Tuples

Collections in Python

Digital Career Institute

Python Tuples

Digital Career Institute

Tuples are like lists but they cannot be changed.
They are the equivalent of constants for collections and are faster than lists.

They:

- Have an **order**.
- **Allow duplicate** values.
- **Do not allow** their objects to be **changed**.
- Allow objects of **different types**.

Defining Tuples

They are defined using parentheses `()`.

Any **iterable** can be converted into a tuple by using the `tuple` constructor.

The `tuple` constructor can also be used to create empty tuples if no argument is given.

The output is the same as using empty parentheses `()`.

```
>>> days = (
...     "Monday",
...     "Tuesday",
...     "Wednesday",
...     "Thursday",
...     "Friday",
... )
>>> hello = tuple("hello")
>>> print(hello)
('h', 'e', 'l', 'l', 'o')
>>> empty_tuple = tuple()
>>> print(empty_tuple)
()
```

Defining Tuples

Digital Career Institute

Tuples can contain values of any type.

Items in the tuple can be of mixed types.

The items themselves can also be tuples.

```
>>> fridge = ("Apple", "Apple")
>>> letters = tuple("hello")
>>> ages = (32, 45, 42, 12, 34, 57)
>>> dates = (datetime, datetime)
>>> data = ("John", 32, datetime)
>>> tuples = (
...     ("John", "Mary", "Amy"),
...     (32, 43, 51)
... ]
```

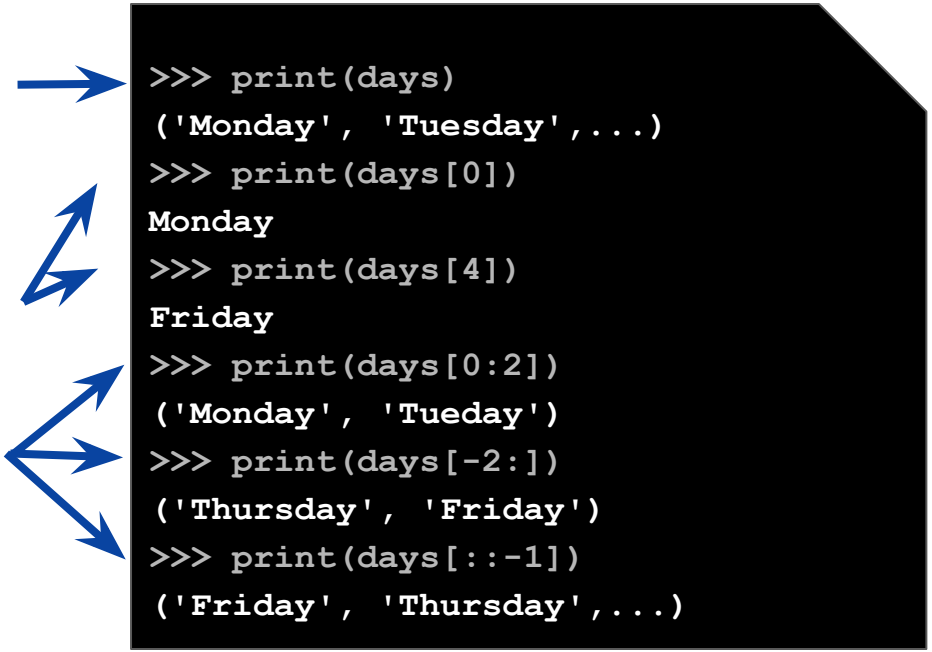
Python Tuples: Accessing Values

Digital Career Institute

Printing the tuple will show its values in the exact **same order** used when the tuple was defined.

Each value in the tuple can be accessed using its numeric **position** in the tuple (starting at zero). This is named the **index**.

Slicing can be used to access parts of the tuple or reverse its order.



Python Tuples: Accessing Values

Digital Career Institute

Tuples also have the method **`index`** implemented to return the index of the first occurrence of the given value in the tuple.

The number of occurrences of a value can also be obtained with the **`count`** method.

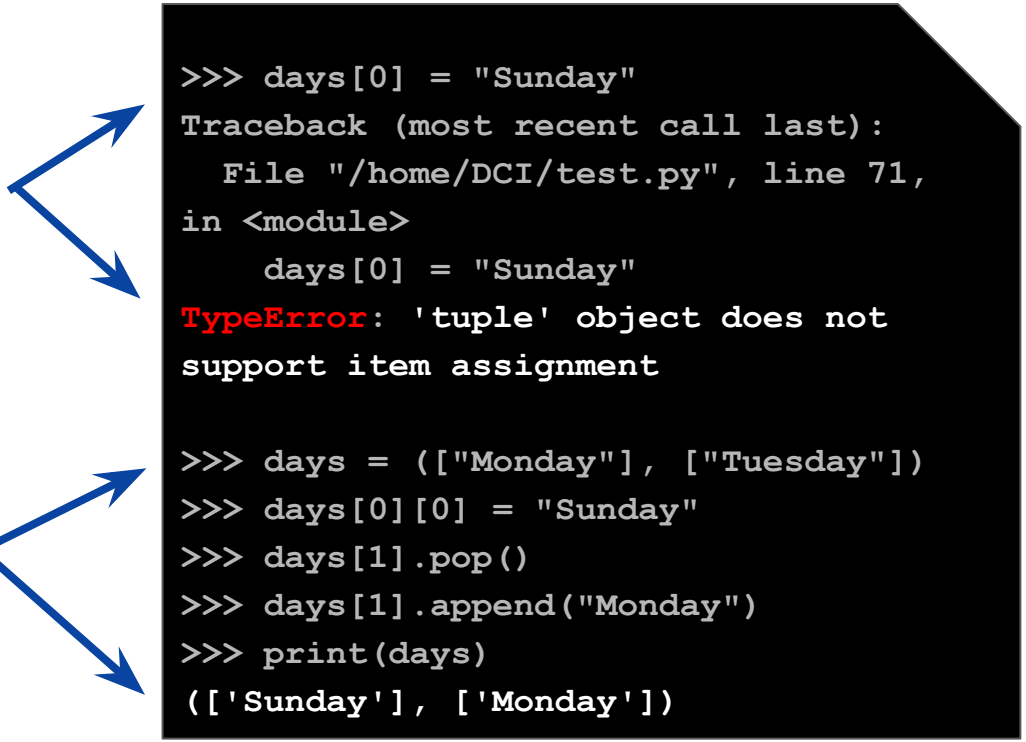
```
>>> print(days.index('Thursday'))
3
>>> print(days[3])
Thursday

>>> print(days.count('Monday'))
1
```

Python Tuples: Changing Values

The items inside a tuple cannot be changed.

If an item in the tuple is another type of iterable, its contents can still be changed.



```
>>> days[0] = "Sunday"
Traceback (most recent call last):
  File "/home/DCI/test.py", line 71,
in <module>
    days[0] = "Sunday"
TypeError: 'tuple' object does not
support item assignment

>>> days = (["Monday"], ["Tuesday"])
>>> days[0][0] = "Sunday"
>>> days[1].pop()
>>> days[1].append("Monday")
>>> print(days)
(['Sunday'], ['Monday'])
```

Python Tuples: Sort, Add & Remove

Digital Career Institute

As opposed to lists, tuples cannot be changed.

Therefore, **they do not have** any of the methods that can be used in lists to manipulate its contents:

- Append
- Extend
- Insert
- Pop
- Remove
- Sort
- Reverse

```
>>> days.append("Saturday")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no
attribute 'append'
>>> days.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no
attribute 'pop'
>>> days.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no
attribute 'sort'
```

Comparing Python Tuples

Digital Career Institute

Comparing two tuples will return **`True`** if the following statements are true for the elements in it:

- They are the same
- They are in the same order

```
>>> list1 = (1, 2, 3)
>>> list2 = (1, 2, 3)
>>> list1 == list2
True
>>> list1 is list2
False
>>> list3 = (3, 2, 1)
>>> list1 == list3
False
>>> list1 == list3[::-1]
True
>>> list4 = [1, 1, 2, 2, 3, 3]
>>> list1 == list4
False
```

Python Tuples: Use Case Examples

Digital Career Institute

WEEK DAYS

- Monday
- Tuesday
- Wednesday
- Thursday
- Friday
- Saturday
- Sunday

MONTHS

- January
- February
- March
- April
- May
- June
- July
- August
- September
- October
- November
- December

MEDAL TYPES

- Gold
- Silver
- Bronze

Python Tuple Methods: Summary

Digital Career Institute

Tuples only have methods to search and analyze the values:

- Index
- Count

```
>>> print(dir(days))
[..., 'count', 'index']
```

Sets

Collections in Python

Digital Career Institute

Python Sets

Sets distinctive feature is that they do not allow duplicate values.

They:

- Have an **no order**.
- **Do not store duplicate** values.
- **Do not allow** their objects to be **changed**.
- Allow objects of **different types**.

Defining Sets

They are defined using curly brackets `{}`.

Any **iterable** can be converted into a set by using the `set` constructor.

The `set` constructor can also be used to create empty sets if no argument is given.

```
>>> fruits = {
...     "Apple",
...     "Orange",
...     "Pear",
...     "Banana",
...     "Apricot"
... }
>>> hello = set("hello")
>>> print(hello)
{'o', 'e', 'h', 'l'}
>>> empty_set = set()
>>> print(empty_set)
set()
```

Defining Sets

Digital Career Institute

Sets can contain values of any type. Repeated values can be added, but they will only be stored once.

Items in the set can be of mixed types.

But the items themselves can not be sets.

A set can also be created using the `copy` method of another set.

```
>>> fruits = {"Apple", "Apple"}
>>> letters = set("hello")
>>> ages = {32, 45, 42, 12, 34, 57}
>>> dates = {datetime, datetime}
>>> data = {"John", 32, datetime}
>>> sets = {
...     {"John", "Mary", "Amy"},
...     {32, 43, 51}
... }
TypeError: unhashable type: 'set'
>>> copy = data.copy()
```

Python Sets: Accessing Values

Digital Career Institute

Printing the set will show its values in a **different order** than the one used when the set was defined.

The values in a set cannot be accessed directly using indexing.

Therefore, there is no `index` method. Because the set has no order, it has no method `sort` or `reverse`. And because the set has no repeated values, it does not have the `count` method either.

```
>>> print(fruits)
{'Apricot', 'Banana', 'Pear',...}

>>> print(fruits[0])
Traceback (most recent call
last):
  File "<stdin>", line 1, in
<module>
TypeError: 'set' object does not
support indexing
```

Python Sets: Adding Values

Digital Career Institute

Adding new values can be done with the method **`add`**. The new value may show anywhere, as there is no order.

The **`update`** method can be used to add multiple values at once. It accepts any iterable as an argument.

Adding a value twice throws no error, but only stores the value once.

```
>>> fruits.add('Pineapple')
>>> print(fruits)
{'Apricot', 'Pineapple',...}

>>> fruits.update(
...     ['Mango', 'Mango']
... )
>>> fruits
{'Apricot', 'Pineapple',
'Banana', 'Mango', 'Pear',
'Apple', 'Orange'}
```

Python Sets: Removing Values

Digital Career Institute

The method **pop** will remove a random element and will return it. It accepts no argument.

The **discard** method removes the given value and returns nothing.

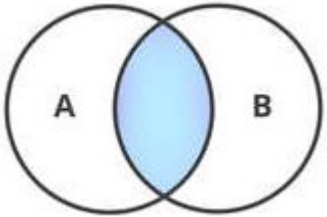
The **remove** method is like **discard** but it throws an error exception if the value does not exist in the set.

The **clear** method will remove all items in the set.

```
>>> random = fruits.pop()
>>> print(random)
Apricot
>>> print(fruits.discard('Mango'))
None
>>> print(fruits)
{'Pineapple', 'Banana', 'Pear',
'Apple', 'Orange'}
>>> fruits.remove('Mango')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Mango'
>>> fruits.clear()
>>> print(fruits)
set()
```

Python Sets: Set Operations

The set type includes methods to perform the standard operations between sets of Set Theory and return new sets.

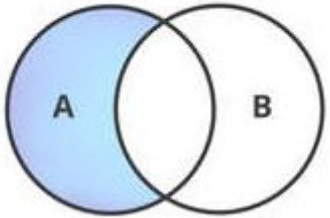


A and B

The method **`intersection`** returns a set containing the values present in both sets. The original sets remain unchanged.

```
>>> fruits = {
...     "Pear",
...     "Mango",
...     "Apple",
...     "Strawberry"
... }
>>> smoothie = {
...     "Mango",
...     "Orange",
...     "Pear"
... }
>>> fruits.intersection(smoothie)
{'Mango', 'Pear'}
```

Python Sets: Set Operations

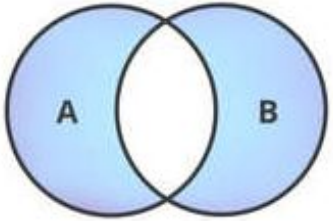


A - B

The method **`difference`** returns a set containing the values present in A that don't exist in B. The original sets remain unchanged.

```
>>> fruits = {
...     "Pear",
...     "Mango",
...     "Apple",
...     "Strawberry"
... }
>>> smoothie = {
...     "Mango",
...     "Orange",
...     "Pear"
... }
>>> fruits.difference(smoothie)
{'Strawberry', 'Apple'}
```

Python Sets: Set Operations



A xor B

The method **`symmetric_difference`** returns a set containing the values present in A or B that don't exist at the same time in both. The original sets remain unchanged.

```
>>> fruits = {
...     "Pear",
...     "Mango",
...     "Apple",
...     "Strawberry"
... }
>>> smoothie = {
...     "Mango",
...     "Orange",
...     "Pear"
... }
>>> fruits.symmetric_difference(smoothie)
{'Orange', 'Strawberry', 'Apple'}
```

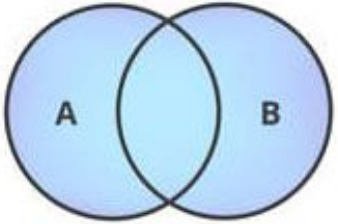
Python Sets: Set Operations

The **`difference_update`**, **`intersection_update`** and **`symmetric_difference_update`** methods will calculate the same thing as do the **`difference`**, **`intersection`** and **`symmetric_difference`** methods.

The difference is that the first methods will overwrite the original set (fruits, in this case) instead of returning the result.

```
>>> fruits = {
...     "Pear",
...     "Mango",
...     "Apple",
...     "Strawberry"
... }
>>> smoothie = {
...     "Mango",
...     "Orange",
...     "Pear"
... }
>>> fruits.difference_update(smoothie)
>>> print(fruits)
{'Strawberry', 'Apple'}
```

Python Sets: Set Operations



A or B

The method **`union`** returns a set containing the values present in A or B (or both). The original sets remain unchanged.

```
>>> fruits = {
...     "Pear",
...     "Mango",
...     "Apple",
...     "Strawberry"
... }
>>> smoothie = {
...     "Mango",
...     "Orange",
...     "Pear"
... }
>>> fruits.union(smoothie)
{'Strawberry', 'Orange', 'Mango', 'Pear',
'Apple'}
```

Python Sets: Set Operations

Digital Career Institute

The previous methods return a set with the result. Sometimes the script only requires to know if some kind of relationships exists between the sets, not the particular values.

The method `isdisjoint` returns `True` if the two sets have no item in common and `False` if they share at least one value.

```
>>> fruits = {
...     "Pear",
...     "Mango",
...     "Apple",
...     "Strawberry"
... }
>>> smoothie = {
...     "Mango",
...     "Orange",
...     "Pear"
... }
>>> fruits.isdisjoint(smoothie)
False
```

Python Sets: Set Operations

Digital Career Institute

The method `issubset` returns `True` if the first set is completely contained in the set passed as argument.

The method `issuperset` returns `True` if the first set completely contains the set passed as argument.

```
>>> fruits = {
...     "Pear",
...     "Mango",
...     "Apple",
...     "Strawberry"
... }
>>> smoothie = {
...     "Mango",
...     "Pear"
... }
>>> fruits.issubset(smoothie)
False
>>> smoothie.issubset(fruits)
True
```

Comparing Python Sets

Digital Career Institute

Comparing two sets will return `True` if the following statement is true for the elements in it:

- They are the same

A set is very useful when we need to know if the items in two iterables are the same.

Using `set()` on both iterables before comparing them will remove duplicates and ignore the order.

```
>>> set1 = {1, 2, 3}
>>> set2 = {1, 2, 3}
>>> set1 == set2
True
>>> set1 is set2
False
>>> set1 == {3, 2, 1}
True
>>> set1 == {1, 1, 2, 2, 3, 3}
True
>>> list1 = [1, 2, 3]
>>> list2 = [3, 1, 2, 1, 3, 2]
>>> set(list1) == set(list2)
True
```

Python Sets: Use Case Examples

Digital Career Institute

CITIES VISITED	REGISTERED	SPORTS
- Berlin	- Mary	- Badminton
- Barcelona	- John	- Tennis
- Stockholm	- Eva	- Athletics
- Trondheim	- Susie	- Swimming
- Salzburg	- Peter	- Basketball
- Brno	- Lucy	- Football
- Girona	- Ronnie	- Baseball
- Manchester	- Gerald	- Table-tennis
- Ljubljana	- Anna	- Skiing
- Tijuana	- Anthony	- Curling

Python Set Methods: Summary

Digital Career Institute

Sets have a long number of methods:

Add
- Add
- Update

Remove
- Pop
- Remove
- Discard
- Clear

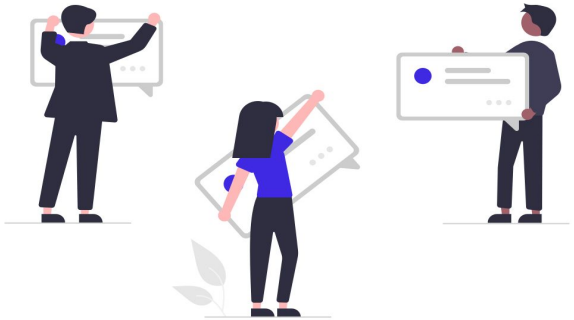
Manage
- Copy

Set Operations
- Intersection
- Difference
- ...

```
>>> print(dir(fruits))
[..., 'add', 'clear', 'copy',
'difference', 'difference_update',
'discard', 'intersection',
'intersection_update',
'isdisjoint', 'issubset',
'issuperset', 'pop', 'remove',
'symmetric_difference',
'symmetric_difference_update',
'union', 'update']
```

We learned ...

- That one of the most common type of collections in Python is the list. Lists have an implicit order according to the position of the value in the list.
- That lists can be manipulated to add, remove and changes its elements.
- That tuples are very similar to lists, but they are immutable. Their elements cannot be changed.
- That comparing between lists works the same as comparing between tuples. They are only the same if they have the same values and in the same order.
- That sets, as opposed to lists and tuples, do not allow repeated values and their values have no order.
- That two sets are the same if they have the same values, no matter the order in which they were added to the set.



Expert Round

- How does comparison work in detail?
- What's the difference between a set and a list?
- Why are sets considered associative collections?
- What are the advantages of using tuples over lists?

Digital Career Institute