Programming with Python

Data Structures in Python

Kuo, Yao-Jen yaojenkuo@datainpoint.com (mailto:yaojenkuo@datainpoint.com) from DATAINPOINT (https://www.datainpoint.com/)

TL; DR

In this lecture, we will talk about the built-in data structures in Python.

The What and Why

What is a data structure?

> *In computer science, a data structure is a data organization, management, and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.*

Source: [https://en.wikipedia.org/wiki/Data_structure (https://en.wikipedia.org/wiki/Data_structure)](https://en.wikipedia.org/wiki/Data_structure)

Why data structure?

As a software engineer, the main job is to perform operations on data, we can simplify that operation into:

1. Take some input
2. Process it
3. Return the output

Quite similar to what we've got from the definition of a function.

To make the process efficient, we need to optimize it via data structure

Data structure decides how and where we put the data to be processed. A good choice of data structure can enhance our efficiency.

We will talk about 4 built-in data structures in Python

- `list`
- `tuple`
- `dict` as in dictionary
- `set`

Built-in data structures refer to those need no self-definition or importing

Quite similar to the comparison of built-in functions vs. self-defined/third party functions.

Built-in Data Structure: `list`

Lists

Lists are the basic ordered and mutable data collection type in Python. They can be defined with comma-separated values between square brackets.

In [1]:
```
primes = [2, 3, 5, 7, 11]
print(type(primes)) # use type() to check type
print(len(primes))  # use len() to check how many elements are stored in the list
```

```
<class 'list'>
5
```

Lists have a number of useful methods

- `.append()`
- `.pop()`
- `.remove()`
- `.insert()`
- `.sort()`
- ...etc.

We can use `TAB` and `SHIFT - TAB` for documentation prompts in a notebook environment.

In [2]:

```
primes.append(13) # appending an element to the end of a list
print(primes)
primes.pop() # popping out the last element of a list
print(primes)
primes.remove(2) # removing the first occurance of an element within a list
print(primes)
primes.insert(0, 2) # inserting certain element at a specific index
print(primes)
primes.sort(reverse=True) # sorting a list, reverse=False => ascending order; reve
rse=True => descending order
print(primes)
```

```
[2, 3, 5, 7, 11, 13]
[2, 3, 5, 7, 11]
[3, 5, 7, 11]
[2, 3, 5, 7, 11]
[11, 7, 5, 3, 2]
```

Python provides access to elements in compound types through

- **indexing** for a single element
- **slicing** for multiple elements

Python uses zero-based indexing

In [3]:
```
primes.sort()
print(primes[0]) # the first element
print(primes[1]) # the second element
```

```
2
3
```

Elements at the end of the list can be accessed with negative numbers, starting from -1

In [4]:
```
print(primes[-1]) # the last element
print(primes[-2]) # the second last element
```

```
11
7
```

While indexing means fetching a single value from the list, slicing means accessing multiple values in sub-lists

- start(inclusive)
- stop(non-inclusive)
- step

```
# slicing syntax
OUR_LIST[start:stop:step]
```

In [5]:
```
print(primes[0:3:1]) # slicing the first 3 elements
print(primes[-3:len(primes):1]) # slicing the last 3 elements
print(primes[0:len(primes):2]) # slicing every second element
```

```
[2, 3, 5]
[5, 7, 11]
[2, 5, 11]
```

If leaving out, it defaults to

- start: 0
- stop: -1
- step: 1

So we can do the same slicing with defaults

In [6]:

```
print(primes[:3]) # slicing the first 3 elements
print(primes[-3:]) # slicing the last 3 elements
print(primes[::2]) # slicing every second element
print(primes[::-1]) # a particularly useful tip is to specify a negative step
```

```
[2, 3, 5]
[5, 7, 11]
[2, 5, 11]
[11, 7, 5, 3, 2]
```

Built-in Data Structure: `tuple`

Tuples

Tuples are in many ways similar to lists, but they are defined with parentheses rather than square brackets.

In [7]:

```
primes = (2, 3, 5, 7, 11)
print(type(primes)) # use type() to check type
print(len(primes))  # use len() to check how many elements are stored in the list
```

```
<class 'tuple'>
5
```

The main distinguishing feature of tuples is that they are immutable

Once they are created, their size and contents cannot be changed.

In [8]:

```
primes = [2, 3, 5, 7, 11]
primes[-1] = 13
print(primes)
primes = tuple(primes)
primes[-1] = 11
```

```
[2, 3, 5, 7, 13]

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-878e61bc04b4> in <module>
      3 print(primes)
      4 primes = tuple(primes)
----> 5 primes[-1] = 11

TypeError: 'tuple' object does not support item assignment
```

Use TAB to see if there is any mutable method for tuple

```
primes.<TAB>
```

Tuples are often used in a Python program; like functions that have multiple return values

In [9]:
```
def get_locale(country, city):
    return country, city

print(get_locale("Taiwan", "Taipei"))
print(type(get_locale("Taiwan", "Taipei")))
```

```
('Taiwan', 'Taipei')
<class 'tuple'>
```

Multiple return values can also be individually assigned

In [10]:
```
my_country, my_city = get_locale("Taiwan", "Taipei")
print(my_country)
print(my_city)
```

```
Taiwan
Taipei
```

Built-in Data Structure: `dict`

Dictionaries

Dictionaries are extremely flexible mappings of keys to values, and form the basis of much of Python's internal implementation. They can be created via a comma-separated list of `key:value` pairs within curly braces.

In [11]:
```
the_celtics = {
    'isNBAFranchise': True,
    'city': "Boston",
    'fullName': "Boston Celtics",
    'tricode': "BOS",
    'teamId': 1610612738,
    'nickname': "Celtics",
    'confName': "East",
    'divName': "Atlantic"
}

print(type(the_celtics))
print(len(the_celtics))
```

```
<class 'dict'>
8
```

Elements are accessed through valid key rather than zero-based order

In [12]:
```python
print(the_celtics['city'])
print(the_celtics['confName'])
print(the_celtics['divName'])
```

```
Boston
East
Atlantic
```

New key:value pair can be set smoothly

In [13]:
```
the_celtics['isMyFavorite'] = True
print(the_celtics)
```

```
{'isNBAFranchise': True, 'city': 'Boston', 'fullName': 'Boston Celtics', 'tric
ode': 'BOS', 'teamId': 1610612738, 'nickname': 'Celtics', 'confName': 'East',
'divName': 'Atlantic', 'isMyFavorite': True}
```

Use `del` to remove a key:value pair from a dictionary

In [14]:
```
del the_celtics['isMyFavorite']
print(the_celtics)
```

```
{'isNBAFranchise': True, 'city': 'Boston', 'fullName': 'Boston Celtics', 'tric
ode': 'BOS', 'teamId': 1610612738, 'nickname': 'Celtics', 'confName': 'East',
'divName': 'Atlantic'}
```

Common mehtods called on dictionaries

- `.keys()`
- `.values()`
- `.items()`

In [15]:
```
print(the_celtics.keys())
print(the_celtics.values())
print(the_celtics.items())
```

```
dict_keys(['isNBAFranchise', 'city', 'fullName', 'tricode', 'teamId', 'nicknam
e', 'confName', 'divName'])
dict_values([True, 'Boston', 'Boston Celtics', 'BOS', 1610612738, 'Celtics', '
East', 'Atlantic'])
dict_items([('isNBAFranchise', True), ('city', 'Boston'), ('fullName', 'Boston
Celtics'), ('tricode', 'BOS'), ('teamId', 1610612738), ('nickname', 'Celtics
'), ('confName', 'East'), ('divName', 'Atlantic')])
```

Built-in Data Structure: `set`

Sets

The fourth basic collection is the set, which contains unordered collections of unique items. They are defined much like lists and tuples, except they use the curly brackets.

In [16]:
```
primes = {2, 3, 5, 7, 11}
odds = {1, 3, 5, 7, 9}
print(type(primes))
print(len(odds))
```

```
<class 'set'>
5
```

Python’s sets have all of the operations like union, intersection, difference, and symmetric difference

Union: elements appearing in either sets

In [17]:
```
print(primes | odds)      # with an operator
print(primes.union(odds)) # equivalently with a method
```

```
{1, 2, 3, 5, 7, 9, 11}
{1, 2, 3, 5, 7, 9, 11}
```

Intersection: elements appearing in both

In [18]:
```
print(primes & odds)              # with an operator
print(primes.intersection(odds)) # equivalently with a method
```

```
{3, 5, 7}
{3, 5, 7}
```

Difference: elements in primes but not in odds

In [19]:
```
print(primes - odds)            # with an operator
print(primes.difference(odds)) # equivalently with a method
```

```
{2, 11}
{2, 11}
```

Symmetric difference: items appearing in only one set

In [20]:

```
print(sorted((primes - odds) | (odds - primes))) # union two differences
print(primes ^ odds)                              # with an operator
print(primes.symmetric_difference(odds))          # equivalently with a method
```

```
[1, 2, 9, 11]
{1, 2, 9, 11}
{1, 2, 9, 11}
```

One of the powerful features of Python's compound objects is that they can contain objects of any type, or even a mix of types

Take data.nba.net for example

The /10s/prod/v1/today.json (https://data.nba.net/10s/prod/v1/today.json) is a compound dictionary contained other dictionary as values.