Basics of Python

Brief History of Python

- Created by Guido van Rossum in 1990
- Specifically designed as an easy to use language
- High focused on readability
- Interpreted and cross platform

Why Choose Python?

- Designed for clear, logical code that is easy to read and learn
- Lots of existing libraries and frameworks written in Python
- Focuses on optimizing developer time
- Great documentation online

  docs.python.org/3

What can you do with Python?

- **Automate simple task**
  - Searching for files, read and modify them
  - Scraping information from websites
  - Reading and editing excel files
  - Work with PDFs
  - Automate emails and text messages
  - Connect to database systems

What can you do with Python?

- **Data Science and Machine Learning**
  - Analyze large data files (numbpy, pandas)
  - Create visualizations (seaborn, plotly)
  - Perform machine learning tasks (sci-kit-learn, tensorflow)
  - Create and run predictive algorithms

What can you do with Python?

- **Create websites**
  - Use web frameworks such as Django and Flask to handle backend of a website and user data
  - Create interactive dashboards for users such as plotly and dash

What can you do with Python?

- **Software development**
  - Building desktop applications, automation scripts, and support tools for software development.

Python

- **Create websites**
  - Use web frameworks such as Django and Flask to handle backend of a website and user data
  - Create interactive dashboards for users such as plotly and dash

Installing Python

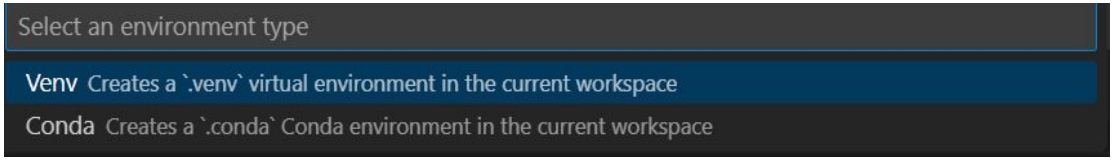
- **Tools**
  - Python 3 python.org/downloads
  - IDE (VS Code)
  - VS Code Python Extension

Create Virtual Environmen

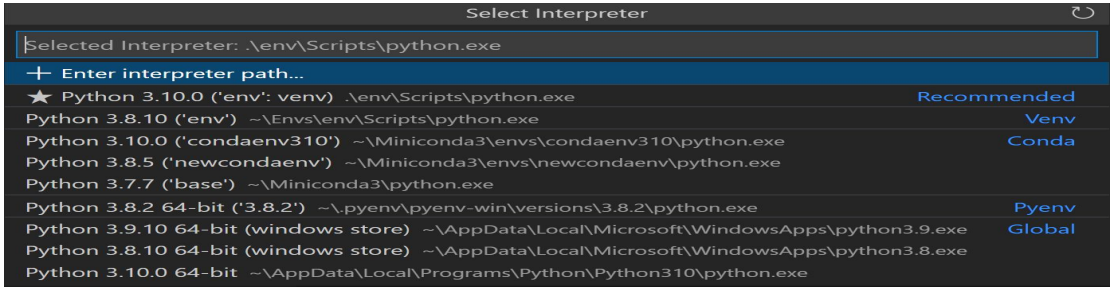
- **Create a virtual environment**
  - Open the Command Palette (Ctrl+Shift+P) or From the **View** menu, click **Command Pallete**
  - Start typing the **Python: Create Environment** command to search, and then select the command.

Create Virtual Environme

- **Create a virtual environment**
  - Select .venv



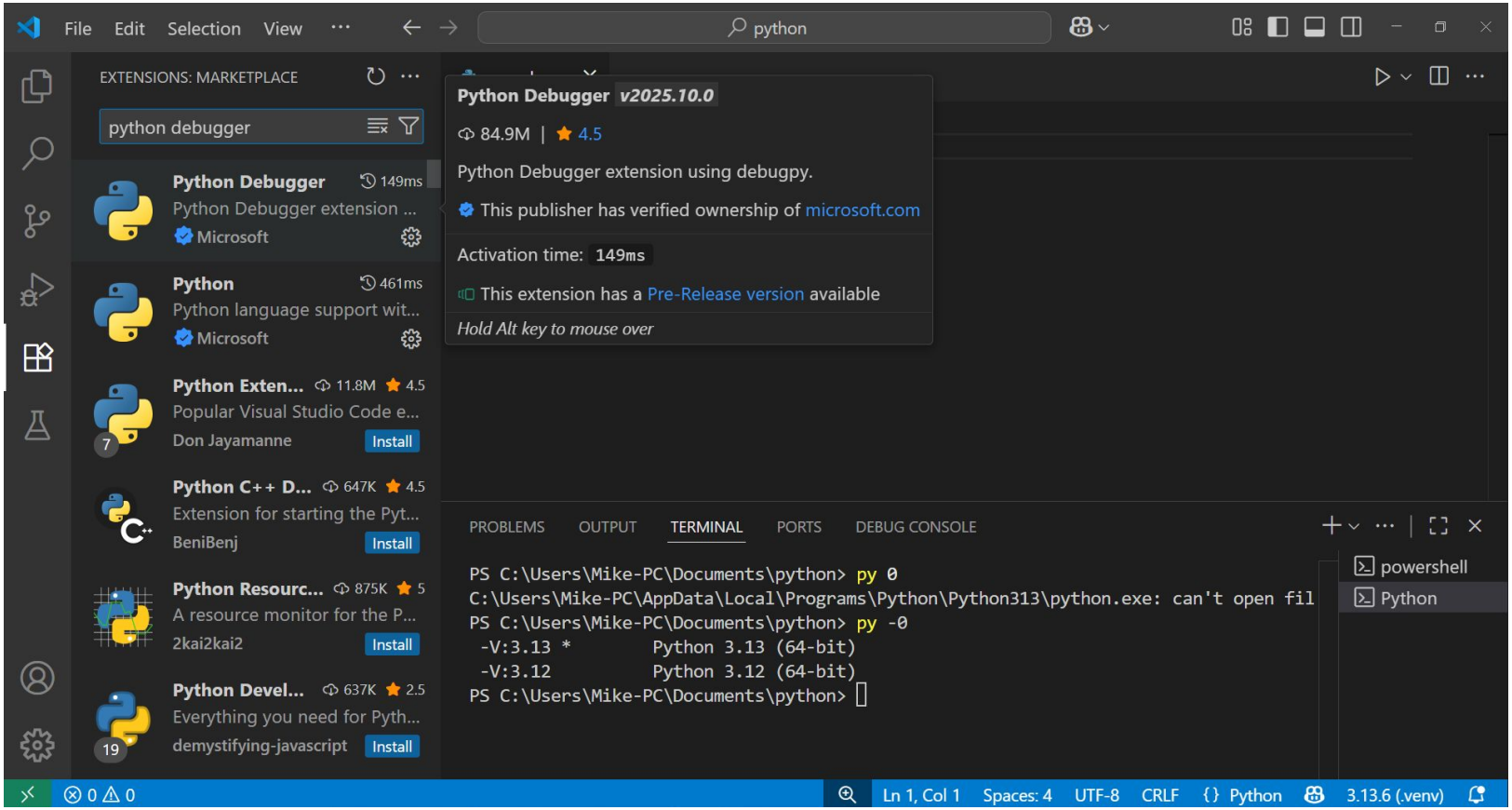
  - Select the Python interpreter you installed



Configure and Run the Debugger

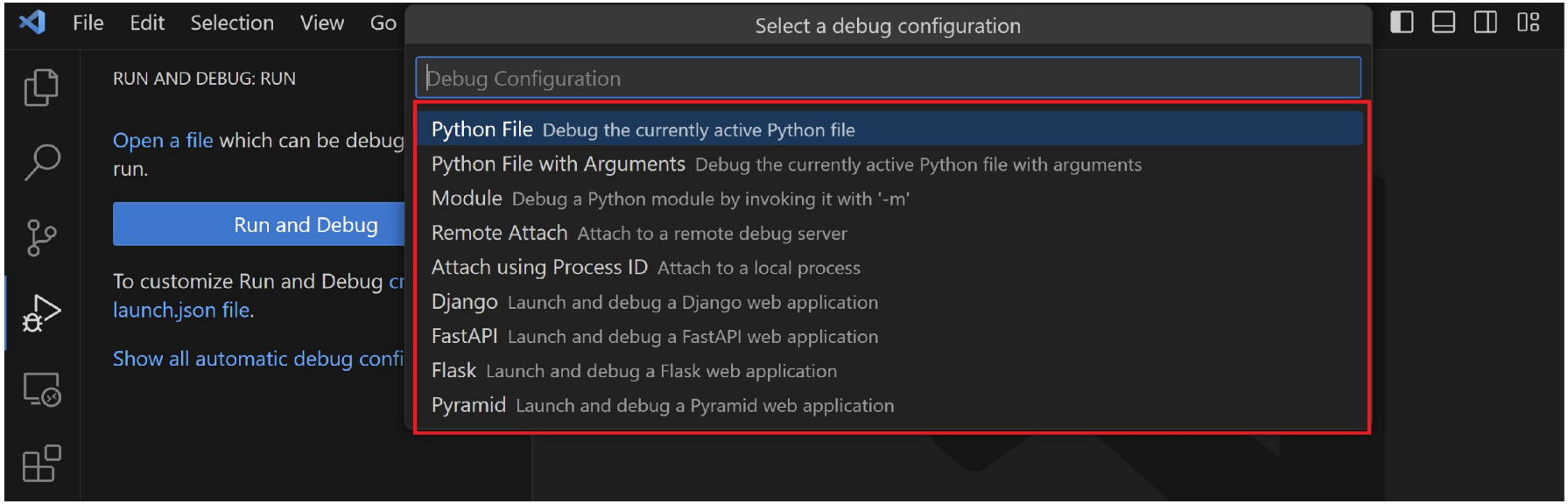
- **View>Extensions (Ctrl+Shift+X)**
  - Search for the installed **Python debugger extension** or if not install the extension by typing **Python debugger** on the search bar

Configure and Run the Debugger

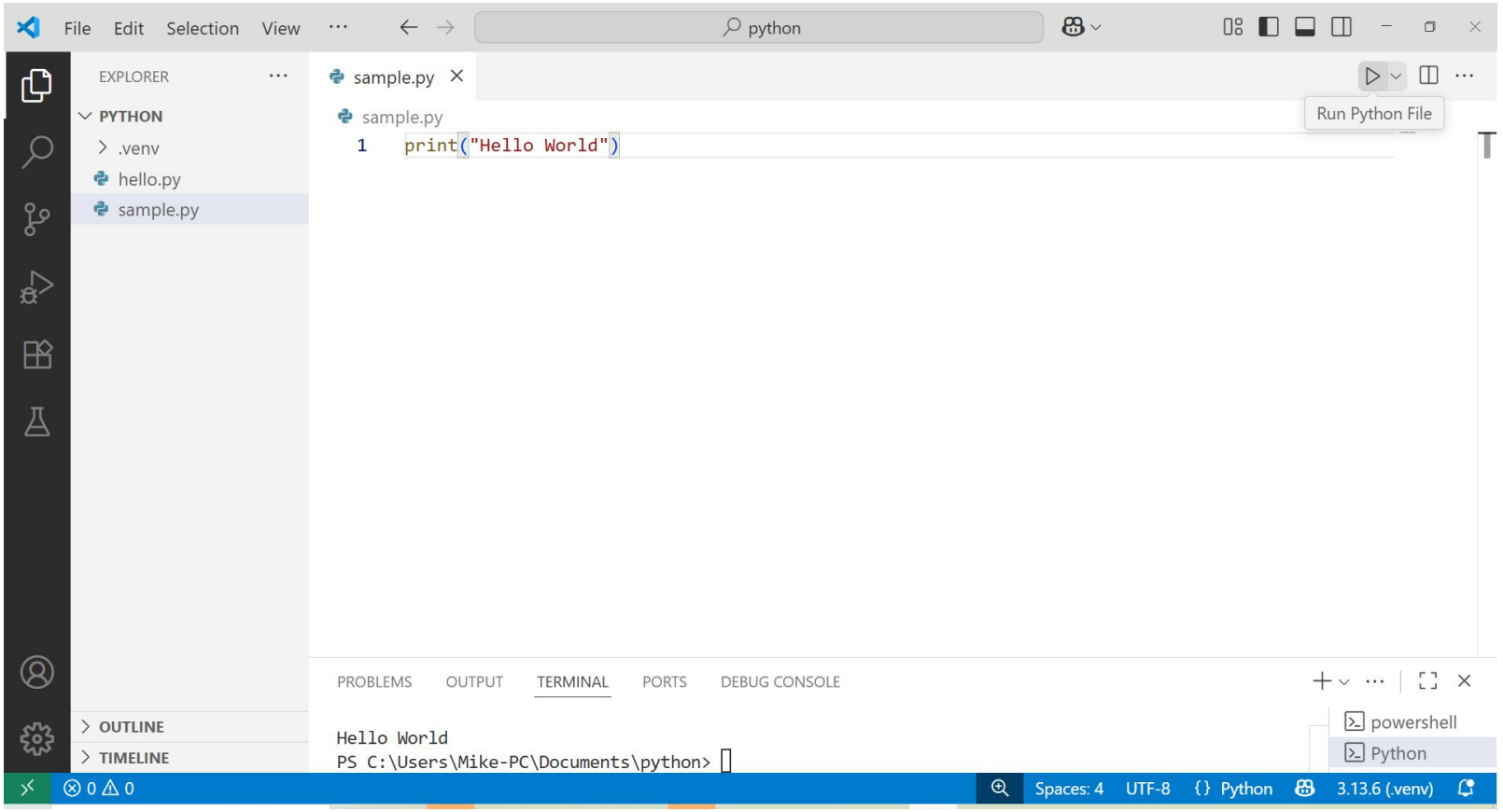


Configure and Run the Debugger

Since this is your first time debugging this file, a configuration menu will open from the Command Palette allowing you to select the type of debug configuration you would like for the opened file.



Create Python source file



Syntax

- Python uses **new lines** to complete a command
- Python relies on **indentation**, using whitespace, to define scope; such as the scope of loops, functions and classes

```
print("Hello, World!")
```

Creating a Comment

```
#This is a comment
print("Hello, World!")
```

```
#This is a comment
#written in
#more than just one
line
print("Hello, World!")
```

Variables

- Rules for Python variables
  - Must start with a letter or the underscore character
  - Names cannot start with a number
  - It can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
  - There can be no spaces in the name, use _ instead
  - Names are case-sensitive
  - Names cannot be a Python reserved word

User Input

- **Syntax**

```
msg=input("Enter your name: ")
print(f"Your name is: {msg}")

print("Enter your name:")
msg=input()
print(f"Your name is: {msg} ")
```

Assigning Values

Multiple Values to Multiple Variables

```
x, y, z = "Mike", "Esteron", "Acosta"
print(x)
print(y)
print(z)
```

One Value to Many Variables

```
x = y = z = "Mike"
print(x)
print(y)
print(z)
```

Unpacking a Collection

```
color = ["blue", "red", "yellow"]
x, y, z = color
print(x)
print(y)
print(z)
```

Variables

```
x = "I love Python"
print(x)
```

```
x = "Python "
y = "is "
z = "awesome "
print(x, y, z)
```

```
x = "Python "
y = "is "
z = "awesome"
print(x + y + z)
```

```
x = 5

y = 10

print(x + y)
```

Data Types

Name	Type	Description
integers	int	whole numbers: 3 200 300
floating point	float	numbers with decimal point: 2.3 4.6 100.0
complex	complex	represents complex numbers, which have both a real and an imaginary part, j represents the imaginary unit (the square root of -1).: 3 + 2j
string	str	Ordered sequence of characters: “hello” ‘Sammy’ “2000”
boolean	bool	logical value indicating True or False
lists	list	ordered sequence of objects: [10, “hello”, 200.3]
tuples	tup	ordered immutable sequence of objects: (10, “hello”, 200.3)
dictionary	dict	Unordered key:value pairs: {“mykey”: “value”, “name”: “frankie”}
sets	set	Unordered collection of unique objects: {“a”, “b”}

Strings

- They are sequences of characters by either single quotation marks, or double quotation marks.
  - ‘hello’
  - “Hello”
  - “100.50”

Strings

- **Because strings are ordered sequence, we can slice and index to grab subsections of the string**
- Indexing notation uses [ ] symbol

Strings

- **Because strings are ordered sequence, we can slice and index to grab subsections of the string**
- Indexing notation uses [ ] symbol

```
Character :  h  e  l  l  o
Index     :  0  1  2  3  4
Reversed  :  0 -4 -3 -2 -1
```

Looping through Strings, len(), in() not in()

```
for x in "banana":
  print(x)

a = "Hello, World!"
print(len(a))

txt = "The best things in life are free!"
print("free" in txt)

txt = "The best things in life are free!"
print("expensive" not in txt)
```

Slicing Strings

- **Slicing allows you to grab a sub-section of mutiple characters, a "slice"of the string**
- Syntax
  - [start:stop:step]

start is the numerical index for the slice start

stop is the index you will go up to (but dont include)

step is the size of the "jump" you take

Slicing Strings

```
b = "Hello, World!"
print(b[2:5])
```

```
b = "Hello, World!"
print(b[:5])
```

```
b = "Hello, World!"
print(b[-5:-2])
```

```
myString="abcdefghijk
"
print(myString[2:7:2])

print(myString[::3])

print(myString[::-1])
```

Modify Strings

String Function	Example
upper()	a = "Hello, World!" print(a.upper())
lower()	a = "Hello, World!" print(a.lower())
strip()	a = " Hello, World! " print(a.strip()) # returns "Hello, World!"
replace()	a = "Hello, World!" print(a.replace("H", "J"))
split()	a = "Hello, World!" print(a.split(",")) # returns ['Hello', ' World!']

Concatenation

```
a = "Hello"
b = "World"
c = a + "   " + b
print(c)
```

f Strings

- **Syntax:**
  f"some text {placeholder}"

```
price = 59000
txt = f"The price is {price:,} dollars"
print(txt)
```

```
age = 36
txt = f"My name is John, I am {age}"
print(txt)
```

A **placeholder** can include a modifier to format the value.

```
price = 59
txt = f"The price is {price:.2f} dollars"
print(txt)
```

Escape sequence

Code	Result
\’	Single quote
\\	Backslash
\n	New line
\r	Carriage return
\t	Tab

ERROR

```
txt = "We are the so-called "Vikings" from
the north."
```

Arithmetic Operators

Operator	Name	Example x=5; y=3	Result
+	Addition	x+y	8
-	Subtraction	x-y	2
*	Multiplication	x*y	15
/	Division	x/y	1.666...7
%	Modulo	x%y	2
**	Exponentiation	x**y	125
//	Floor division	x//y	1

Comparison Operators

Operator	Name	Example x=5; y=3
==	equal	x==y
!=	not equal	x!=y
>	greater than	x>y
<	less than	x<y
>=	greater than or equal	x>=y
<=	less than or equal	x<=y

Assignment Operators

Operator	Sames as …	Description
x = y	x = y	The left operand gets set to the value of the expression on the right
x += y	x = x + y	Addition
x -= y	x = x - y	Subtraction
x *= y	x = x * y	Multiplication
x /= y	x = x / y	Division
x %= y	x = x % y	Modulus

Logical Operators

Operator	Description	Example
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4

Identity Operators

- Used to compare objects if they are actually the same object with the same memory location

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Identity Operators

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x

print(x is z)

# returns True because z is the same object as x

print(x is y)

# returns False because x is not the same object as y,
even if they have the same content
```

Membership Operators

- Used to test if a sequence is presented in an object:

Operator	Description	Example	
in	Returns True if a sequence with the specified value is present in the object	x in y	x = ["apple", "banana"] print("banana" in x)
not in	Returns True if a sequence with the specified value is not present in the object	x not in y	x = ["apple", "banana"] print("pineapple" not in x)

Conditional Statements

if statement - executes some code only if a specified condition is true

if...else statement - executes some code if a condition is true and another code if the condition is false

if...elif....else statement - specifies a new condition to test, if the first condition is false

match statement - selects one of many blocks of code to be executed

if Statements

- **Syntax**

  if expression:

  statementN

```
day1=100
day2=7
if (day1 - day2) < 100:
    print(f"Its {day1-day2} days left before Christmas ")
```

if else Statements

- **Syntax**

```
if condition:
    statmentN
else:
    statementN
```

```
mood="Sad"
if mood="Happy":
    print("Yehey! Masaya ako nasa mood ako")
else:
    print("Di masaya kaya malungkot")
```

if elif Statements

- **Syntax**

```
if expression:
    statementN
elif expression:
    statementN
else:
    statementN
```

```
score = 85
if score >= 90:
    print("Grade A")
elif score >= 80:
    print("Grade B")
elif score >= 70:
    print("Grade C")
else:
    print("Grade D")
```

match Statements

- **Syntax**

match expression:

case x:

statmentN

case y:

statementN

case z:

statementN

case _:

```
day = 4
match day:
  case 6:
    print("Today is Saturday")
  case 7:
    print("Today is Sunday")
  case _:
    print("Looking forward to the Weekend")
```

while loop

- **Syntax**

  while expression:

  statementN

```
i = 1
while i < 6:
  print(i)
  i += 1
```

break statement

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

continue statement

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

range()

- loop through a set of code a specified number of times
- returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

```
for x in range(6):
  print(x)
```

```
for x in range(2, 6):
  print(x)
```

range()

The `range()` function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter

```
for x in range(2, 30, 3):
  print(x)
```

range(): Exercises

Write a Python program using the following patterns:

```
1  2  3  4  5
6  7  8  9  10
```

```
*
* *
* * *
* * * *
* * * * *
```

```
* 0 * 0 *
* 0 * 0 *
* 0 * 0 *
```

for loop

- **Syntax**

  for var in *object*:

  statementN

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

nested loop

```
for x in range(1,5):
    for y in range(6,10):
        print(f"{x}{y} ", end="")
    print("")
```

Python Data Structures

List [ ]

- Ordered sequences that can hold a variety of object types.
- It uses [ ] brackets and commas to separate objects in the list.
- Lists can be nested

List [ ]

- List Operations
  - Index
  - Slicing
  - Adding Items
  - Changing Items
  - Deleting Items
  - Loop Lists

List Indexing

```
mylist=["honda", "toyota", "nissan", "mitsubishi"]

print(mylist[2])

print(mylist[-1])

print(mylist[1:3])

print(mylist[-3:-1])

print(mylist[:3:2])

print(mylist[::2])

print(mylist[::-1])
```

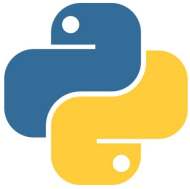
Insert Items

```
mylist=["honda", "toyota", "nissan", "mitsubishi"]

print(mylist.append("hyundai"))

print(mylist.insert(1,"byd"))
```

Changing Items



```
mylist=["honda", "toyota", "nissan", "mitsubishi"]

mylist[3]="byd"
```

Deleting Items

```
mylist=["honda", "toyota", "nissan", "mitsubishi"]

print(mylist.remove("nissan"))

print(mylist.pop(1))

print(mylist.pop())
```

Sort Items

```
mylist=["honda", "toyota", "nissan", "mitsubishi"]

print(mylist.sort())
```

Looping Items

```
mylist=["honda", "toyota", "nissan", "mitsubishi"]

for x in mylist:
  print(x)

for x in range(len(mylist)):
  print(mylist[x])
```

Nested Loop Items

```
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for sublist in nested_list:

    for item in sublist:
        print(item, end=" ")

    print("")
```

Tuples ( )

- They are very similar to lists, however, tuples are IMMUTABLE.

Accesing Tuples

```
mytuple=("honda", "toyota", "nissan", "mitsubishi")

print(mytuple[2])

print(mytuple[-1])

print(mytuple[1:3])

print(mytuple[-3:-1])

print(mytuple[1:])

print(mytuple[:3:2])

print(mytuple[::2])

print(mytuple[::-1])
```

Insert, Changing, Deleting Tuple Values

```
#adding values
mytuple=("honda", "toyota", "nissan", "mitsubishi")

mytuple2=list(mytuple)

mytuple2.append("byd")

mytuple=tuple(mytuple2)

#changing values
print(mytuple)
mytuple=("honda", "toyota", "nissan", "mitsubishi")

mytuple2=list(mytuple)

mytuple2[1]="byd"

mytuple=tuple(mytuple2)

print(mytuple)
```

Tuple index(), count() methods

```
mytuple=("honda", "toyota", "nissan", "mitsubishi", "nissan")

print(mytuple.index("nissan"))

print(mytuple.count())
```

Dictionaries { }

- Unordered mappins for storing objects.
- It uses key-value pairing
- This key value-pir allows user to quickly grab objects without needing to know an index location.

```
tala={
    "id": 1111,
   "name": "Mike Acosta",
   "location": "Urdaneta"
   }
```

Accesing Dictionaries

```
tala={
    "id": 1111,
   "name": "Mike Acosta",
   "location": "Urdaneta"
   }

   print(tala[name]);
```

Inserting and Changing Items

```
tala={
    "id": 1111,
   "name": "Mike Acosta",
   "location": "Urdaneta"
   }
   #adding items
   tala["salary"]=5000;
   print(tala.keys());
   #changing items
   tala["salary"]=15000;
   print(tala);
```

Removing Items

```
tala={
    "id": 1111,
   "name": "Mike Acosta",
   "location": "Urdaneta"
   }

   print(tala.pop("name"));
   #delete last item
   print(tala.popitem());
```

Looping Items in Dict

```
tala={
    "id": 1111,

   "name": "Mike Acosta",
   "location": "Urdaneta"
   }
```

```
#display all keys
for x in tala:
  print(x)
```

```
#display all values
for x in tala:
  print(tala[x])
```

```
#display all keys value pairs
for x,y in tala.items:
  print(x,y)
```

Nested Loop in Dictionarie

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
```

```
for x, obj in myfamily.items():
    print(x)

    for y in obj:
        print(y + ':', obj[y])
```