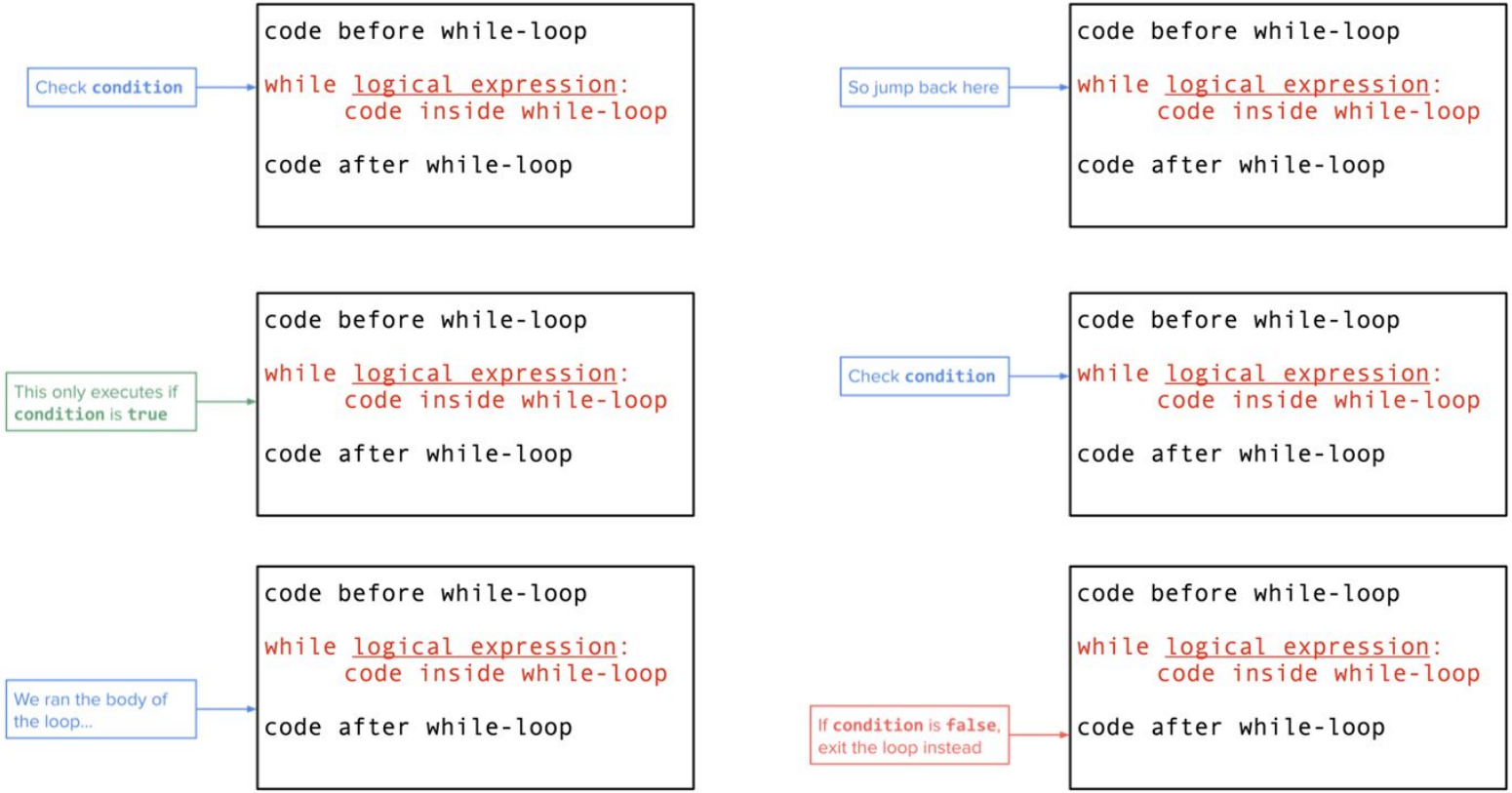
```
while condition:

    print(value)
```

body of the loop

indented by 4 spaces (or tab)

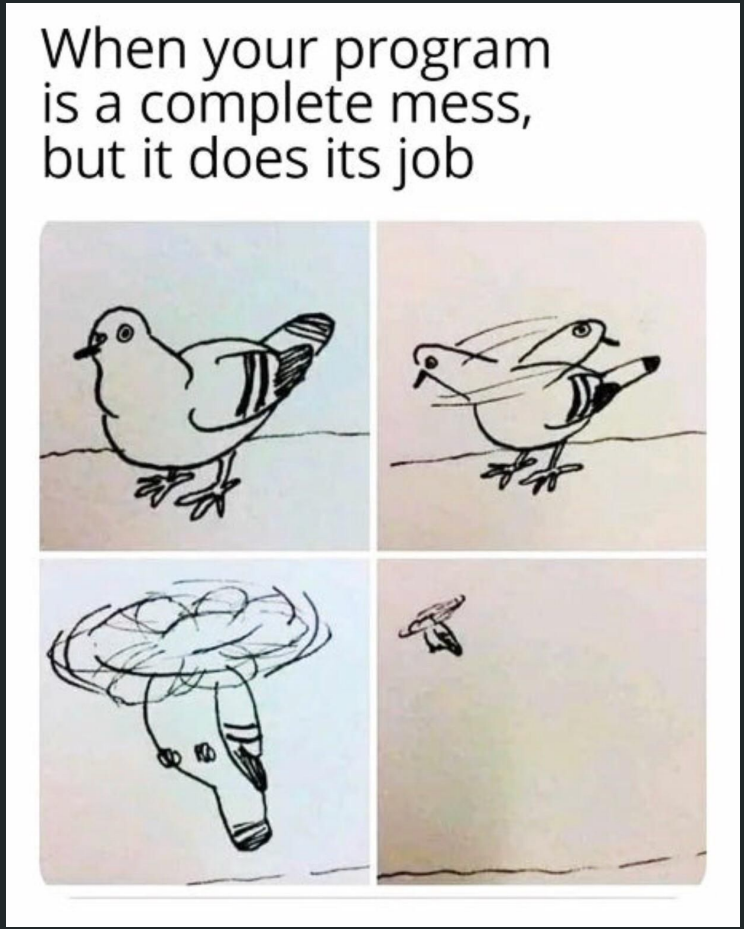
While this condition is true, the loop will run!

It will repeat until the condition is no longer True.



Order of execution in a while loop (from Stepik)

Let’s put our programming skills to the test.



Objectives for today

- Review the top 10 programming features you need to know
- Establish guidelines for using AI assistants in this course (and beyond)

Some terms

An **AI-assistant** is an artificial intelligence agent (a computer program) that responds to normal human inputs like speech and text with human-like answers (e.g., Alexa, Siri)

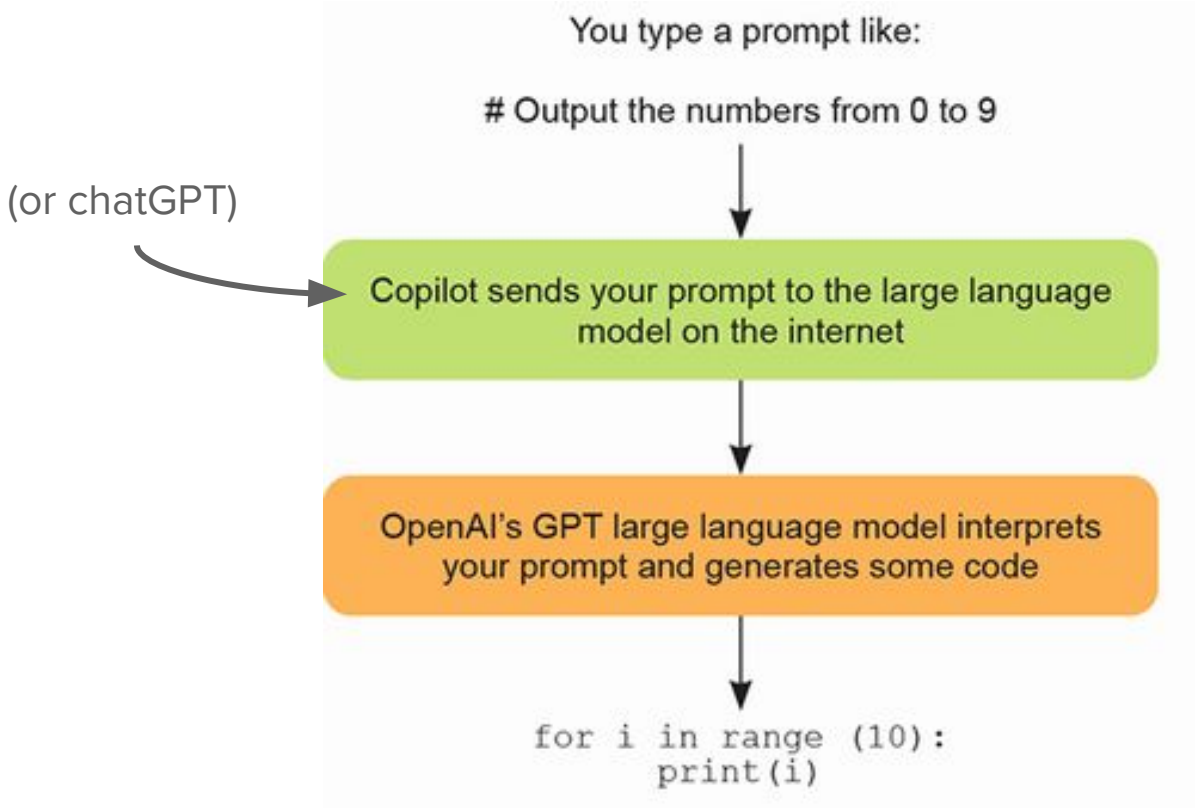
There are several code writing AI assistants, such as Google Copilot.

AI assistants are driven by **large language models (LLMs)**, which store information about relationships between words, including which words make sense in certain contexts, and use this to predict the best sequence of words to respond to a prompt.

The **prompt** is what you tell the AI assistant, describing the program you want.

Prompt engineering is the skill of writing tactful and precise prompts.

From Porter & Zingaro,
Learn AI-Assisted Python Programming



Going from *prompt* to *program*

From Porter & Zingaro,
Learn AI-Assisted Python Programming

Other things AI assistants can do for us:

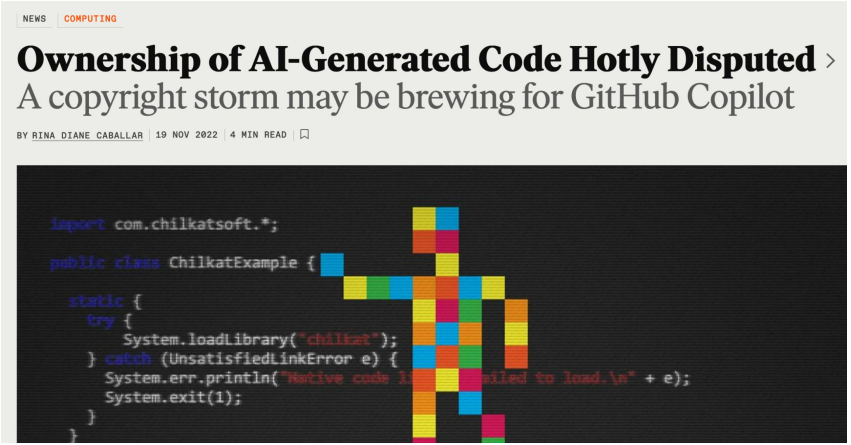
- Explain code — *“What is this line of code doing?”*
  - Note that this can be wrong…
- Make code easier — *“Simplify this program so that it is easier to read.”*
- Fixing bugs — “Why isn’t this program _____?”

Inspired by Porter & Zingaro,
Learn AI-Assisted Python Programming

However (and this is big):
AI assistants are not perfect.
They make mistakes.
As novice programmers, we need to be able to catch them.

Other issues with generative AI

- It doesn't know what it doesn't know → this leads to **fabrication**
- It is **biased** because it replicates biases in its training set (try "generate a list of names")
- There are ongoing copyright issues —
  LLMs are trained on other people's code!
- Societal concerns about jobs, the human condition, etc.



https://spectrum.ieee.org/ai-code-generation-ownership

Good, free options for AI assistants

ChatGPT

Pro: doesn’t require anything except an account

Con: not code-specific

GitHub Copilot

Pro: code-specific, built into IDE (e.g. Visual Studio Code)

Con: requires GitHub account & use of IDE

Steps to local install of Copilot + VS Code (optional)

1. Set up your GitHub account & sign up for Copilot:

1. Go to https://github.com/signup and sign up for a GitHub account.
2. Go into your settings in GitHub and enable Copilot. This is the point where you'll either need to verify you are a student or sign up for the 30-day free trial.

2. Install Python:

3. Go to www.python.org/downloads/.
4. Download and install the latest version of Python (3.11.1 at the time of writing).

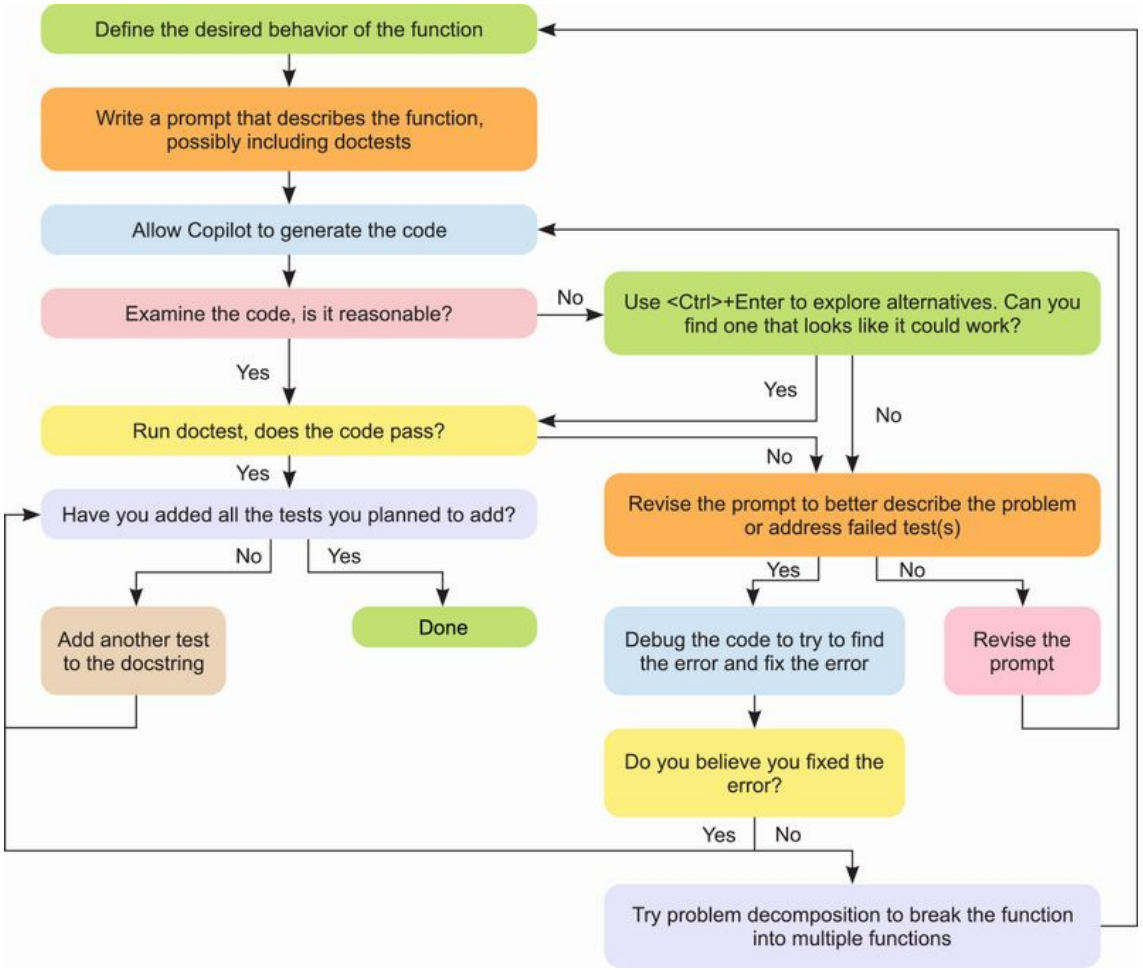
3. Install VS Code:

- Go to https://code.visualstudio.com/download, select the main download for your operating system (e.g., Windows Download or Mac Download).
- Download and install the latest version of VS Code.

4. Install the following VS Code extensions (details).

- *Python (by Microsoft)*—Follow the instructions at https://code.visualstudio.com/docs/languages/python to set up the Python extension correctly (specifically, selecting the correct interpreter).
- *GitHub Copilot (by GitHub)*

Coding workflow with an AI assistant



From Porter & Zingaro,
Learn AI-Assisted Python Programming

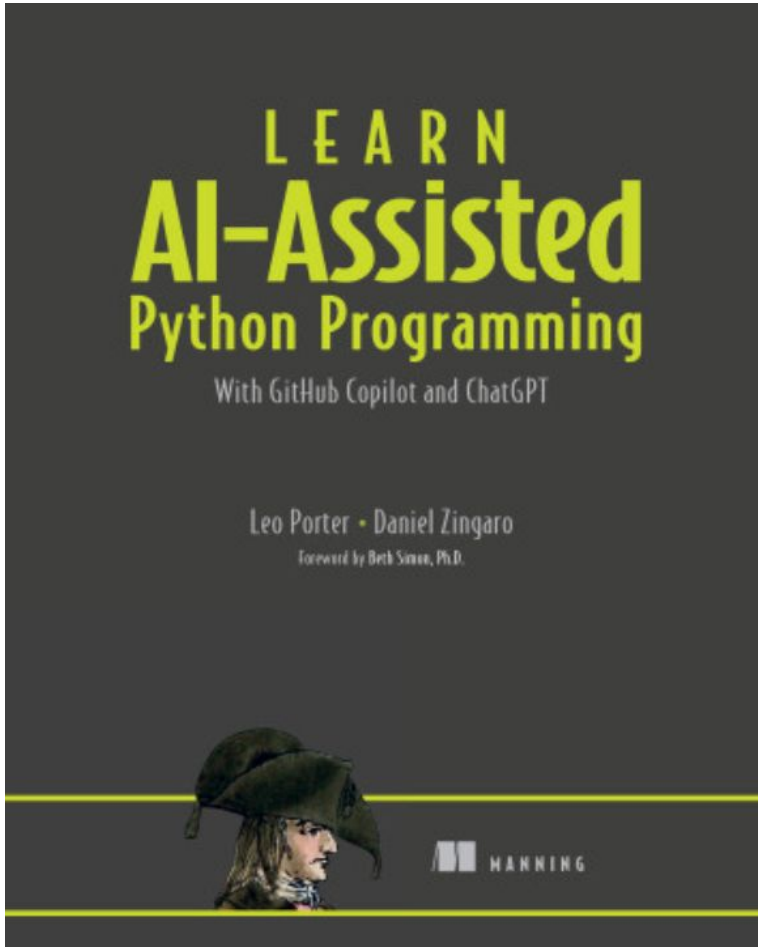
Summary

- AI assistants are powerful tools but they are not perfect.
- We still need to be able to break problems into small tasks, and we still need to understand code to some degree.
  - Writing good prompts requires a basic understanding of what computers know and what they don't.
- **Testing our code is important**.

Inspired by Porter & Zingaro,
Learn AI-Assisted Python Programming

Resources

Porter & Zingaro (2023) *Learn AI-Assisted Python Programming* — *entire book on using Copilot & VS Code!*



Resources

A List of Good Python YouTube Channels

CodeAcademy Python Syntax Cheatsheet

Software Carpentry: Python Fundamentals

Software Carpentry: Variables & Assignment

Software Carpentry: Data Types & Type Conversion

Error types in Python