Cheat sheet for Python Topics Cheat Sheet

by srinivas.ram via cheatography.com/183208/cs/38151/

Python Lists

#Definition:

A list is a mutable collection of ordered elements enclosed in square brackets []. Lists can contain any type of data, including other lists.

#Creating a List :

Lists can be created using square brackets [] or the list() constructor.

#Using Square brackets:

my_list = [1, 2, 3]

# Using the list () constructor:

my_list = list([1, 2, 3])

myList = list(range(5)) # a list with five items [0, 1, 2, 3, 4]

myList = [i*2 for i in range(5)] a list with five items [0, 2, 4, 6, 8]

Accessing Elements :

Elements in a list can be accessed using indexing or slicing.

# Accessing an element using indexing :

my_list = [1, 2, 3, 4, 5]

Print(my_list[0]) # output: 1 accesses the first element in the list (1).

# Accessing a slice of elements using slicing :

print(my_list[1:4])) # Output: [2, 3, 4]

Some examples on indexing and slicing :

my_list = ['apple', 'banana', 'cherry']

print(my_list[1]) # Output: banana accesses the second element in the list ('banana').

print(my_list[-1]) # Output: cherry accesses the last element in the list ('cherry').

print(my_list[1:]) # Output: ['banana', 'cherry'] accesses all elements in the list from the second element to the end

Modifying Elements :

Python Lists (cont)

Elements in a list can be modified using indexing or slicing.

my_list = [1, 2, 3]

my_list[0] = 4

print(my_list) # Output: [4, 2, 3]

Some Examples to modify elements using index and slice()

my_list = ['apple', 'banana', 'cherry']

my_list[1] = 'orange' changes the second element in the list to 'orange'.

my_list.append('grape') adds 'grape' to the end of the list.

my_list.extend(['kiwi', 'watermelon']) adds the list ['kiwi', 'watermelon'] to the end of the list.

my_list.remove('cherry') removes the element 'cherry' from the list.

my_list.pop(0) removes and returns the first element in the list ('apple').

List Methods :

Lists have many built-in methods, including:

# Using append() method :

my_list = [1, 2, 3]

my_list.append(4)

print(my_list)

# Output: [1, 2, 3, 4]

# Using extend() method :

my_list.extend([5, 6])

print(my_list)

# Output: [1, 2, 3, 4, 5, 6]

# Using insert() method :

my_list.insert(0, 0)

print(my_list)

# Output: [0, 1, 2, 3, 4, 5, 6]

# Using remove() method :

my_list.remove(3)

print(my_list)

# Output: [0, 1, 2, 4, 5, 6]

# Using pop() method :

my_list.pop(2)

print(my_list)

# Output: [0, 1, 4, 5, 6]

Python Lists (cont)

# Using sort() method :

my_list.sort()

print(my_list)

# Output: [0, 1, 4, 5, 6]

# Using reverse() method :

my_list.reverse()

print(my_list)

# Output: [6, 5, 4, 1, 0]

Copying a List:

my_list = ['apple', 'banana', 'cherry']

new_list = my_list.copy()

Nested Lists:

my_list = [[1, 2], [3, 4]]

print(my_list[0][1]) # Output: 2

List Functions:

sum(my_list)

all(my_list)

any(my_list)

enumerate(my_list)

zip(my_list1, my_list2)

List Comprehensions :

List comprehensions provide a concise way to create lists based on existing lists.

# Creating a new list using list comprehension :

my_list = [1, 2, 3, 4, 5]

new_list = [x * 2 for x in my_list]

print(new_list) # Output: [2, 4, 6, 8, 10]

my_list = [x for x in range(1, 6)]

even_list = [x for x in range(1, 11) if x % 2 == 0]

Advantage of Using Lists :

Lists are mutable, which makes them more flexible to use than tuples.

Tuples In Python

What are Tuples?

A tuple is an ordered, immutable collection of elements.
In Python, tuples are created using parentheses ()
and the elements are separated by commas ,.

Creating Tuples

```
# Create an empty tuple
my_tuple = ()
# Create a tuple with elements
my_tuple = (1, 2, 3)
# Create a tuple with a single element
my_tuple = (1,)
# Create a tuple without parentheses
my_tuple = 1, 2, 3
```

Accessing Elements

Tuples are ordered collections,
So you can access individual elements using indexing.

```
my_tuple = ('apple', 'banana', 'cherry')
```

# Access the first element

```
print(my_tuple[0]) # Output: 'apple'
```

# Access the last element

```
print(my_tuple[-1]) # Output: 'cherry'
```

Immutability

Tuples are immutable, which means that you can't modify their contents after they're created.

```
my_tuple = (1, 2, 3)
# Trying to modify the first element will result in an error
my_tuple[0] = 4 # TypeError: 'tuple' object does not support item assignment
```

Tuple Methods

Tuples have a few built-in methods that you can use:

count()

Tuples In Python (cont)

Returns the number of times a specified value occurs in a tuple.

```
my_tuple = (1, 2, 2, 3, 2, 4)
# Count the number of times the value 2 appears
print(my_tuple.count(2)) # Output: 3
```

index()

Returns the index of the first occurrence of a specified value in a tuple.

```
my_tuple = (1, 2, 2, 3, 2, 4)
# Find the index of the first occurrence of the value 3
print(my_tuple.index(3)) # Output: 3
```

Tuple Unpacking

You can also "unpack" tuples, which allows you to assign the values in a tuple to separate variables.

```
my_tuple = ('John', 'Doe', 30)
# Unpack the tuple into separate variables
first_name, last_name, age = my_tuple
print(first_name) # Output: 'John'
print(last_name) # Output: 'Doe'
print(age) # Output: 30
```

Advantages of Tuples

Tuples are immutable, so they're useful for storing data that shouldn't be changed accidentally.
Tuples are faster than lists, since they're smaller and can be stored more efficiently in memory.
Tuples can be used as dictionary keys, while lists cannot.

Dictionaries in Python

Creating a dictionary

```
# Empty dictionary
my_dict = {}
# Dictionary with initial values
my_dict = {"key1": "value1", "key2": "value2", "key3": "value3"}
# Using the dict() constructor
my_dict = dict(key1="value1", key2="value2", key3="value3")
```

Accessing values

Dictionaries in Python (cont)

```
# Accessing a value by key
my_dict["key1"] # returns "value1"
# Using the get() method to avoid KeyError
my_dict.get("key1") # returns "value1"
my_dict.get("key4") # returns None
# Using the get() method with a default value
my_dict.get("key4", "default_value") # returns "default_value"
```

Adding and updating values

```
# Adding a new key-value pair
my_dict["key4"] = "value4"
# Updating a value
my_dict["key1"] = "new_value1"
# Using the update() method to add/update multiple values
my_dict.update({"key5": "value5", "key6": "value6"})
```

Removing values

```
# Removing a key-value pair
del my_dict["key1"]
# Using the pop() method to remove a key-value pair and return the valmy_dict.pop("key2") # returns "value2"ue

# Using the pop() method with a default value
my_dict.pop("key4", "default_value") # returns "default_value"
# Using the clear() method to remove all key-value pairs
my_dict.clear()
```

Other methods

```
# Getting the number of key-value pairs
len(my_dict)
# Checking if a key exists in the dictionary
"key1" in my_dict
# Getting a list of keys
my_dict.keys()
# Getting a list of values
my_dict.values()
# Getting a list of key-value pairs as tuples
my_dict.items()
```

Advantages of Python dictionaries

Dictionaries in Python (cont)

Python dictionaries offer fast lookups, flexible key/value storage,
dynamic resizing, efficient memory usage, and ease of use, making them a versatile and powerful data structure widely used in Python programming.

Python strings

Creating strings

```
my_string = "Hello, World!" # double quotes
my_string = 'Hello, World!' # single quotes
my_string = """Hello, World!""" # triple quotes (for multiline strings)
```

Accessing characters in a string

```
my_string = "Hello, World!"
print(my_string[0]) # H
print(my_string[-1]) # !
```

Slicing strings

```
my_string = "Hello, World!"
print(my_string[0:5]) # Hello
print(my_string[7:]) # World!
print(my_string[:5]) # Hello
print(my_string[::2]) # Hlo ol!
```

String methods

```
my_string = "Hello, World!"
print(my_string.upper()) # HELLO, WORLD!
print(my_string.lower()) # hello, world!
print(my_string.replace("Hello", "Hi")) # Hi, World!
print(my_string.split(",")) # ['Hello', ' World!']
print(my_string.strip()) # Hello, World! (remove whitespace)
print(len(my_string)) # 13 (length of the string)
```

Concatenating strings

```
str1 = "Hello"
str2 = "World"
print(str1 + " " + str2) # Hello World
```

String formatting

Python strings (cont)

```
name = "John"
age = 30
print("My name is {} and I'm {} years old".format(name, age))
# My name is John and I'm 30 years old
# f-strings (Python 3.6+)
print(f"My name is {name} and I'm {age} years old")
# My name is John and I'm 30 years old
```

Encoding and decoding strings

```
my_string = "Hello, World!"
encoded_string = my_string.encode("utf-8")
# b'Hello, World!'
decoded_string = encoded_string.decode("utf-8") # Hello, World!
```

Advantages of strings in Python

Strings in Python have several advantages, including their flexibility,
ease of use, and extensive library of built-in string methods,
making it easy to manipulate and format text data for various purposes
such as data analysis, web development, and automation tasks.