Types

MAD 102

Getting Input

- Information that is required for your program to operate comes in the form of input
- Capture input using the **input()** function
- The input function captures information from the user as text
  - This text is known as a **string**

Getting input

- The input string can be assigned to a variable and used in your algorithm
- The input method takes an **argument** that represents the **prompt** text
  - Text that will be displayed to give the user an idea about what you are requesting

```
name = input("Enter your name")
```

Displayed in the console → Enter your name

Getting input

- The input always returns a string value
    - If we are creating a calculator application, and we ask the user for two numbers, they are returned as ‘2’ and ‘3’ not 2 and 3
        - They are string types, not numerical types
    - We need to convert the values from one **type** to another

Converting input types (Explicit)

- Example of explicit conversion

```
age = input("Enter your Age")
print(type(age))
```

Data Type: <class ‘string’>

- int() to perform explicit type conversion “age” of to integer type.

```
age = int(input("Enter your Age"))
print(type(age))
```

Data Type: <class ‘int’>

Converting input types

- Implicit conversion

```
num1 = 1

num2 = 3.4

sum = num1+num2
```

Data Type: <class ‘int’>

Outputting Information

- Displaying the results to the screen is a common programming task
- Python has a print() function that will display the results in the console
    - The print function takes a string literal – text enclosed in quotes or it can take a variable name to display
    -

Print function

- By default, the print statement ends with a new line character
  - Multiple print statements in a row will display the results on a separate line
- White space matters!
  - Each space will be displayed on the screen

Print function

- The print function can take one or more arguments, separated by a comma.
  - This means that each string value will be displayed with a space in between
  - This can be a mix of literals and variables

Print function

- A single string literal can be placed on different lines using the \n (newline character)

```
print("Hello \nWorld!")
```

- The final parameter can be end=" " which will force the next print function to display on the same line as the previous

```
print("Hello", end='')
print(" World!")
```

Strings

- **Strings** are a sequence of characters
- A **string literal** is a string value that you create surrounded by quotes
  - Can be double or single quotes
- Strings are a **sequence** type
  - An ordered collection of objects from left to right

Strings

- Each character in a string, can be identified by its position in the sequence
  - The position is called the character’s **index**
  - The first character is located at an index of 0
  - The second character is located at an index of 1
  - …

Strings

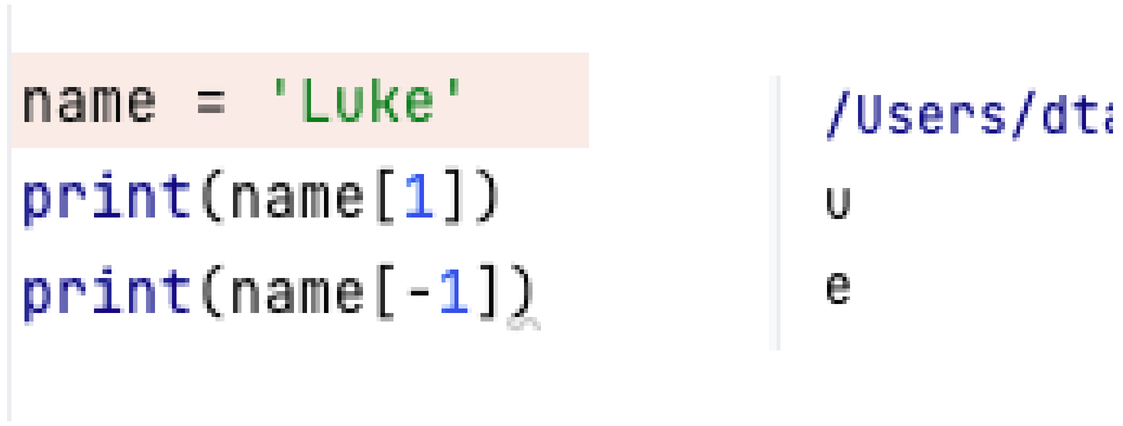
- Individual characters can be accessed using their index (position)
- Use [ ] and the index position to access

```
name = 'Luke'
print(name[1])
```

```
u
```

Strings

- Positive index values start at the left-hand side
- Negative index values start at the right-hand side



```
name = 'Luke'
print(name[1])
print(name[-1])
```

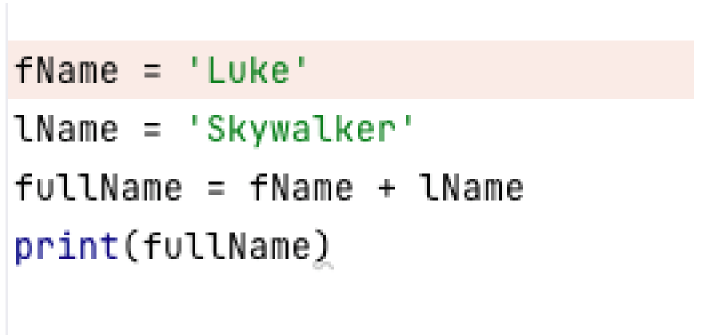
```
/Users/dt
u
e
```

Strings

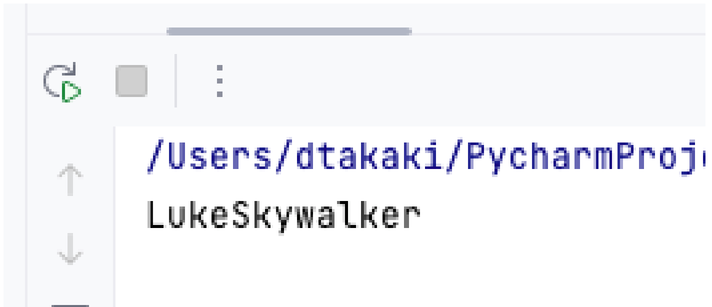
- Strings are **immutable** – you cannot change any values once they are created
  - Need to use an assignment statement when updating an **entire** string variable

Strings

- New characters can be added to the end of a string
  - This process is known as **concatenation**
  - This is supported by most sequence types
  - The concatenation operator is the + sign
  - **NOTE**: Concatenation directly joins two strings together – it does not provide any spaces – you will need to provide this



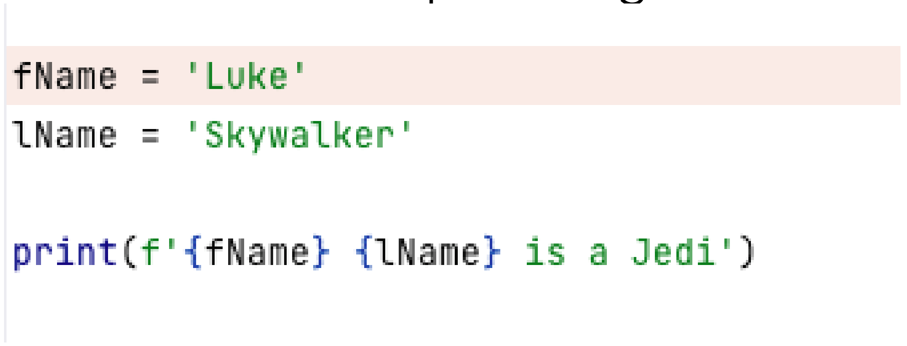
```
fName = 'Luke'
lName = 'Skywalker'
fullName = fName + lName
print(fullName)
```



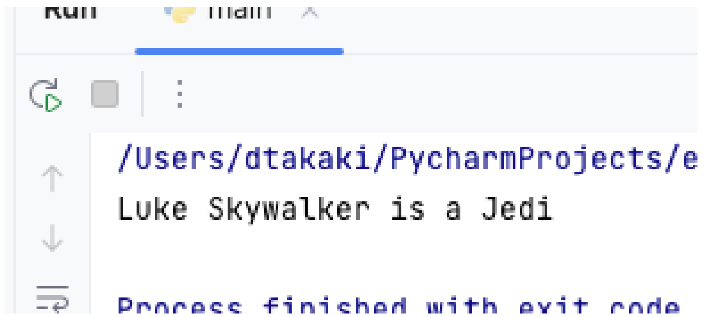
```
/Users/dtakaki/PycharmProj
LukeSkywalker
```

Formatted String

- A formatted string literal (**f-string**) uses placeholder expressions
- Begins with the **f** character – followed by curly braces { }
- The placeholder expressions are wrapped by the curly braces
- Placeholder expressions (**replacement fields**) are values that will be evaluated when the program runs
    - This involves providing variable names, etc.



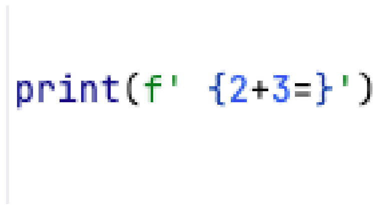
```python
fName = 'Luke'
lName = 'Skywalker'

print(f'{fName} {lName} is a Jedi')
```



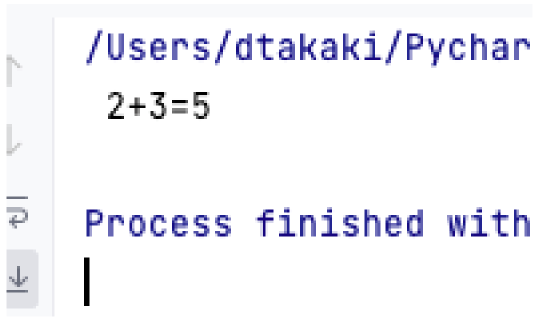
```
/Users/dtakaki/PycharmProjects/e
Luke Skywalker is a Jedi

Process finished with exit code
```

Formatted Strings

- f-string features include the ability to use the = sign
- This means, display the expression and the results
-



```python
print(f' {2+3=}')
```



```
/Users/dtakaki/Pychar
 2+3=5

Process finished with
```

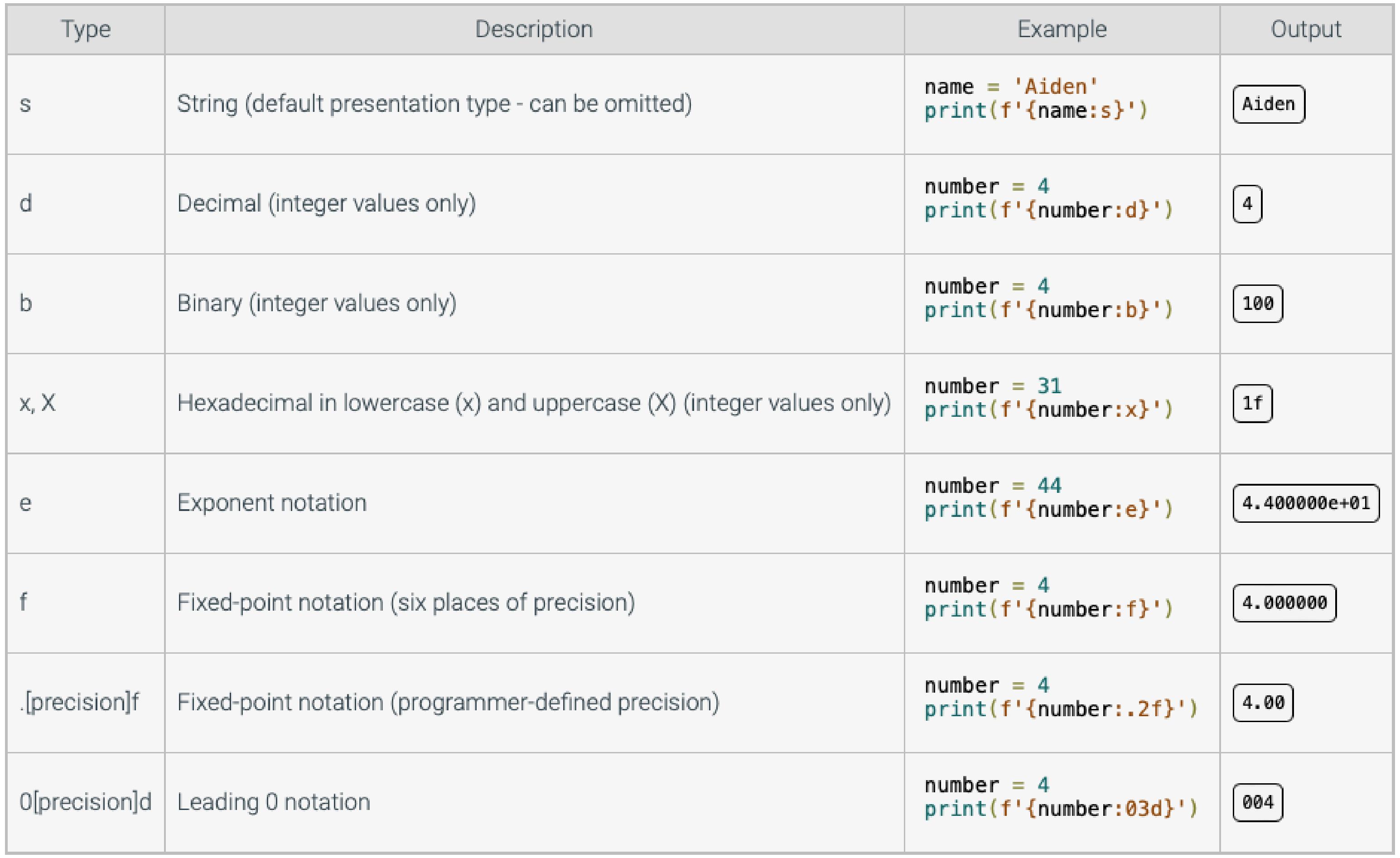
Formatted Strings

- A **format specification** can be provided in a replacement field
  - This allows customization of the formatting
  - It is introduced with a :
  - Separates what you are formatting (left) with how you are formatting it (right)

Formatted Strings

- The presentation type (right side) determines how to represent the value **in text form**

Formatted Strings



Type	Description	Example	Output

s	String (default presentation type - can be omitted)	`name = 'Aiden'`
		`print(f'{name:s}')`
			`Aiden`

d	Decimal (integer values only)	`number = 4`
		`print(f'{number:d}')`
			`4`

b	Binary (integer values only)	`number = 4`
		`print(f'{number:b}')`
			`100`

x, X	Hexadecimal in lowercase (x) and uppercase (X) (integer values only)	`number = 31`
		`print(f'{number:x}')`
			`1f`

e	Exponent notation	`number = 44`
		`print(f'{number:e}')`
			`4.400000e+01`

f	Fixed-point notation (six places of precision)	`number = 4`
		`print(f'{number:f}')`
			`4.000000`

.[precision]f	Fixed-point notation (programmer-defined precision)	`number = 4`
		`print(f'{number:.2f}')`
			`4.00`

0[precision]d	Leading 0 notation	`number = 4`
		`print(f'{number:03d}')`
			`004`

Strings

- A string that does not have any characters is known as an **empty string**
  - **NOTE:** there are no spaces in between the opening and closing otherwise se, it is no longer an empty string – it is a space

```
name = ''
```

Strings

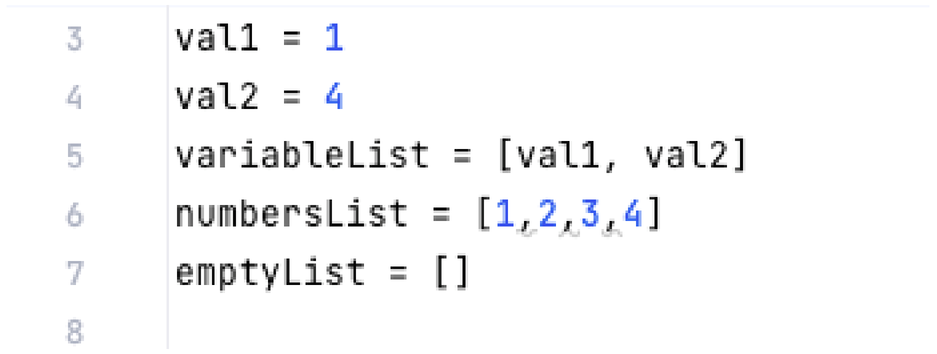
- A common operation is to determine the number of characters there are in a string
  - The **len()** function is used to **return** the number of characters (the length) of any
  - This function works on **any sequence type**

List Type

- A **container** is used to group *related* values together
- A list is a **sequence** of variables or literals surround by square brackets [ ]
    - Each of the variables or literals is called an **element**
    - Since it is a **sequence**, it is **ordered** and elements can be accessed by their **index**

List Types

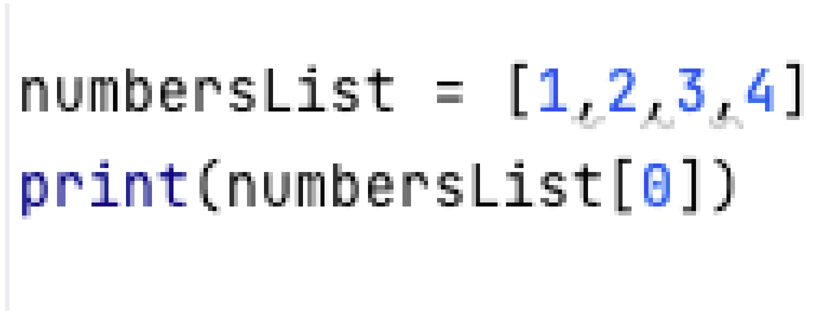
- A list can be created by supplying variables
- Or literals
- Or nothing – which creates an empty list



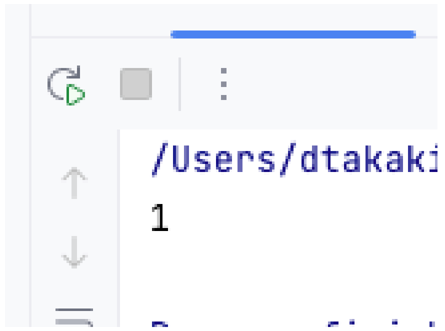
```
val1 = 1
val2 = 4
variableList = [val1, val2]
numbersList = [1,2,3,4]
emptyList = []
```

Accessing List Items

- Use the index value – an integer representing the position
  - Remember – the first position is 0



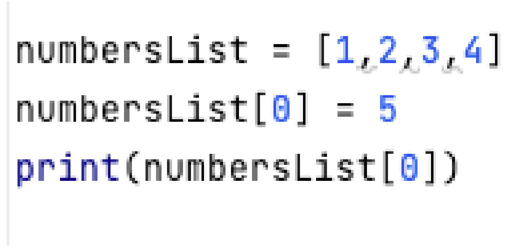
```
numbersList = [1,2,3,4]
print(numbersList[0])
```



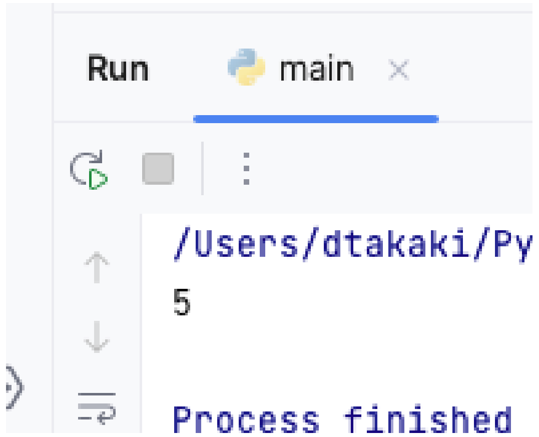
```
/Users/dtakaki
1
```

Lists

- Lists are mutable – they can be changed
- Update items in the list by assigning a new value to a position in the list



```
numbersList = [1,2,3,4]
numbersList[0] = 5
print(numbersList[0])
```

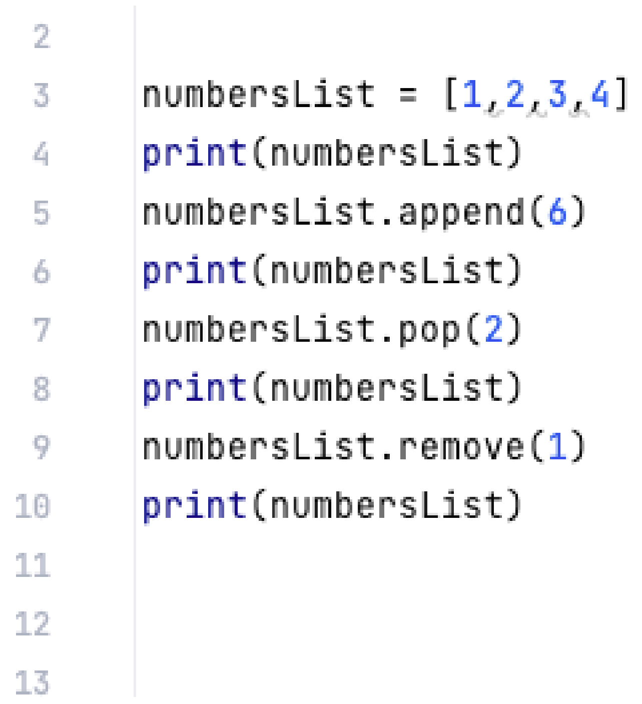


```
/Users/dtakaki/Py
5

Process finished
```

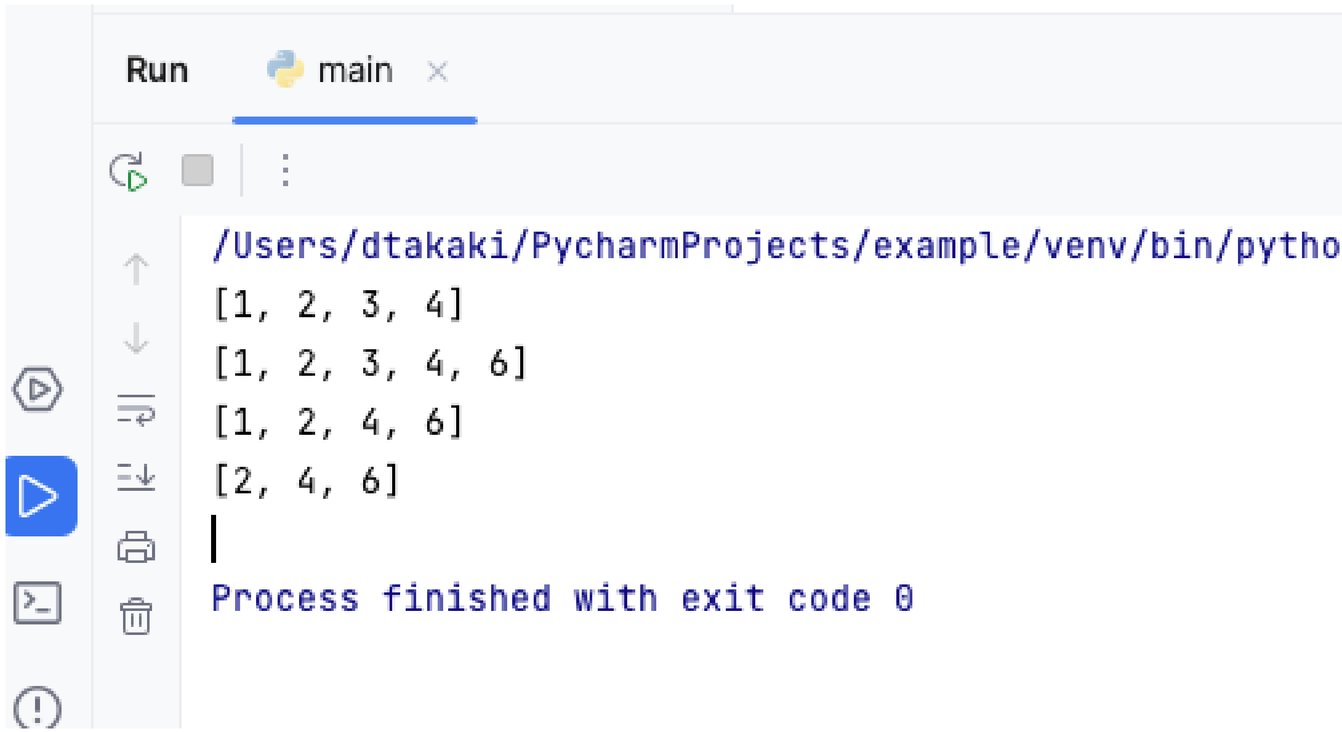
Lists

- There a methods (functions) available on list items that allow you to add or remove
- The **append()** method will add new elements to the list
- The **pop()** method and the **remove()** method will remove items
  - Pop removes the item at the specified **index**
  - Remove removes the item with the specified **value**

Lists

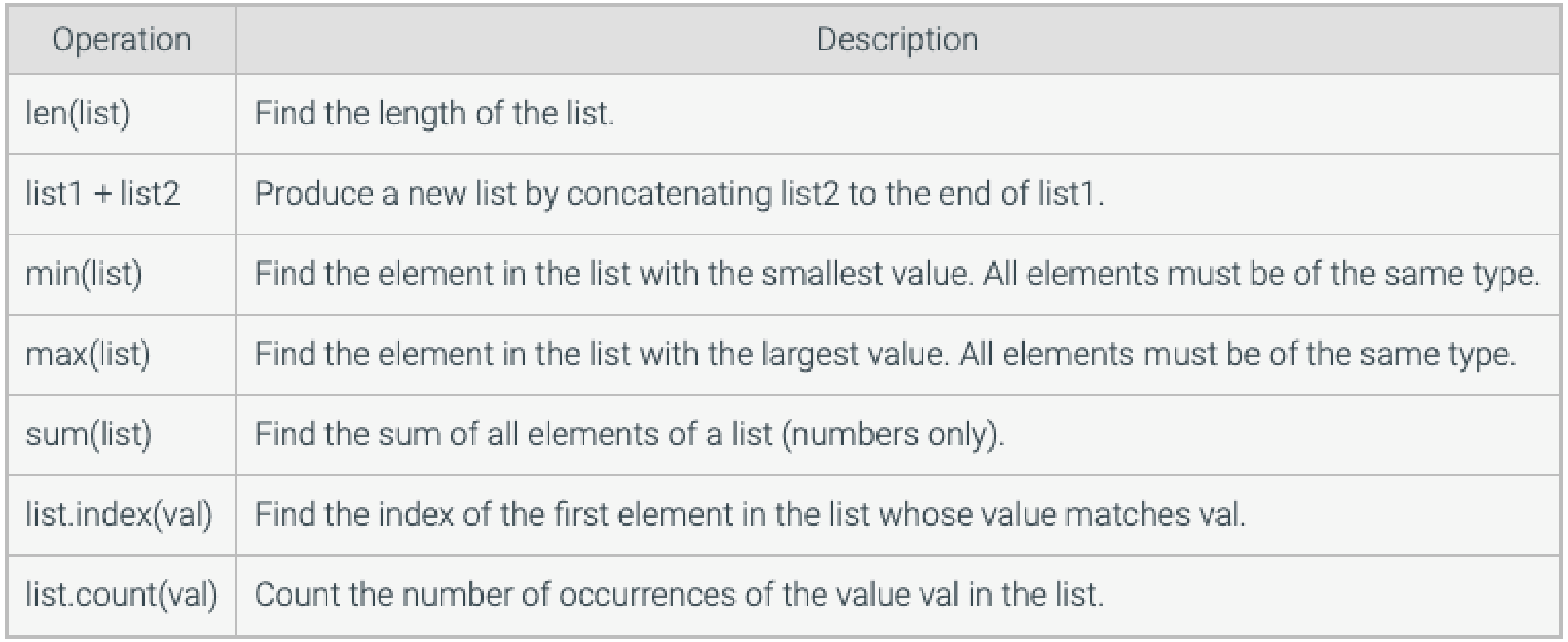


```
numbersList = [1,2,3,4]
print(numbersList)
numbersList.append(6)
print(numbersList)
numbersList.pop(2)
print(numbersList)
numbersList.remove(1)
print(numbersList)
```



```
/Users/dtakaki/PycharmProjects/example/venv/bin/pytho
[1, 2, 3, 4]
[1, 2, 3, 4, 6]
[1, 2, 4, 6]
[2, 4, 6]

Process finished with exit code 0
```

Useful methods and function



Operation	Description
len(list)	Find the length of the list.
list1 + list2	Produce a new list by concatenating list2 to the end of list1.
min(list)	Find the element in the list with the smallest value. All elements must be of the same type.
max(list)	Find the element in the list with the largest value. All elements must be of the same type.
sum(list)	Find the sum of all elements of a list (numbers only).
list.index(val)	Find the index of the first element in the list whose value matches val.
list.count(val)	Count the number of occurrences of the value val in the list.

Tuples

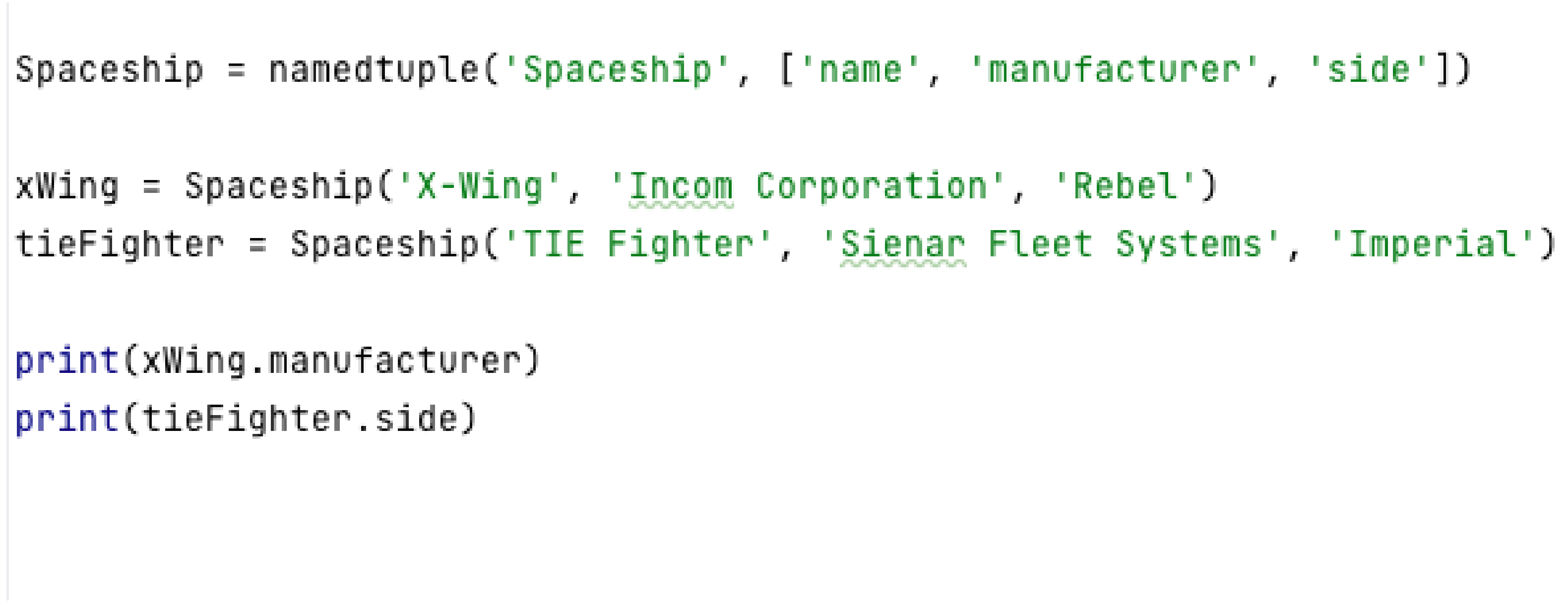
- Tuples are sequence types used to store a collection of data
    - They are immutable
    - Tuples are surrounded by parentheses ( )

```
coordinates = (83.232, 32.321)
```

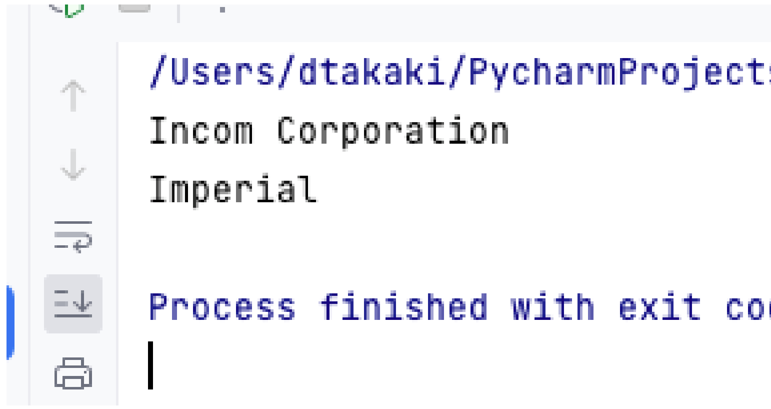
Named tuples

- Allow programmers to define new simple data with some named attributes
- Steps for creating a namedtuple
  1. Import namedtuple container
  2. Create the tuple
  3. Use the named tuple
- Attributes can be accessed using dot notation

Named Tuples



```
Spaceship = namedtuple('Spaceship', ['name', 'manufacturer', 'side'])

xWing = Spaceship('X-Wing', 'Incom Corporation', 'Rebel')
tieFighter = Spaceship('TIE Fighter', 'Sienar Fleet Systems', 'Imperial')

print(xWing.manufacturer)
print(tieFighter.side)
```



```
/Users/dtakaki/PycharmProject
Incom Corporation
Imperial

Process finished with exit co
```

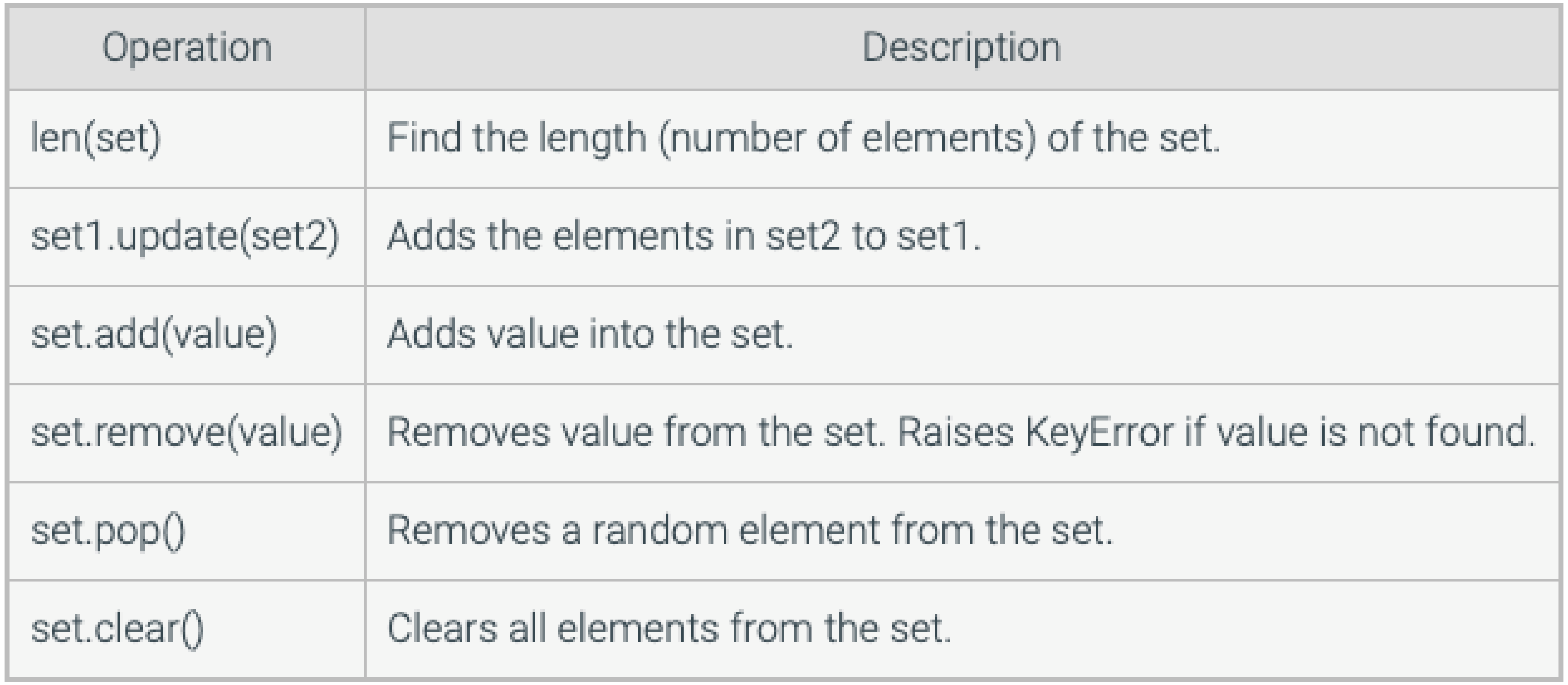
Sets

- Sets are unordered collections of unique types
    - Elements do not have a position or index
    - No repeating elements allowed
- Created using the set() function with a series of literals
- Written as a set literal using { }
    - Like a list, but uses curly braces instead of square brackets

Sets

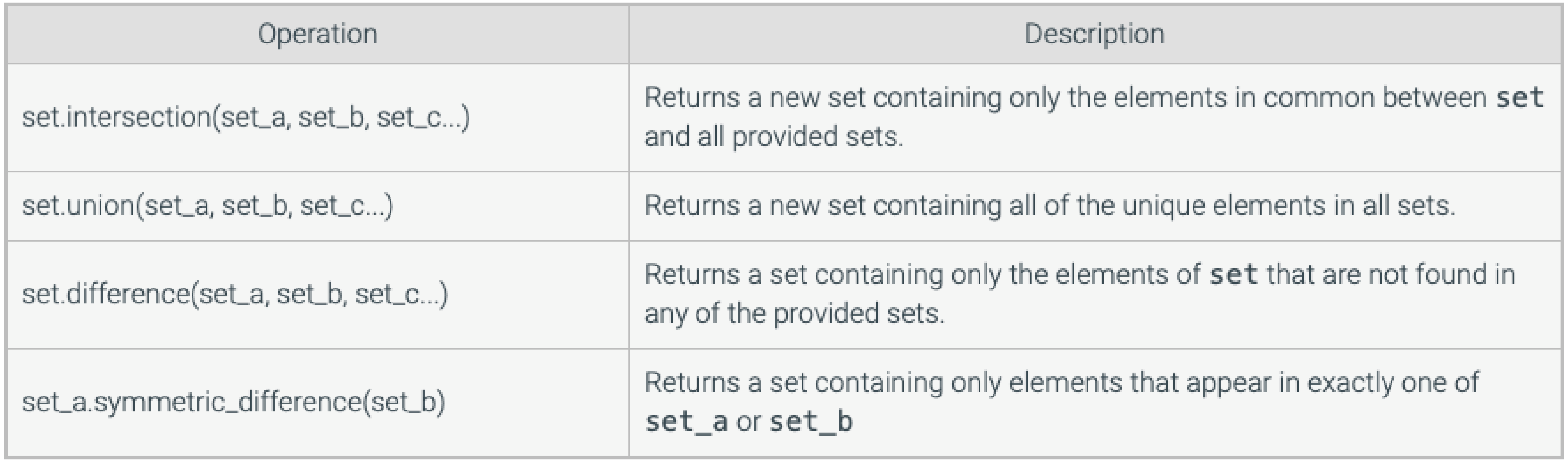
- Sets are mutable
  - The add() method will add elements to a set
  - The remove() method will remove elements
    - Removes the element that matches the given value
  - The pop() method will remove elements
    - This removes a random element

Set functions and methods



Operation	Description
len(set)	Find the length (number of elements) of the set.
set1.update(set2)	Adds the elements in set2 to set1.
set.add(value)	Adds value into the set.
set.remove(value)	Removes value from the set. Raises KeyError if value is not found.
set.pop()	Removes a random element from the set.
set.clear()	Clears all elements from the set.

Set Operations

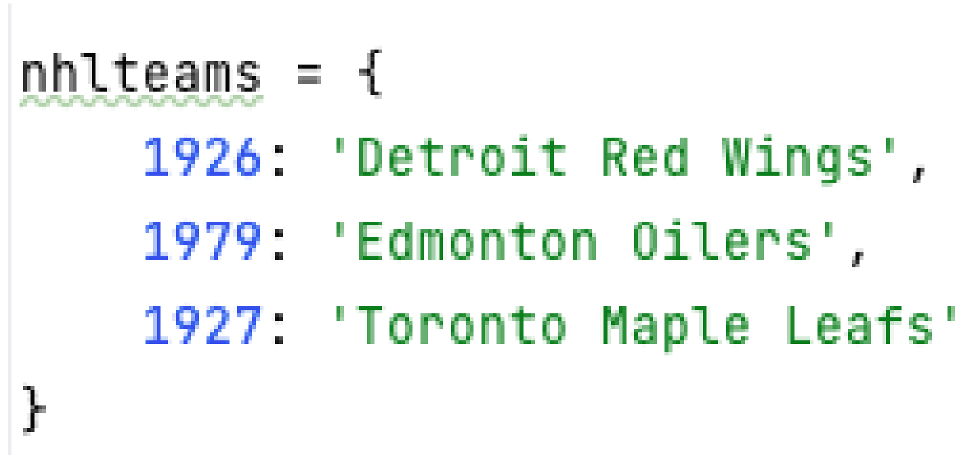


Operation	Description
set.intersection(set_a, set_b, set_c...)	Returns a new set containing only the elements in common between `set` and all provided sets.
set.union(set_a, set_b, set_c...)	Returns a new set containing all of the unique elements in all sets.
set.difference(set_a, set_b, set_c...)	Returns a set containing only the elements of `set` that are not found in any of the provided sets.
set_a.symmetric_difference(set_b)	Returns a set containing only elements that appear in exactly one of `set_a` or `set_b`

Dictionary

- Dictionary is a container with an *associative relationship*
  - Represented by the **dict** object type, it is a combination of **key** and **value** pairs
    - The **key** – is a unique term
    - The **value** – the data associated with the term
  - The key is separated from the value with a : and multiple key, value pairs are separated by a ,
  - All are surrounded by { } curly braces

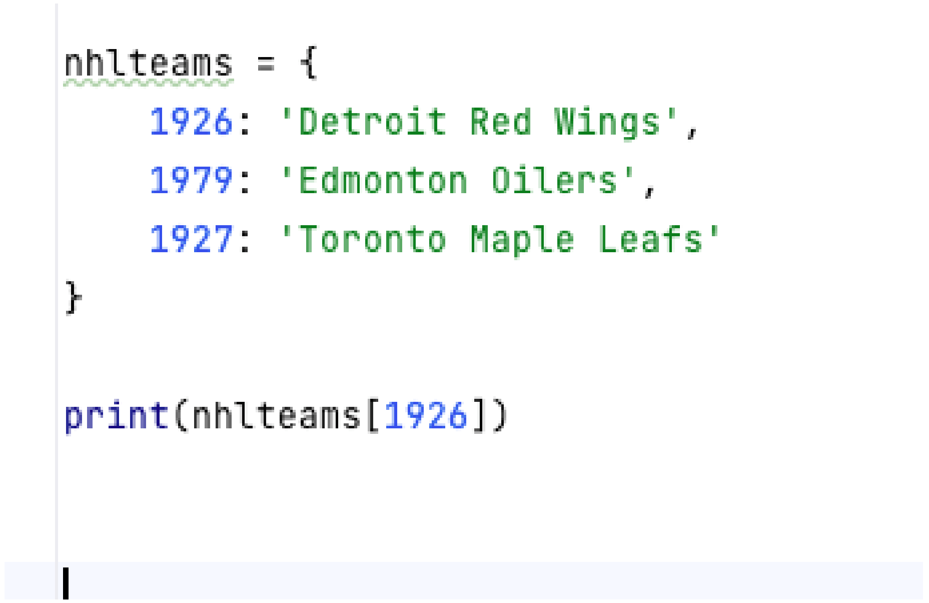
Dictionaries



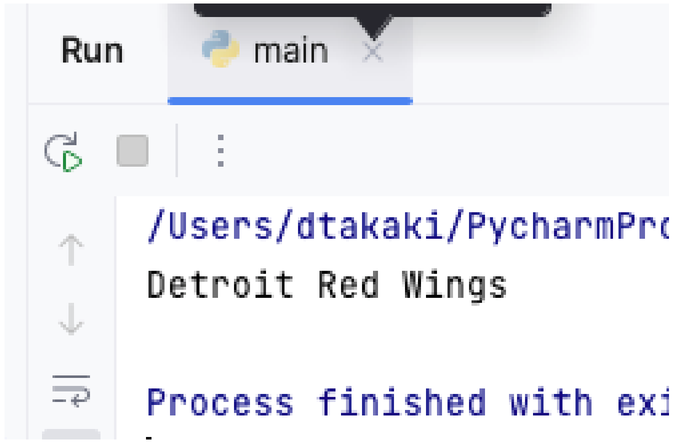
```
nhlteams = {
    1926: 'Detroit Red Wings',
    1979: 'Edmonton Oilers',
    1927: 'Toronto Maple Leafs'
}
```

Dictionaries

- Accessing items is not achieved through the index value
- Access entries using keys



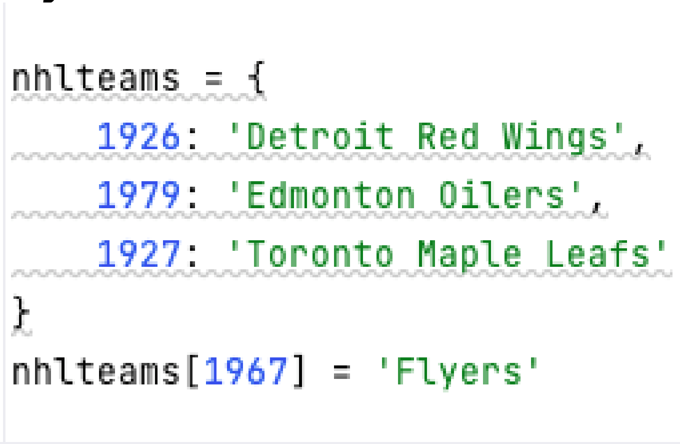
```
nhlteams = {
    1926: 'Detroit Red Wings',
    1979: 'Edmonton Oilers',
    1927: 'Toronto Maple Leafs'
}

print(nhlteams[1926])
```



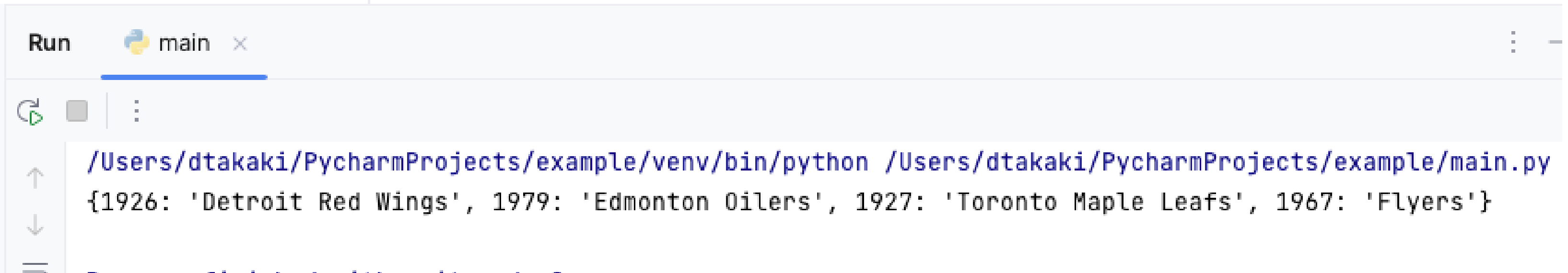
```
/Users/dtakaki/PycharmPro
Detroit Red Wings

Process finished with exi
```

Dictionaries

- Adding to dictionaries
  - Provide a new key and new value



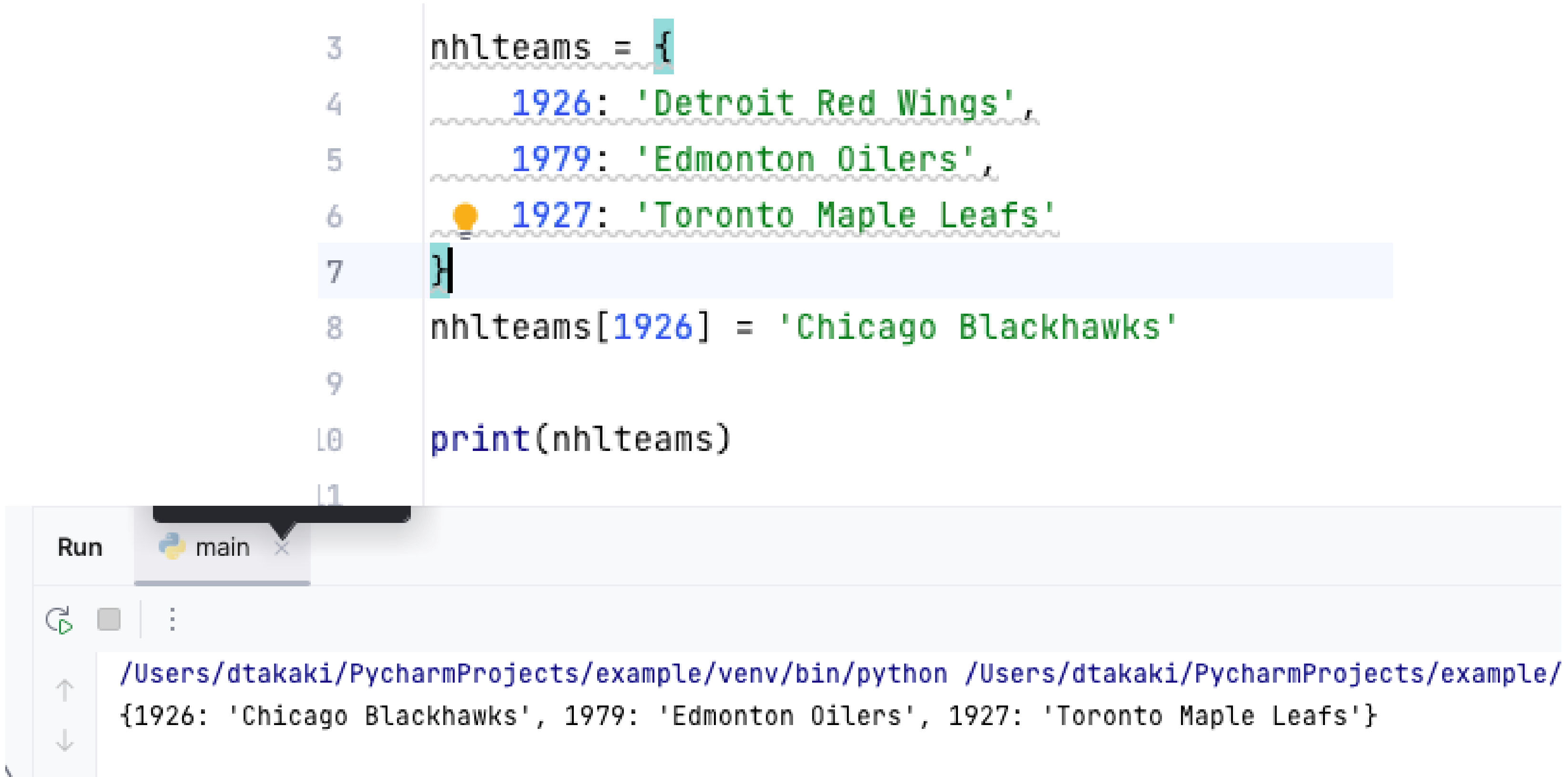
```
nhlteams = {
    1926: 'Detroit Red Wings',
    1979: 'Edmonton Oilers',
    1927: 'Toronto Maple Leafs'
}
nhlteams[1967] = 'Flyers'
```



```
/Users/dtakaki/PycharmProjects/example/venv/bin/python /Users/dtakaki/PycharmProjects/example/main.py
{1926: 'Detroit Red Wings', 1979: 'Edmonton Oilers', 1927: 'Toronto Maple Leafs', 1967: 'Flyers'}
```

Dictionaries

- Modifying – use the key and assign a new value

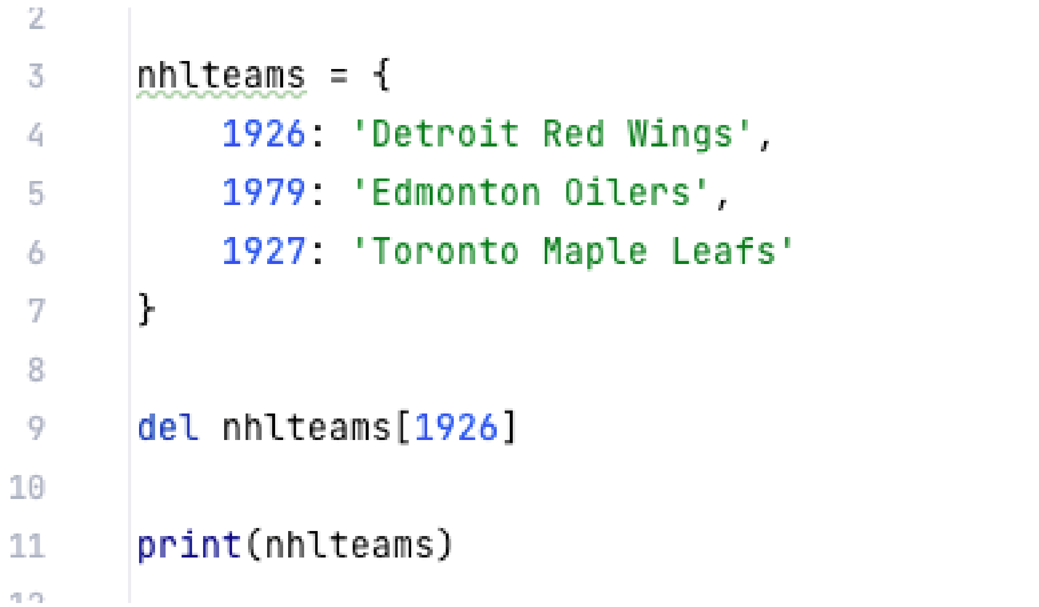


```
nhlteams = {
    1926: 'Detroit Red Wings',
    1979: 'Edmonton Oilers',
    1927: 'Toronto Maple Leafs'
}
nhlteams[1926] = 'Chicago Blackhawks'

print(nhlteams)
```

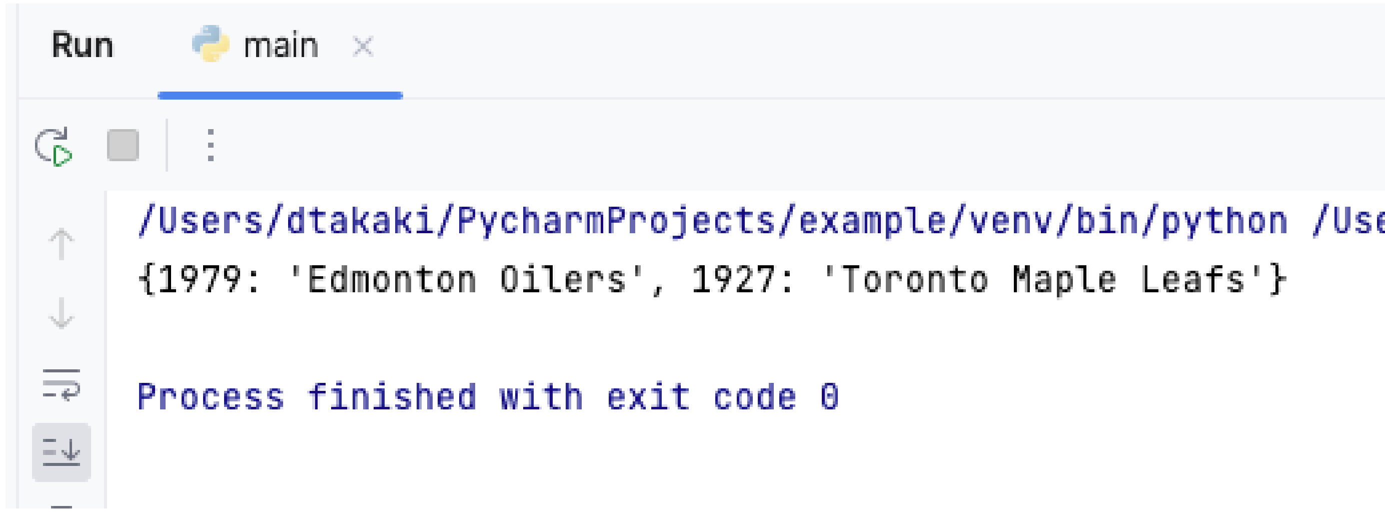
```
/Users/dtakaki/PycharmProjects/example/venv/bin/python /Users/dtakaki/PycharmProjects/example/
{1926: 'Chicago Blackhawks', 1979: 'Edmonton Oilers', 1927: 'Toronto Maple Leafs'}
```

Dictionaries

- Use the del keyword with the key and it will remove (delete) the entry



```python
nhlteams = {
    1926: 'Detroit Red Wings',
    1979: 'Edmonton Oilers',
    1927: 'Toronto Maple Leafs'
}

del nhlteams[1926]

print(nhlteams)
```



```
/Users/dtakaki/PycharmProjects/example/venv/bin/python /Use
{1979: 'Edmonton Oilers', 1927: 'Toronto Maple Leafs'}

Process finished with exit code 0
```