python_data_structure

September 18, 2021

1 Module 3 (Pandas Core): Data Structures

1.1 Table of Contents

1.2 Import libraries

[ ]:
```
import pandas as pd
import numpy as mp
from pandas_extensions.database import collect_data
```

1.3 Objects

[ ]:
```
# Import data
df = pd.DataFrame(collect_data())
# Get object class
print(type(df))
```

```
<class 'pandas.core.frame.DataFrame'>
```

[ ]:
```
# Get the object's inheritance structure in the order that methods are searched␣
 ↪for
print(type(df).mro())
```

```
[<class 'pandas.core.frame.DataFrame'>, <class 'pandas.core.generic.NDFrame'>,
<class 'pandas.core.base.PandasObject'>, <class
'pandas.core.accessor.DirNamesMixin'>, <class
'pandas.core.base.SelectionMixin'>, <class
'pandas.core.indexing.IndexingMixin'>, <class 'pandas.core.arraylike.OpsMixin'>,
<class 'object'>]
```

It turns out that Python's objects get methods and attributes from the classes that they inherit from. The output above is the search path similar to R's pacakge environments. This allows us to use methods from all the classes in the inheritance structure.

[ ]:
```
# Objects have attributes
# Vscode shows object attributes with the wrench icon
print(df.shape)
```

```
(15644, 13)
```

[ ]:
```
print(df.columns)
```

```
Index(['order_id', 'order_line', 'order_date', 'quantity', 'price',
       'total_revenue', 'model', 'category_1', 'category_2', 'frame_material',
       'bikeshop_name', 'city', 'state'],
      dtype='object')
```

[ ]:
```
# Objects have methods
# Vscode shows object methods with the cube
print(df.query("order_id == 3"))
```

```
   order_id  order_line order_date  quantity  price  total_revenue  \
4         3           1 2011-01-10         1  10660          10660
5         3           2 2011-01-10         1   3200           3200
6         3           3 2011-01-10         1  12790          12790
7         3           4 2011-01-10         1   5330           5330
8         3           5 2011-01-10         1   1570           1570

                          model category_1      category_2 frame_material  \
4        Supersix Evo Hi-Mod Team       Road      Elite Road         Carbon
5                  Jekyll Carbon 4   Mountain   Over Mountain         Carbon
6         Supersix Evo Black Inc.       Road      Elite Road         Carbon
7  Supersix Evo Hi-Mod Dura Ace 2       Road      Elite Road         Carbon
8                Synapse Disc 105       Road  Endurance Road       Aluminum

              bikeshop_name        city state
4  Louisville Race Equipment  Louisville    KY
5  Louisville Race Equipment  Louisville    KY
6  Louisville Race Equipment  Louisville    KY
7  Louisville Race Equipment  Louisville    KY
8  Louisville Race Equipment  Louisville    KY
```

1.4 Key data structure

Data frame is a key structure that holds Pandas series; it has columns and index attributes. The data frame is an object with many methods.

[ ]:
```
# Each column in a data frame is a pandas series
print(type(df["order_date"]))
```

```
<class 'pandas.core.series.Series'>
```

[ ]:
```
# Each series has methods that are dependent on its attributes
# Accessors can be used to call the series attribute "dt" and extract the year␣
↪from the column
df["order_date"].dt.year
```

[ ]:
```
0        2011
1        2011
2        2011
3        2011
4        2011
         …
15639    2015
15640    2015
15641    2015
15642    2015
15643    2015
Name: order_date, Length: 15644, dtype: int64
```

Pandas series are built on top of Numpy arrays. The series adds an index and meta data like series name. The Numpy array then provides core functionality, like `numpy.sum()`.

[ ]:
```
# Acess the array from the series
df["order_date"].values
```

[ ]:
```
array(['2011-01-07T00:00:00.000000000', '2011-01-07T00:00:00.000000000',
       '2011-01-10T00:00:00.000000000', …,
       '2015-12-25T00:00:00.000000000', '2015-12-25T00:00:00.000000000',
       '2015-12-25T00:00:00.000000000'], dtype='datetime64[ns]')
```

[ ]:
```
# Get type
type(df["order_date"].values)
```

[ ]:
```
numpy.ndarray
```

[ ]:
```
# The numpy array is actually a low level object that only has "object" in its␣
↪inheritance
type(df["order_date"].values).mro()
```

[ ]:
```
[numpy.ndarray, object]
```

Numpy data types are extended built-in data types. It uses special data types (e.g. int64), which are usually optimized for memory allocation.

[ ]:
```
# Get classes and the data types that the numpy arrays actually contain
print(df["price"].values.dtype)
print(df["order_date"].values.dtype)
```

```
int64
datetime64[ns]
```

1.5 Built-in sequences

Container sequences

- list, tuple, and collections.deque can hold items of different types, including nested containers.

Flat sequences

- str, bytes, bytearray, memoryview, and array.array hold items of one simple type.

Another way to group these sequences is by mutability:

Mutable sequences

- list, bytearray, array.array, collections.deque, and memoryview

Immutable sequences

- tuple, str, and bytes

One important distinction between container and flat sequences is: A container sequence holds references to the objects it contains, which may be of any type, while a flat sequence stores the value of its contents in its own memory space, and not as distinct objects. Examine the diagram below:

Here is a really nice visualization of the Python object ecosystem:

Link

1.5.1 List

Lists are Python's most flexible ordered collection object type. Unlike strings, lists can contain any sort of object: numbers, strings, and even other lists. Also, unlike strings, lists may be *changed in place* by assignment to offsets and slices, list method calls, deletion statements, and more—they are **mutable** objects. Some properties are:

Ordered collections of arbitrary objects - From a functional view, lists are just places to collect other objects so you can treat them as groups. Lists also maintain a left-to-right positional ordering among the items they contain (i.e., they are sequences)

Accessed by offset - Just as with strings, you can fetch a component object out of a list by indexing the list on the object's offset. Because items in lists are ordered by their positions, you can also do tasks such as slicing and concatenation.

Variable-length, heterogeneous, and arbitrarily nestable - Unlike strings, lists can grow and shrink in place (their lengths can vary), and they can contain any sort of object, not just one-character strings (they're heterogeneous). Because lists can contain other complex objects, they also support arbitrary nesting; you can create lists of lists of lists, and so on.

Of the category "mutable sequence" - In terms of our type category qualifiers, lists are mutable (i.e., can be changed in place) and can respond to all the sequence operations used with strings, such as indexing, slicing, and concatenation. In fact, sequence operations work the same on lists as they do on strings; the only difference is that sequence operations such as concatenation and slicing return

new lists instead of new strings when applied to lists. Because lists are mutable, however, they also support other operations that strings don't, such as deletion and index assignment operations, which change the lists in place.

Arrays of object references - Technically, Python lists contain zero or more references to other objects. Lists might thought of as arrays of pointers (addresses).

[ ]:
```
# Empty list
L = []
print(L)
```

```
[]
```

[ ]:
```
# Create list with 4 elements with indices 0:3
L = [123, 'abc', 1.23, {}]
print(L)
```

```
[123, 'abc', 1.23, {}]
```

[ ]:
```
# Nested Sublist
L = ['Bob', 40.0, ['dev', 'mgr']]
print(L)
```

```
['Bob', 40.0, ['dev', 'mgr']]
```

[ ]:
```
# List of values from -4 to 4
list(range(-4, 4))
```

[ ]:
```
[-4, -3, -2, -1, 0, 1, 2, 3]
```

[ ]:
```
# Subseting
L[0]
```

[ ]:
```
'Bob'
```

[ ]:
```
# Index of index
# The third element is a sublist and we extract the fist element from this␣
 ↪sublist
print(L[2][0])
# This should be a string
print(type(L[2][0]))
```

```
dev
<class 'str'>
```

[ ]:
```
# Slice to subset multiple elements
# Notice that the last integer indexed element is not extracted, so slice is␣
 ↪not inclusive
L[0:2]
```

[ ]: ['Bob', 40.0]

[ ]:
```
# Length is the number of elements
len(L)
```

[ ]: 3

[ ]:
```
# Concatenate, repeat
print(L * 2)
print(L + L + L)
```

```
['Bob', 40.0, ['dev', 'mgr'], 'Bob', 40.0, ['dev', 'mgr']]
['Bob', 40.0, ['dev', 'mgr'], 'Bob', 40.0, ['dev', 'mgr'], 'Bob', 40.0, ['dev',
'mgr']]
```

[ ]:
```
# Iteration
for var in L: print(var)
```

```
Bob
40.0
['dev', 'mgr']
```

[ ]:
```
# Membership (are these elements in the list?)
print(3 in L)
print("Bob" in L)
```

```
False
True
```

[ ]:
```
# Method grow
# A list is mutable, modifies in place, and so growing in Python will not be␣
 ↪memory inefficient like R
L.append(False)
print(L)
```

```
['Bob', 40.0, ['dev', 'mgr'], False]
```

[ ]:
```
# Method extend
L.extend([5, 6, 7])
print(L)
```

```
['Bob', 40.0, ['dev', 'mgr'], False, 5, 6, 7]
```

[ ]:
```
# Method insert
# Insert string object as the fifth element
L.insert(4, "Ken")
print(L)
```

```
['Bob', 40.0, ['dev', 'mgr'], False, 'Ken', 5, 6, 7]
```

[ ]:
```
# Method searching
# Get the index of the element
# This is similar to match() in R
print(L)
print(L.index("Ken"))
print(L.index(40))
```

```
['Bob', 40.0, ['dev', 'mgr'], False, 'Ken', 5, 6, 7]
4
1
```

[ ]:
```
# Method count returns the number of elements with the specified value
L.count(False)
```

[ ]:
```
1
```

[ ]:
```
# Sort by descending order
# The default for sort is ascending, i.e., reverse=False
L_1 = [3, 4, 9, 3, 2, 24, 45, 13, 9]
L_1.sort(reverse=True)
print(L_1)
```

```
[45, 24, 13, 9, 9, 4, 3, 3, 2]
```

[ ]:
```
# Reverse the list
L_2 = [3, 4, 5, 7]
L_2.reverse()
print(L_2)
```

```
[7, 5, 4, 3]
```

[ ]:
```
# Copy creates a copy of the list
L_new = L.copy()
# Copy on modify
L_new[2] = 9
# Check new modified copy
print(L_new)
# Check original object (should be unchanged)
print(L)
```

```
['Bob', 40.0, 9, False, 'Ken', 5, 6, 7]
['Bob', 40.0, ['dev', 'mgr'], False, 'Ken', 5, 6, 7]
```

[ ]:
```
# Before clear
print(L_1)
print(L_2)
```

```
# Clear all elements
L_1.clear()
L_2.clear()
# Now L_1 and L_2 should be empty lists
print(L_1)
print(L_2)
```

```
[45, 24, 13, 9, 9, 4, 3, 3, 2]
[7, 5, 4, 3]
[]
[]
```

[ ]:
```
# Pop removes element at the specified position
print(L)
# Remove the third element with index 2
L.pop(2)
# Examine the list after the removal
print(L)
```

```
['Bob', 40.0, ['dev', 'mgr'], False, 'Ken', 5, 6, 7]
['Bob', 40.0, False, 'Ken', 5, 6, 7]
```

[ ]:
```
# Another way to remove by position
print(L)
# Remove the first element using del
del L[0]
print(L)
```

```
['Bob', 40.0, False, 'Ken', 5, 6, 7]
[40.0, False, 'Ken', 5, 6, 7]
```

[ ]:
```
# Remove by name
print(L)
# Remove by value name
L.remove(False)
print(L)
```

```
[40.0, False, 'Ken', 5, 6, 7]
[40.0, 'Ken', 5, 6, 7]
```

[ ]:
```
# Remove slices
print(L_new)
# Remove the first three elements of the list, 0, 1, 2 not including 3
del L_new[0:3]
print(L_new)
```

```
['Bob', 40.0, 9, False, 'Ken', 5, 6, 7]
[False, 'Ken', 5, 6, 7]
```

[ ]:
```
# Subset and assignment
# This is similar to list[1:3] <- NULL in R
L = [3, "ken", [2, "3"], True]
print(L)
# Remove the first two elements 0 and 1
# The element indexed by 2 is not included
L[0:2] = []
print(L)
```

```
[3, 'ken', [2, '3'], True]
[[2, '3'], True]
```

[ ]:
```
# Subset one specific element and assign
print(L)
L[1] = 3
print(L)
```

```
[[2, '3'], True]
[[2, '3'], 3]
```

[ ]:
```
# Subset a slice and assign
L = list(range(-4, 5))
print(L)
# Recall that the last indexed element is not included and the length of the␣
 ↪slice L[3:7] is 4
L[3:7] = ["Ken", "needs", "a", "job"]
print(L)
```

```
[-4, -3, -2, -1, 0, 1, 2, 3, 4]
[-4, -3, -2, 'Ken', 'needs', 'a', 'job', 'now', 3, 4]
```

1.5.2 Why Slices and Range Exclude the Last Item?

The Pythonic convention of excluding the last item in slices and ranges works well with the zero-based indexing. Some convenient features are as follows:

- It's easy to see the length of a slice or range when only the stop position is given: `range(3)` and `my_list[:3]` both produce three items. This is not possible in R.
- It's easy to compute the length of a slice or range when start and stop are given: just subtract stop - start. For instance, `my_list[4:15]` would returns a slice with `15 - 4 = 11` elements. Whereas in R, we need to think a bit about it and `my_list[4:15]` would return a list with `15 - 4 + 1 = 12` since the last item is included.
- It's easy to split a sequence in two parts at any index x, without overlapping: simply get `my_list[:x]` and `my_list[x:]`

[ ]:
```
# Create a list
L = [10, 20, 30, 40, 50, 60]
```

```
print(L)
# Split at 2
# Or end at 2
print(L[:2])
# Split at 2
# Or start at 2
print(L[2:])
```

```
[10, 20, 30, 40, 50, 60]
[10, 20]
[30, 40, 50, 60]
```

1.5.3 List Comprehension

[ ]:
```
# Create a list
L = [10, 20, 30, 40, 50, 60]
# List comprehensions basic syntax
print([num for num in L])
# This essentially just prints the list itself
print([num for num in L] == L)
```

[ ]:
```
# We can now do set operations
# Create a smaller set P
P = [2, 20, 50, 4]
# Asymmetric difference between L and a smaller set P
[num for num in L if num not in P]
```

1.5.4 Dictionary

If we think of lists as ordered collections of objects, we can think of dictionaries as unordered collections; the chief distinction is that in dictionaries, items are *stored and fetched by key, instead of by positional offset*. The properties of dictionaries are as follows:

Accessed by key, not offset position - Dictionaries are sometimes called associative arrays or hashes. They associate a set of values with keys, so you can fetch an item out of a dictionary using the key under which you originally stored it. You use the same indexing operation to get components in a dictionary as you do in a list, but the index takes the form of a key, not a relative offset.

Unordered collections of arbitrary objects - Unlike in a list, items stored in a dictionary aren't kept in any particular order; in fact, Python pseudo-randomizes their left-to-right order to provide quick lookup. Keys provide the symbolic (not physical) locations of items in a dictionary. CPython 3.6 started preserving the insertion order of the keys as an implementation detail, and Guido van Rossum declared it an offical language feature in Python 3.7, so we can depend on the insertion order of a dictionary now.

Variable-length, heterogeneous, and arbitrarily nestable - Like lists, dictionaries can grow and shrink in place (without new copies being made), they can contain objects of any type, and they support nesting to any depth (they can contain lists, other dictionaries, and so on). Each key

can have just one associated value, but that value can be a collection of multiple objects if needed, and a given value can be stored under any number of keys.

Of the category "mutable mapping" - You can change dictionaries in place by assigning to indexes (they are mutable), but they don't support the sequence operations that work on strings and lists. Because dictionaries are unordered collections, operations that depend on a fixed positional order (e.g., concatenation, slicing) don't make sense. Instead, diction- aries are the only built-in, core type representatives of the mapping category— objects that map keys to values.

Tables of object references (hash tables) - If lists are arrays of object references that support access by position, *dictionaries are unordered tables of object references that support access by key.* Internally, dictionaries are implemented as hash tables (data structures that support very fast retrieval), which start small and grow on demand. Moreover, Python employs optimized hashing algorithms to find keys, so retrieval is quick. Like lists, dictionaries store object references (not copies, unless you ask for them explicitly).

[ ]:
```
# Empty dictionary
D = {}
print(D)
```

```
{}
```

[ ]:
```
# Two item (key-value pairs) dictionary
D = {'name': 'Bob', 'age': 40}
print(D)
```

```
{'name': 'Bob', 'age': 40}
```

[ ]:
```
# Nesting where a key is associated with a value that is another dictionary
E = {'cto': {'name': 'Bob', 'age': 40}}
print(E)
```

```
{'cto': {'name': 'Bob', 'age': 40}}
```

[ ]:
```
# Alternative ways to create dictionary
D = dict(name='Bob', age=40)
print(D)
```

[ ]:
```
# Using tuples
print(type(('name', 'Bob')))
# Create dictionary
D = dict([('name', 'Bob'), ('age', 40)])
print(D)
```

```
<class 'tuple'>
{'name': 'Bob', 'age': 40}
```

[ ]:
```
# Using lists
D = dict(zip(["Ken", "Wu"], [True, 3]))
```

```
print(D)
```

```
{'Ken': True, 'Wu': 3}
```

[ ]:
```
# Keys and Values
keys = {'a', 'e', 'i', 'o', 'u' }
value = [1]
# Using dictionary method
D = dict.fromkeys(keys, value)
print(D)
# Updating the value
value.append(7)
print(D)
```

```
{'o': [1], 'i': [1], 'e': [1], 'u': [1], 'a': [1]}
{'o': [1, 7], 'i': [1, 7], 'e': [1, 7], 'u': [1, 7], 'a': [1, 7]}
```

[ ]:
```
# Create a dictionary
D = {'name': 'Bob', 'age': 40}
# Index by key
D['name']
```

[ ]:
```
'Bob'
```

[ ]:
```
# Dictionary
print(E)
# Index of index
# The value associated with "cto" is another dictionary, and we want to select␣
 ↪the value with the key "age"
E['cto']['age']
```

```
{'cto': {'name': 'Bob', 'age': 40}}
```

[ ]:
```
40
```

[ ]:
```
# Create a dictionary
temp_D = dict(zip(['firstKey', 'secondKey', 'thirdKey'], [3, 5, False]))
print(temp_D)
# Index multiple key-value pairs
keys = ['firstKey', 'secondKey', 'thirdKey']
for key in keys:
    print(temp_D[key])
```

```
{'firstKey': 3, 'secondKey': 5, 'thirdKey': False}
3
5
False
```

[ ]:
```
# Dictionary
print(D)
# Membership (Is the key present in the dictionary?)
'age' in D
```

```
{'Physics': 67, 'Maths': 87, 'Practical': 48}
```

[ ]:
```
False
```

[ ]:
```
# Method show all keys
D.keys()
```

[ ]:
```
dict_keys(['Physics', 'Maths', 'Practical'])
```

[ ]:
```
# Method show all values
D.values()
```

[ ]:
```
dict_values([67, 87, 48])
```

[ ]:
```
# Method show all key-value tuples
# This is a dictionary item object
D.items()
```

[ ]:
```
dict_items([('Physics', 67), ('Maths', 87), ('Practical', 48)])
```

[ ]:
```
# Method create a copy
New_D = D.copy()
# Copy on modify
New_D['name'] = 43
# New dictionary
print(New_D)
# The original object should remain unchanged
print(D)
```

```
{'Physics': 67, 'Maths': 87, 'Practical': 48, 'name': 43}
{'Physics': 67, 'Maths': 87, 'Practical': 48}
```

[ ]:
```
# Method remove all items
New_D.clear()
print(New_D)
```

```
{}
```

[ ]:
```
# Method merge by key
D = {'Physics': 67, 'Maths': 87}
print(D)
D2 = {'Practical': 48}
print(D2)
```

```
# Update
D.update(D2)
print(D)
```

```
{'Physics': 67, 'Maths': 87}
{'Practical': 48}
{'Physics': 67, 'Maths': 87, 'Practical': 48}
```

[ ]:
```
# The get() method returns the value for the specified key if the key is in the␣
 ↪dictionary
print(D)
print(D.get('Practical'))
```

```
{'Physics': 67, 'Maths': 87, 'Practical': 48}
48
```

[ ]:
```
# The get() method returns a default value if the key is missing
# Unlike dict[key], a KeyError is raised when trying to get a missing key
print(D.get('Hi', "Key does not exist"))
```

```
Key does not exist
```

[ ]:
```
# Method remove by key, if absent the the default value is returned (or error␣
 ↪if no default is specified)
D.pop('Hi', "No key to delete")
```

[ ]:
```
'No key to delete'
```

[ ]:
```
# Dictionary
print(D)
# The setdefault() method returns the value of a key (if the key is in␣
 ↪dictionary)
# If not, it inserts key with a value to the dictionary
D.setdefault('Yo', True)
print(D)
```

```
{'Physics': 67, 'Maths': 87, 'Practical': 48}
{'Physics': 67, 'Maths': 87, 'Practical': 48, 'Yo': True}
```

[ ]:
```
# The Python popitem() method removes and returns the last element (key, value)␣
 ↪pair inserted into the dictionary
# Before Python 3.7, the popitem() method returned and removed an arbitrary␣
 ↪element (key, value) pair from the dictionary
# This should remove the "Yo":True pair
D.popitem()
print(D)
```

```
{'Physics': 67, 'Maths': 87, 'Practical': 48}
```

[ ]:
```
# Length, which is the number of stroed entries
len(D)
```

[ ]: 3

[ ]:
```
# Dictionary
print(D)
# Adding changing keys
D['Physics'] = 42
D['New'] = "Ken"
print(D)
```

```
{'Physics': 42, 'Maths': 87, 'Practical': 48}
{'Physics': 42, 'Maths': 87, 'Practical': 48, 'New': 'Ken'}
```

[ ]:
```
# Dictionary
print(D)
# Changing multiple keys
keys = ['Physics', 'YOLO', 'R']
values = [100, False, "studio"]
# Adding and changing multiple keys
for (key, value) in zip(keys, values):
    D[key] = value
# Check new dictionary
print(D)
```

```
{'Physics': 42, 'Maths': 87, 'Practical': 48, 'New': 'Ken'}
{'Physics': 100, 'Maths': 87, 'Practical': 48, 'New': 'Ken', 'YOLO': False, 'R':
'studio'}
```

[ ]:
```
# Dictionary
print(D)
# Deleting entries by key
del D['Maths']
print(D)
```

```
{'Physics': 100, 'Maths': 87, 'Practical': 48, 'New': 'Ken', 'YOLO': False, 'R':
'studio'}
{'Physics': 100, 'Practical': 48, 'New': 'Ken', 'YOLO': False, 'R': 'studio'}
```

[ ]:
```
# Dictionary views
print(list(D.keys()))
print(list(D.values()))
print(list(D.items()))
```

```
['Physics', 'Practical', 'New', 'YOLO', 'R']
[100, 48, 'Ken', False, 'studio']
```

```
[('Physics', 100), ('Practical', 48), ('New', 'Ken'), ('YOLO', False), ('R',
'studio')]
```

[ ]:
```
# Comprehensions
D = {x: x ** 4 for x in range(1, 5)}
# Take 1 and pair it with the value 1 raised to the power 4
# Take 2 and pair it with the value 2 raised to the power 4
# Take 3 and pair it with the value 3 raised to the power 4
# Take 4 and pair it with the value 4 raised to the power 4
print(D)
```

```
{1: 1, 2: 16, 3: 81, 4: 256}
```

[ ]:
```
# Loop over any iterable
D = {c: c * 4 for c in 'SPAM'}
# Take c in 'SPAM', which is the letter S, and pair it with the value S␣
 ↪concatenated 4times
# Take c in 'SPAM', which is the letter P, and pair it with the value P␣
 ↪concatenated 4 times
# Take c in 'SPAM', which is the letter A, and pair it with the value A␣
 ↪concatenated 4 times
# Take c in 'SPAM', which is the letter M, and pair it with the value M␣
 ↪concatenated 4 times
print(D)
```

```
{'S': 'SSSS', 'P': 'PPPP', 'A': 'AAAA', 'M': 'MMMM'}
```

[ ]:
```
# Dictionary comprehensions can be useful when initializing a dictionary
D = {x:0 for x in ['a', 'b', 'c']}
# Each key in ['a', 'b', 'c'] is initialized with the value 0
print(D)
```

```
{'a': 0, 'b': 0, 'c': 0}
```

[ ]:
```
# Initialize with None
D = {x: None for x in 'spam'}
print(D)
```

```
{'s': None, 'p': None, 'a': None, 'm': None}
```

1.5.5 Tuples

Tuples construct simple groups of objects. They work exactly like lists, except that tuples can't be changed in place (they're immutable) and are usually written as a series of items in parentheses, not square brackets. Their properties are:

Ordered collections of arbitrary objects - Like strings and lists, tuples are *positionally ordered* collections of objects (i.e., they maintain a left-to-right order among their contents); like lists, they can embed any kind of object.

Accessed by offset - Like strings and lists, items in a tuple are accessed by offset (not by key); they support all the offset-based access operations, such as indexing and slicing.

Of the category "immutable sequence" - Like strings and lists, tuples are sequences; they support many of the same operations. However, like strings, tuples are immutable; they don't support any of the in-place change operations applied to lists.

Fixed-length, heterogeneous, and arbitrarily nestable - Because tuples are immutable, you cannot change the size of a tuple without making a copy. On the other hand, tuples can hold any type of object, including other compound objects (e.g., lists, dictionaries, other tuples), and so they support arbitrary nesting.

Arrays of object references - Like lists, tuples are best thought of as object reference arrays; tuples store access points to other objects (references), and indexing a tuple is relatively quick.

[ ]:
```
# Empty Tuple
T = ()
print(T)
```

```
()
```

[ ]:
```
# One item tuple
# To create a one-item tuple, add a comma after the item, or else Python will␣
 ↪not recognize the variable as a tuple
T = (0,)
print(T)
len(T)
```

```
(0,)
```

[ ]:
```
1
```

[ ]:
```
# Create a tuple
T = (0, 'Ni', 1.2, 3)
print(T)
# Alternative way to create a tuple
Z = 0, 'Ken', 1.2, 3
print(Z)
```

```
(0, 'Ni', 1.2, 3)
(0, 'Ken', 1.2, 3)
```

[ ]:
```
# Nested Tuple
T = ('Bob', ('dev', 'mgr'))
print(T)
```

```
('Bob', ('dev', 'mgr'))
```

[ ]:
```
# Tuple of items in an iterable
T = tuple('spam')
print(T)
```

```
('s', 'p', 'a', 'm')
```

[ ]:
```
# Creating a tuple from a list
t2 = tuple([1, 4, 6])
print('t2 =', t2)
# Creating a tuple from a string
t1 = tuple('Python')
print('t1 =',t1)
# Creating a tuple from a dictionary
t1 = tuple({1: 'one', 2: 'two'})
print('t1 =',t1)
```

```
t2 = (1, 4, 6)
t1 = ('P', 'y', 't', 'h', 'o', 'n')
t1 = (1, 2)
```

[ ]:
```
# Index
print(T)
print(T[2])
```

```
('s', 'p', 'a', 'm')
a
```

[ ]:
```
# Index of index
T = ('Bob', ('dev', 'mgr'))
print(T)
# This should be the str 'dev'
print(T[1][0])
```

```
('Bob', ('dev', 'mgr'))
dev
```

[ ]:
```
# Slicing
t1 = tuple('Python')
print(t1[2:5])
```

```
('t', 'h', 'o')
```

[ ]:
```
# Tuple
print(t1)
# More slicing
# Start from right and move to left, skip two elements at a time
t1[::3]
```

```
('P', 'y', 't', 'h', 'o', 'n')
```

[ ]:
```
('P', 'h')
```

[ ]:
```
# Tuple
print(t1)
# More slicing
# Start from left and move to right, skip 1 element at a time
t1[::-2]
```

```
('P', 'y', 't', 'h', 'o', 'n')
```

[ ]:
```
('n', 'h', 'y')
```

[ ]:
```
# Tuple
print(t1)
# More slicing
# Index backwards
t1[::-1]
```

```
('P', 'y', 't', 'h', 'o', 'n')
```

[ ]:
```
('n', 'o', 'h', 't', 'y', 'P')
```

[ ]:
```
# Length
len(T)
```

[ ]:
```
2
```

[ ]:
```
# Concatenate
print(t1 + t1[::-1])
# Repeat
print(t1 * 3)
```

```
('P', 'y', 't', 'h', 'o', 'n', 'n', 'o', 'h', 't', 'y', 'P')
('P', 'y', 't', 'h', 'o', 'n', 'P', 'y', 't', 'h', 'o', 'n', 'P', 'y', 't', 'h',
'o', 'n')
```

[ ]:
```
# Iteration
for x in T: print(x)
```

```
Bob
('dev', 'mgr')
```

[ ]:
```
# Membership
print('spam' in T)
print('Bob' in T)
```

```
False
True
```

[ ]:
```python
# Create a tuple
# The numbers 2 and 10 are not inclusive
T = tuple(range(2, 10))
print(T)
# Comprehension
# For each element in T, raise it to the power 2
[x ** 2 for x in T]
```

```
(2, 3, 4, 5, 6, 7, 8, 9)
```

[ ]:
```
[4, 9, 16, 25, 36, 49, 64, 81]
```

[ ]:
```python
# Create tuple
T = (3, 4, 2, 2, 4, 2, 5, 6, 1, 10, 2)
print(T)
# Search
print(T.index(4))
# Count
print(T.count(2))
```

```
(3, 4, 2, 2, 4, 2, 5, 6, 1, 10, 2)
1
4
```

Python's `namedtuple()` is a factory function available in `collections`. It allows us to create tuple subclasses with named fields. We can access the values in a given named tuple using the dot notation and the field names, like in obj.attr. In general, we can use `namedtuple` instances wherever we need a tuple-like object. Named tuples have the advantage that they provide a way to access their values using field names and the dot notation. This will make our code more Pythonic.

[ ]:
```python
from collections import namedtuple
# Create a namedtuple type, Point
Point = namedtuple("Point", "x y")
print(issubclass(Point, tuple))
# Instantiate the new type
point = Point(2, 4)
print(point)
# Dot notation to access coordinates
print(point.x)
print(point.y)
```

```
True
Point(x=2, y=4)
2
4
```

The phrase "instantiating a class" means the same thing as "creating an object." When you create an object, you are creating an "instance" of a class, therefore "instantiating" a class.

[ ]:
```
# Make a generated class
Rec = namedtuple('Rec', ['name', 'age', 'jobs'])
# Create a named tuple
bob = Rec(name='Bob', age=40.5, jobs=['dev', 'mgr'])
print(bob)
# Access by attribute
print(bob.age)
print(bob.jobs)
```

```
Rec(name='Bob', age=40.5, jobs=['dev', 'mgr'])
40.5
['dev', 'mgr']
```

The +, *, and slicing operations return new tuples when applied to tuples, and that tuples don't provide the same methods as those for strings, lists, and dictionaries. If we want to sort a tuple, for example, we'll usually have to either first convert it to a list to gain access to a sorting method call and make it a mutable object, or use the newer sorted built-in that accepts any sequence object:

[ ]:
```
# Create a tuple
T = ('cc', 'aa', 'dd', 'bb')
# Convert to list
tmp = list(T)
# Sort list
tmp.sort()
# Examine
print(tmp)
# Convert back to tuple
T = tuple(tmp)
print(T)
```

```
['aa', 'bb', 'cc', 'dd']
('aa', 'bb', 'cc', 'dd')
```

[ ]:
```
print(type(T))
# Use built-in function
sorted(T)
```

```
<class 'tuple'>
```

[ ]:
```
['aa', 'bb', 'cc', 'dd']
```

1.5.6 The relative immutability of tuples

Tuples, like most Python collections—lists, dicts, sets, etc.—are containers: they hold references to objects. If the referenced items are mutable, they may change even if the tuple itself does not.

In other words, the immutability of tuples really refers to the *physical contents of the tuple data structure (i.e., the references it holds), and does not extend to the referenced objects.*

[ ]:
```
# The tuple t1 is immutable, but t1[-1] or t1[2] is mutable
# Negative indexing simple means extract from left to right
t1 = (1, 2, [30, 40])
t2 = (1, 2, [30, 40])
# The two tuples are equal
print(t1==t2)
# The id() function returns identity (unique integer) of an object
print(id(t1[-1]))
# Modify the list element in place
t1[-1].append(99)
print(t1)
# The identity of the immutable tuple has now changed
print(id(t1[-1]))
# The mutable object it references has changed
print(t1==t2)
```

```
True
140353022685256
(1, 2, [30, 40, 99])
140353022685256
False
```

The relative immutability of tuples can be demonstrated diagrammatically: