We continue our tour of the Python Standard Library with Specialized Data Types.

These are data types that are not part of the core language, but are implemented within the Standard Library. Typically, these data types are special purpose (e.g. `datetime`) or implementations of structures found in other languages (e.g. heaps). Today, we will cover:

- `datetime`
- `collections`
- `copy`

There are other built-in specialized data types, but some (for example, `array`) are better defined by external libraries.
As the name suggests, the datetime module defines structures for storing and manipulating date and time information.

- Time information is represented with a time object.
- Date information is represented with a date object.
- Date and time information can be represented with a datetime object.
- Changes in date and/or time can be represented with a timedelta object.
- Time zones can be represented by classes derived from the tzinfo class.
- `datetime.MINYEAR` and `datetime.MAXYEAR` are defined for you.
• `datetime.time(hour, minute, second, microsecond, tzinfo)`

```python
>>> import datetime
>>> t = datetime.time(3, 15, 45)
>>> print t
03:15:45
>>> t.microsecond
0
```

• `datetime.time.min` is the earliest representable time object. Try `max` and `resolution`, too.

```python
>>> print datetime.time.min
00:00:00
>>> print datetime.time.max
23:59:59.999999
>>> print datetime.time.resolution
0:00:00.000001
```
• `datetime.date(year, month, day)`

```python
>>> import datetime
>>> d = datetime.date(2015, 1, 26)
>>> print d
2015-01-26
>>> d.year
2015
>>> d.month
1
>>> d.day
26
```
• We can initialize by other convenient methods.

```python
>>> import time
>>> import datetime
>>> d = datetime.date.fromtimestamp(time.time())
>>> d
datetime.date(2015, 1, 25)
>>> d = datetime.date.fromordinal(735624)
>>> d
datetime.date(2015, 1, 26)
>>> d.replace(year = 2016)
datetime.date(2016, 1, 26)
```

• Use `datetime.date.replace()` for convenient modification.

```python
>>> d.replace(year = 2016)
datetime.date(2016, 1, 26)
```
• `datetime.date` methods.

```python
>>> d = datetime.date(2015, 1, 26)
>>> d.toordinal()
735624
>>> d.weekday()  # 0 is Monday, 6 is Sunday
0
```
**DATETIME**

- `datetime.datetime(year, month, day, hour, minute, second, microsecond, tzinfo)`
  ```python
def main():
    >>> d = datetime.datetime(2015, 1, 26, 1, 25, 0)
    >>> print d
    2015-01-26 01:25:00
    
    >>> d = d.replace(hour = 2, minute = 15)
    >>> print d
    2015-01-26 02:15:00
  ```

- Has all the methods and attributes of `datetime.date` and `datetime.time` classes.
  ```python
def main():
    >>> d = d.replace(hour = 2, minute = 15)
    >>> print d
    2015-01-26 02:15:00
    >>> datetime.datetime.now()
    datetime.datetime(2015, 1, 25, 15, 9, 11, 68389)
    >>> d.toordinal()
    735624
    >>> d.weekday()
    0
  ```
DATE

• `datetime.timedelta()`
  ```
  >>> import datetime
  >>> d = datetime.datetime(2015, 1, 26, 1, 25)
  >>> print d
  2015-01-26 01:25:00
  >>> t = datetime.timedelta(weeks = 2, days = 3, seconds = 25)
  >>> print d+t
  2015-02-12 01:25:25
  >>> print d-t
  2015-01-09 01:24:35
  ```

• Compare objects using typical comparison operators.
  ```
  >>> d > d-t
  True
  >>> d < d-t
  False
  ```
Create a class derived from tzinfo to store timezone information (do not instantiate tzinfo itself!) – the methods to write are as follows:

- `tzinfo.utcoffset(self, dt)`: Return offset of local time from UTC, in minutes east of UTC.
- `tzinfo.dst(self, dt)`: Return the daylight saving time (DST) adjustment, in minutes east of UTC.
- `tzinfo.tzname(self, dt)`: Return the time zone name corresponding to the datetime object `dt`, as a string.
The simplest tzinfo-based class is the one that defines the UTC timezone.

```python
import datetime

class UTC(datetime.tzinfo):
    def utcoffset(self, dt):
        return datetime.timedelta(0)

    def tzname(self, dt):
        return "UTC"

    def dst(self, dt):
        return datetime.timedelta(0)
```
The collections module defines high-performance implementations of various data structures as an alternative to the built-in data types list, dict, set, and tuple. These data structures are:

- Counter
- deque
- OrderedDict
Counter, defined in collections, is a subclass of the dict class which counts hashable objects.

```python
>>> from collections import Counter
>>> c1 = Counter(['a', 'b', 'c', 'a', 'b', 'b'])
>>> print c1
Counter({'b': 3, 'a': 2, 'c': 1})
>>> c2 = Counter(a=2, b=3, c=1)
>>> print c2
Counter({'b': 3, 'a': 2, 'c': 1})
>>> c3 = Counter({'a':2, 'b':3, 'c': 1})
>>> print c3
Counter({'b': 3, 'a': 2, 'c': 1})
```
>>> print c1
Counter({'b': 3, 'a': 2, 'c': 1})
>>> c1.update('abcdabbd')
>>> print c1
Counter({'b': 6, 'a': 4, 'c': 2, 'd': 2})

>>> c1.most_common(2)
[('b', 6), ('a', 4)]

>>> for k in list(c1.keys()):
...     print k, ': ', c1[k]
...
 a :  4
c :  2
b :  6
d :  2
**DEQUE**

Defined in collections, deque provides a generalization for stacks and queues that can be initialized from iterables.

- `pop()`, `popleft()`
- `append()`, `appendleft()`
- `extend()`, `extendleft()`
- `clear()`, `count()`, `remove()`, `reverse()`

>>> from collections import deque
>>> d = deque('example')
>>> d
deque(['e', 'x', 'a', 'm', 'p', 'l', 'e'])

>>> d.append('s')
>>> d
deque(['e', 'x', 'a', 'm', 'p', 'l', 'e', 's'])

>>> d.pop()
's'

>>> d
deque(['e', 'x', 'a', 'm', 'p', 'l', 'e'])

>>> d[2]
'a'

>>> d.pop()
'e'

>>> d.extend('es')
>>> d
deque(['e', 'x', 'a', 'm', 'p', 'l', 'e', 's'])

>>> d.popleft()
'e'

>>> d
deque(['x', 'a', 'm', 'p', 'l', 'e', 's'])
OrderedDict objects are just like regular dict objects, except that they preserve order.

```python
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple':4, 'pear': 1, 'orange': 2}
>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])
>>> # dictionary sorted by value
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])
>>> # dictionary sorted by length of the key string
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2), ('banana', 3)])
```
**ORDERED DICT**

OrderedDict objects are subclasses of dict, but also support the following method:

```
OrderedDict.popitem(last = True) pops a key, value pair from the dictionary. The order if LIFO unless last is False.
```
Assignments in Python work by creating bindings between names (or targets) and objects. This can be dangerous when dealing with mutable objects, so copying may be necessary.

- `copy.copy(x)`
  Return a shallow copy of x.

- `copy.deepcopy(x)`
  Return a deep copy of x.

The difference becomes relevant when our objects contain other objects (lists of lists, for example). Only `deepcopy()` will recursively create copies of nested objects.
To illustrate the importance of `deepcopy()`, consider the following:

```python
>>> import copy
>>> a = [[1, 2, 3], ['one', 'two', 'three']]
>>> b = copy.copy(a)
>>> b[0][0] = 4

>>> a
[[4, 2, 3], ['one', 'two', 'three']]

>>> b = copy.deepcopy(a)

>>> b
[[4, 2, 3], ['one', 'two', 'three']]

>>> b[0][0] = 5

>>> b
[[5, 2, 3], ['one', 'two', 'three']]

>>> a
[[4, 2, 3], ['one', 'two', 'three']]```
Next, we will look at a number of numeric modules that are available in the Standard Library. It’s important to note that while these are available, there are external numeric and scientific libraries that tend to be more widely-used for actually scientific or mathematical applications.

- math
- cmath
- random
- itertools
The math module defines mathematical functions defined by the C standard.

```python
>>> import math
>>> math.ceil(4.6)
5.0
>>> math.ceil(4.3)
5.0
>>> math.fabs(-5.7)
5.7
>>> math.factorial(5)
120
>>> math.floor(8.7)
8.0
>>> math.frexp(9) # frexp(x) returns (m,e) s.t. x == m * 2**e
(0.5625, 4)
>>> math.fsum([1, 3, 5, 7])
16.0
```
Additional methods include:

- `math.isinf(x)`: True if x is infinity.
- `math.isnan(x)`: True if x is NaN (Not a Number).

```python
>>> math.exp(2)
7.38905609893065
>>> math.log(100, 10)
2.0
>>> math.pow(2, 3)
8.0
>>> math.sqrt(16)
4.0
```
Additional math methods:

• Trigonometric functions (x in radians): math.sin(x), math.cos(x), math.tan(x), etc…
• Degree conversions: math.radians(x) and math.degrees(x).
• Error functions: math.gamma(x), math.erf(x), …
• Constants: math.pi, math.e
The modulecmath provides mathematical functions for complex numbers.

```python
>>> import cmath
>>> c = complex(-1, 0)
>>> cmath.phase(c)  # calculate arg
3.141592653589793
>>> c = complex(1, 3)
>>> cmath.polar(c)  # cartesian → polar
(3.1622776601683795, 1.2490457723982544)
>>> cmath.rect(1, math.pi/2)  # polar → cartesian
(6.123031769111886e-17+1j)
```
Additionally, we have some of the same math functions defined for complex data types.

- `cmath.exp(c)`, `cmath.log(c)`, `cmath.sqrt(c)`
- `cmath.sin(c)`, `cmath.cos(c)`, etc...
- `cmath.isinf(c)`, `cmath.isnan(c)`
- `cmath.pi`, `cmath.e`
The random module provides methods for creating pseudo-random numbers.

```python
>>> random.random() # Random float x
0.37444887175646646
>>> random.uniform(1, 10) # Random float x
1.1800146073117523
>>> random.randint(1, 10) # Integer from 1 to 10
7
>>> random.randrange(0, 101, 2) # Even int from 0 to 100
26
>>> random.choice('abcdefghij') # Choose a random element 'c'
'c'
>>> items = [1, 2, 3, 4, 5, 6, 7]
>>> random.shuffle(items)
>>> items
[7, 3, 2, 5, 6, 4, 1]
>>> random.sample([1, 2, 3, 4, 5], 3) # Choose 3 elements
[4, 1, 5]
```
Additional methods:

• `random.seed([x])` initializes the random number generator. Current time is used otherwise.

• Various distributions methods, e.g. `random.gammavariate(alpha, beta)` is a gamma distribution with parameters alpha and beta.
The *itertools* module provides methods for manipulating iterables (typically, sequence) data types.

The methods defined in *itertools* are fast and memory efficient. The methods are meant to be building-blocks, which can be combined.
• `itertools.chain()` creates a single iterator over its arguments as if they were one iterable.

```python
>>> import itertools
>>> for i in itertools.chain([1, 2, 3], ['four', 'five', 'six'], '789):
...     print i
...
1
2
3
four
five
six
7
8
9
```
**ITERTOOLS**

- `itertools.izip()` returns an iterator over the zipped values of the arguments.

```python
>>> for i in itertools.izip([1, 2, 3], ['one', 'two', 'three']):
...     print i
...
(1, 'one')
(2, 'two')
(3, 'three')
```
ITERTOOLS

• `itertools.tee()` returns multiple independent iterators based on the argument (defaults to 2).

• Do not consume values from the original iterator – it will affect the new ones.

```python
>>> i1, i2 = itertools.tee(itertools.chain([1,2], [3,4]))
>>> for i in i1:
...     print i
...     print i
1
2
3
4
>>> for i in i2:
...     print i
...     print i
1
2
3
4
```
The `itertools.imap()` function works just like the `map()` function. It returns an iterator that calls a function on the values in the input iterators, and returns the results.

```python
>>> for i in itertools.imap(lambda x: 2*x, range(5)):
...     print i
... 0
... 2
... 4
... 6
... 8
```
• The `itertools.count(x=0)` function returns an iterator over consecutive integers, starting with `x` which defaults to 0.

```python
>>> for i in itertools.izip(itertools.count(), ['a', 'b', 'c', 'd', 'e', 'f', 'g']):
...     print i
...
(0, 'a')
(1, 'b')
(2, 'c')
(3, 'd')
(4, 'e')
(5, 'f')
(6, 'g')
```
ITERTOOLS

• The `itertools.cycle()` method returns an iterator that repeatedly cycles over the contents of its argument.

```python
>>> counter = 0
>>> for i in itertools.cycle(['a', 'b', 'c']):
...     print(i)
...     counter = counter + 1
...     if counter > 8:
...         break
...
a b c a b c a b c
```
Other useful itertools methods include:

- `itertools.ifilter()`, similar to `filter()`
- `itertools.repeat()`
- `itertools.groupby()` returns an iterator over sets of values grouped by a common key. For example, you can use it to group keys in a dictionary with the same value.
More of the Standard Library!