Built-in Collections

Austin Bingham
COFOUNDER - SIXTY NORTH

@austin_bingham

Robert Smallshire
COFOUNDER - SIXTY NORTH

@robsmallshire

Overview

Deeper look at `str`, `list`, and `dict`

New collection types:

- `tuple`
- `range`
- `set`

Protocols that unite collections

tuple

Immutable sequences of arbitrary objects

```
>>> a
4
>>> b
3
>>> c
2
>>> d
1
>>> a = 'jelly'
>>> b = 'bean'
>>> a, b = b, a
>>> a
'bean'
>>> b
'jelly'
>>> tuple([561, 1105, 1729, 2465])
(561, 1105, 1729, 2465)
>>> tuple("Carmichael")
('C', 'a', 'r', 'm', 'i', 'c', 'h', 'a', 'e', 'l')
>>> 5 in (3, 5, 17, 257, 65537)
True
>>> 5 not in (3, 5, 17, 257, 65537)
False
>>>
```

Tuple unpacking

Destructuring operation that unpacks data structures into named references

Strings

```
>>> len("llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch")
58
>>> "New" + "found" + "land"
'Newfoundland'
>>> s = "New"
>>> s += "found"
>>> s += "land"
>>> s
'Newfoundland'
>>>
```

Strings are immutable. You can not modify them in place.

Use `str.join()` to Join Strings

1. **Concatenation with + results in temporaries**
2. **`str.join()` inserts a separator between a collection of strings**
3. **Call `join()` on the *separator string***

```
>>> colors = ';'.join(['#45ff23', '#2321fa', '#1298a3', '#a32312'])
>>> colors
'#45ff23;#2321fa;#1298a3;#a32312'
>>> colors.split(';')
['#45ff23', '#2321fa', '#1298a3', '#a32312']
>>> ''.join(['high', 'way', 'man'])
'highwayman'
>>>
```

Moment of Zen

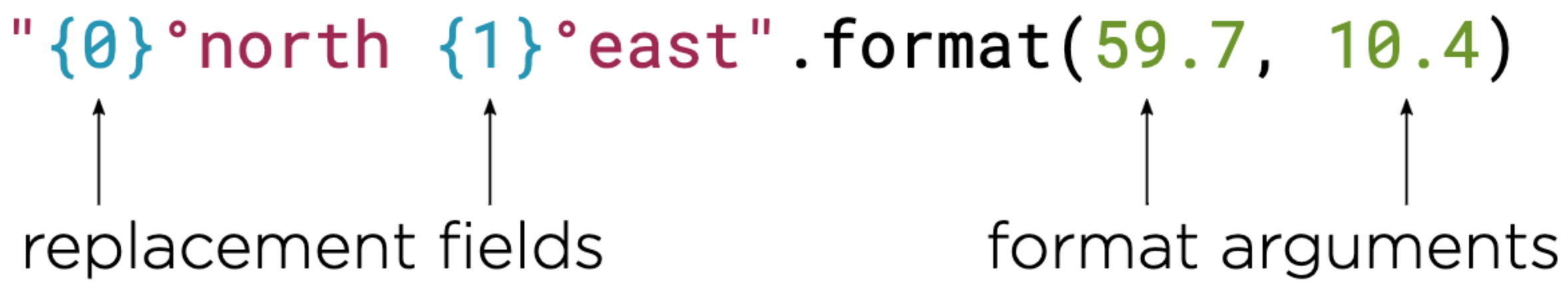
The way may not be obvious at first

To concatenate
Invoke join on empty text
Something from nothing

```
>>> "unforgetable".partition('forget')
('un', 'forget', 'able')
>>> departure, separator, arrival = "London:Edinburgh".partition(':')
>>> departure
'London'
>>> arrival
'Edinburgh'
>>> origin, _, destination = "Seattle-Boston".partition('-')
>>>
```

String formatting

String Formatting



```
>>> "The age of {0} is {1}".format('Jim', 32)
'The age of Jim is 32'
>>> "The age of {0} is {1}. {0}'s birthday is on {2}".format('Fred', 24, 'Octobe
r 31')
"The age of Fred is 24. Fred's birthday is on October 31"
>>> "Reticulating spline {} of {}.".format(4, 23)
'Reticulating spline 4 of 23.'
>>> "Current position {latitude} {longitude}".format(latitude="60N", longitude="
5E")
'Current position 60N 5E'
>>> "Galactic position x={pos[0]}, y={pos[1]}, z={pos[2]}".format(pos=(65.2, 23.
1, 82.2))
'Galactic position x=65.2, y=23.1, z=82.2'
>>> import math
>>> "Math constants: pi={m.pi}, e={m.e}".format(m=math)
'Math constants: pi=3.141592653589793, e=2.718281828459045'
>>> "Math constants: pi={m.pi:.3f}, e={m.e:.3f}".format(m=math)
'Math constants: pi=3.142, e=2.718'
>>> value = 4 * 20
>>> 'The value is {value}'.format(value=value)
'The value is 80'
>>>
```

PEP 498: Literal String Interpolation

Commonly called f-strings

"Embed expressions inside literal strings, using a minimal syntax"

F-string Syntax

```
f"one plus one is {1 + 1}"
```

Evaluated and inserted at runtime

```
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
:
```

Range

Sequence representing an arithmetic progression of integers

```
>>> range(5)
range(0, 5)
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>> range(5, 10)
range(5, 10)
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(10, 15))
[10, 11, 12, 13, 14]
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
>>>
```

`range()` Signature

```
range(stop)
```

```
range(start, stop)
```

```
range(start, stop, step)
```

Range does not support keyword arguments

```
>>> s = [0, 1, 4, 6, 13]
>>> for i in range(len(s)):
...     print(s[i])
...
0
1
4
6
13
>>> s = [0, 1, 4, 6, 13]
>>> for v in s:
...     print(v)
...
0
1
4
6
13
>>>
```

enumerate

Constructs an iterable of `(index, value)` tuples around another iterable object

```
>>> t = [6, 372, 8862, 148800, 2096886]
>>> for p in enumerate(t):
...     print(p)
...
(0, 6)
(1, 372)
(2, 8862)
(3, 148800)
(4, 2096886)
>>> for i, v in enumerate(t):
...     print(f"i = {i}, v = {v}")
...
i = 0, v = 6
i = 1, v = 372
i = 2, v = 8862
i = 3, v = 148800
i = 4, v = 2096886
>>>
```

Lists

Negative indices

Index from the end of sequences using negative numbers.

The last element is at index -1.

```
>>> r = [1, -4, 10, -16, 15]
>>> r[-1]
15
>>> r[-2]
-16
>>> r[len(r) - 1]
15
>>> r[0]
1
>>> r[-0]
1
>>>
```

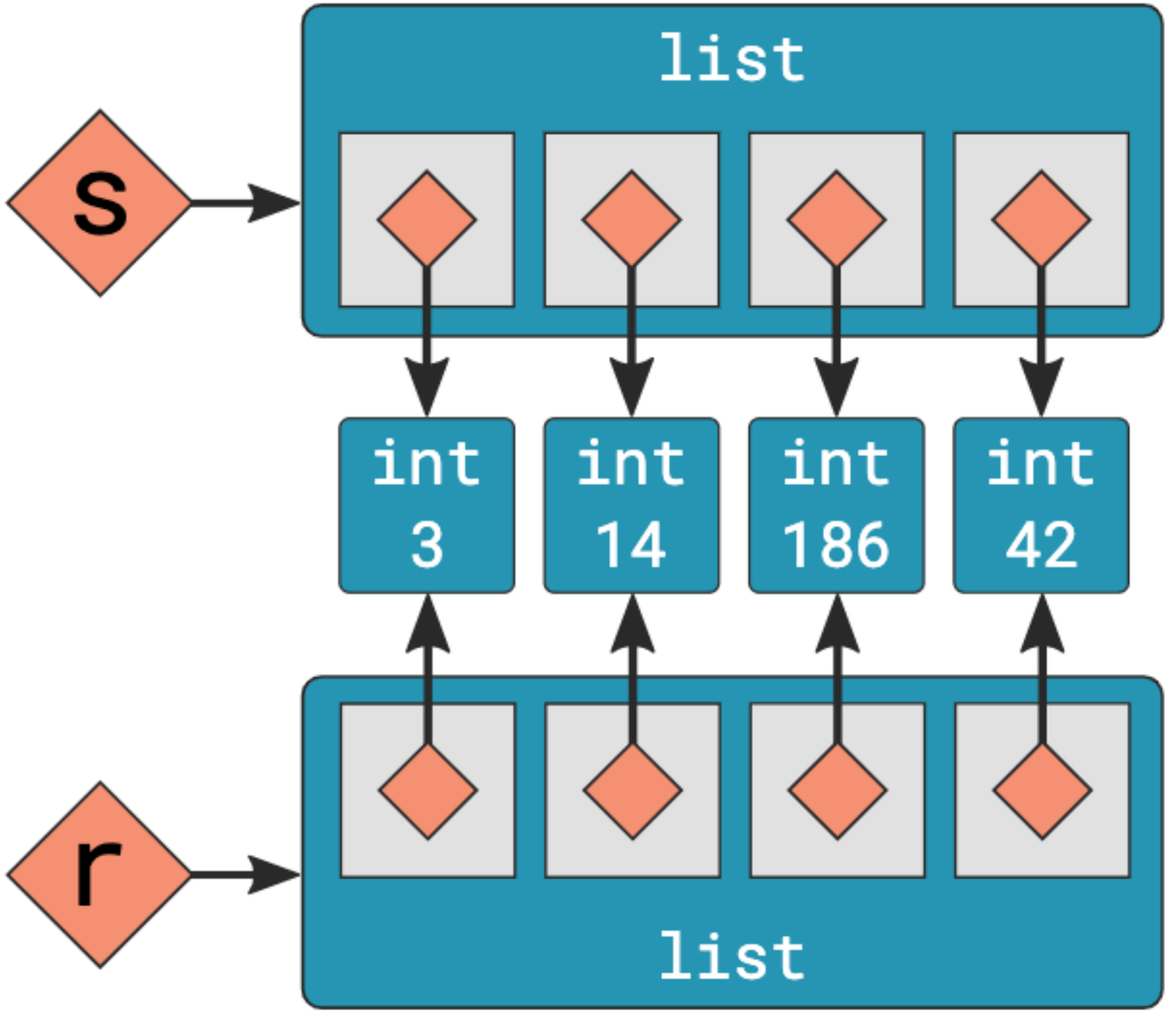
Slicing

Extended form of indexing for referring to a portion of a list or other sequence.

Syntax: `a_list[start:stop]`

```
>>> s = [3, 186, 4431, 74400, 1048443]
>>> s[1:3]
[186, 4431]
>>> s[1:-1]
[186, 4431, 74400]
>>> s[2:]
[4431, 74400, 1048443]
>>> s[:2]
[3, 186]
>>> s[:]
[3, 186, 4431, 74400, 1048443]
>>> t = s
>>> t is s
True
>>> r = s[:]
>>> r is s
False
>>> r == s
True
>>> u = s.copy()
>>> u is s
False
>>> v = list(s)
>>>
```

Only References Are Copied



All of these techniques perform shallow copies.

```
False
>>> a == b
True
>>> a[0]
[1, 2]
>>> b[0]
[1, 2]
>>> a[0] is b[0]
True
>>> a[0] = [8, 9]
>>> a[0]
[8, 9]
>>> b[0]
[1, 2]
>>> a[1].append(5)
>>> a[1]
[3, 4, 5]
>>> b[1]
[3, 4, 5]
>>> a
[[8, 9], [3, 4, 5]]
>>> b
[[1, 2], [3, 4, 5]]
>>>
```

```
>>> c = [21, 37]
>>> d = c * 4
>>> d
[21, 37, 21, 37, 21, 37, 21, 37]
>>> [0] * 9
[0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> s = [ [-1, +1] ] * 5
>>> s
[[-1, 1], [-1, 1], [-1, 1], [-1, 1], [-1, 1]]
>>> s[2].append(7)
>>> s
[[-1, 1, 7], [-1, 1, 7], [-1, 1, 7], [-1, 1, 7], [-1, 1, 7]]
>>>
```

`list.index()`

Find the location of an object in a list.

Returns the index of the first list element which is equal to the argument.

```
>>> w = "the quick brown fox jumps over the lazy dog".split()
>>> w
['the', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
>>> i = w.index('fox')
>>> i
3
>>> w[i]
'fox'
>>> w.index('unicorn')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'unicorn' is not in list
>>> w.count("the")
2
>>> 37 in [1, 78, 9, 37, 34, 53]
True
>>> 78 not in [1, 78, 9, 37, 34, 53]
False
>>>
```

`del`

Remove an element from a list by index.

Syntax: `del a_list[index]`

```
>>> u = "jackdaws love my big sphinx of quartz".split()
>>> u
['jackdaws', 'love', 'my', 'big', 'sphinx', 'of', 'quartz']
>>> del u[3]
>>> u
['jackdaws', 'love', 'my', 'sphinx', 'of', 'quartz']
>>> u.remove('jackdaws')
>>> u
['love', 'my', 'sphinx', 'of', 'quartz']
>>> del u[u.index('quartz')]
>>> u.remove('pyramid')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>>
```

`list.insert()`

Insert an item into a list.

Accepts an item and the index of the new item.

```
>>> a = 'I accidentally the whole universe'.split()
>>> a
['I', 'accidentally', 'the', 'whole', 'universe']
>>> a.insert(2, "destroyed")
>>> a
['I', 'accidentally', 'destroyed', 'the', 'whole', 'universe']
>>> ' '.join(a)
'I accidentally destroyed the whole universe'
>>>
```

```
>>> m = [2, 1, 3]
>>> n = [4, 7, 11]
>>> k = m + n
>>> k
[2, 1, 3, 4, 7, 11]
>>> k += [18, 29, 47]
>>> k
[2, 1, 3, 4, 7, 11, 18, 29, 47]
>>> k.extend([76, 129, 199])
>>> k
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 129, 199]
>>>
```

`list.reverse()` and `list.sort()`

Common operations that modify a list in place.

```
>>> g = [1, 11, 21]
>>> g.reverse()
>>> g
[21, 11, 1]
>>> d = [17, 41, 29]
>>> d.sort()
>>> d
[17, 29, 41]
>>> d.sort(reverse=True)
>>> d
[41, 29, 17]
>>>
```

Key Parameter to list.sort()

Can be any callable object that accepts a single parameter.

Items passed to callable and sorted on its return value.

```
>>> h = 'not perplexing do handwriting family where I illegibly know doctors'.sp
lit()
>>> h
['not', 'perplexing', 'do', 'handwriting', 'family', 'where', 'I', 'illegibly',
'know', 'doctors']
>>> h.sort(key=len)
>>> h
['I', 'do', 'not', 'know', 'where', 'family', 'doctors', 'illegibly', 'perplexin
g', 'handwriting']
>>> ' '.join(h)
'I do not know where family doctors illegibly perplexing handwriting'
>>>
```

Reversing and sorting into copies

`reversed()` and `sorted()` are out-of-place equivalents to `list.reverse()` and `list.sort()`

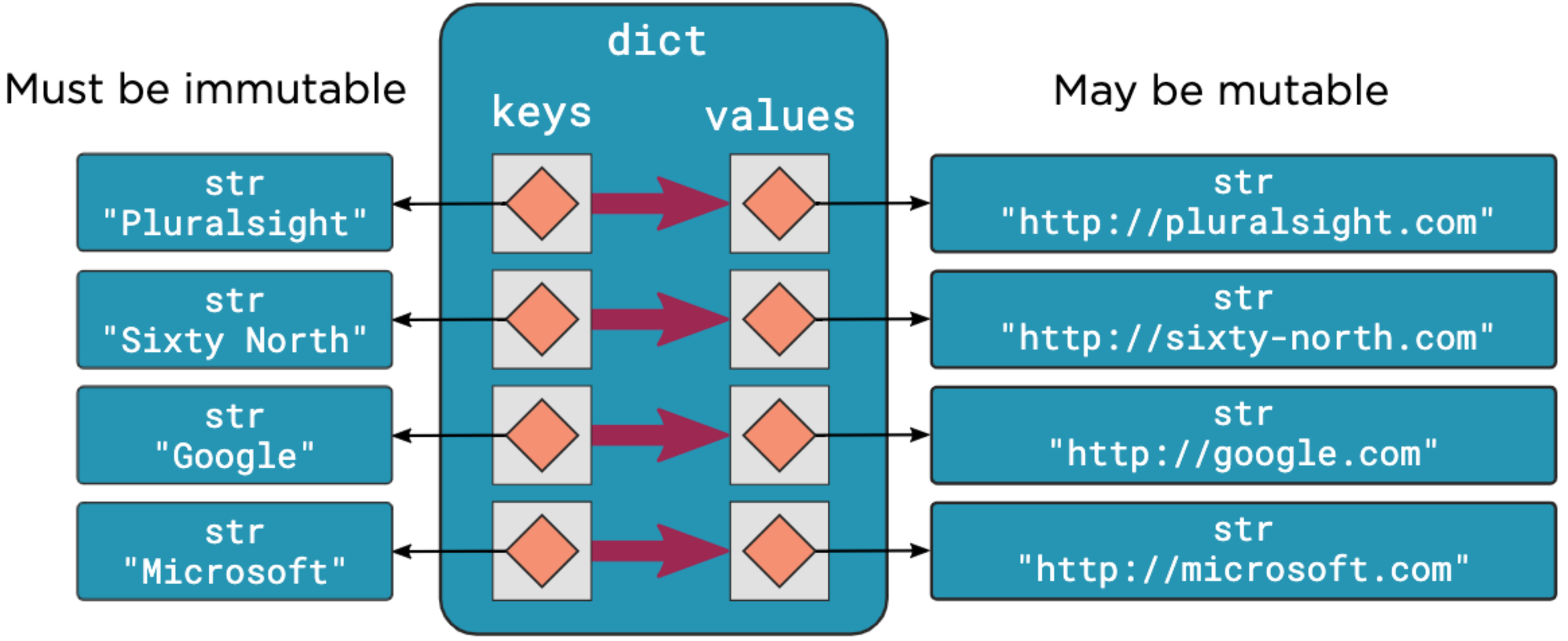
They return a reverse iterator and a new list, respectively

```
>>> x = [4, 9, 2, 1]
>>> y = sorted(x)
>>> y
[1, 2, 4, 9]
>>> x
[4, 9, 2, 1]
>>> p = [9, 3, 1, 0]
>>> q = reversed(p)
>>> q
<list_reverseiterator object at 0x10e93e290>
>>> list(q)
[0, 1, 3, 9]
>>>
```

Dictionaries

```
>>> urls = { 'Google': 'http://google.com',
...           'Pluralsight': 'http://pluralsight.com',
...           'Sixty North': 'http://sixty-north.com',
...           'Microsoft': 'http://microsoft.com' }
>>>
>>> urls['Pluralsight']
'http://pluralsight.com'
>>>
```

Dictionary Internals



Do not rely on the order of items in the dictionary

```
>>> names_and_ages = [ ('Alice', 32), ('Bob', 48), ('Charlie', 28), ('Daniel', 3
3) ]
>>> d = dict(names_and_ages)
>>> d
{'Alice': 32, 'Bob': 48, 'Charlie': 28, 'Daniel': 33}
>>> phonetic = dict(a='alfa', b='bravo', c='charlie', d='delta', e='echo', f='fo
xtrot')
>>> phonetic
{'a': 'alfa', 'b': 'bravo', 'c': 'charlie', 'd': 'delta', 'e': 'echo', 'f': 'fox
trot'}
>>>
```

As with lists, dictionary copying is shallow.

```
>>> d = dict(goldenrod=0xDAA520, indigo=0x4B0082, seashell=0xFFF5EE)
>>> e = d.copy()
>>> e
{'goldenrod': 14329120, 'indigo': 4915330, 'seashell': 16774638}
>>> f = dict(e)
>>> f
{'goldenrod': 14329120, 'indigo': 4915330, 'seashell': 16774638}
>>> g = dict(wheat=0xF5DEB3, khaki=0xF0E68C, crimson=0xDC143C)
>>> f.update(g)
>>> f
{'goldenrod': 14329120, 'indigo': 4915330, 'seashell': 16774638, 'wheat': 161133
31, 'khaki': 15787660, 'crimson': 14423100}
>>> stocks = {'GOOG': 891, 'AAPL': 416, 'IBM': 194}
>>> stocks.update({'GOOG': 894, 'YHOO': 25})
>>> stocks
{'GOOG': 894, 'AAPL': 416, 'IBM': 194, 'YHOO': 25}
>>>
```

`dict.update()`

Adds entries from one dictionary into another.

Call this on the dictionary that is to be updated.

Dictionary iteration

Dictionaries yield the next key on each iteration.

Values can be retrieved using the square-bracket operator.

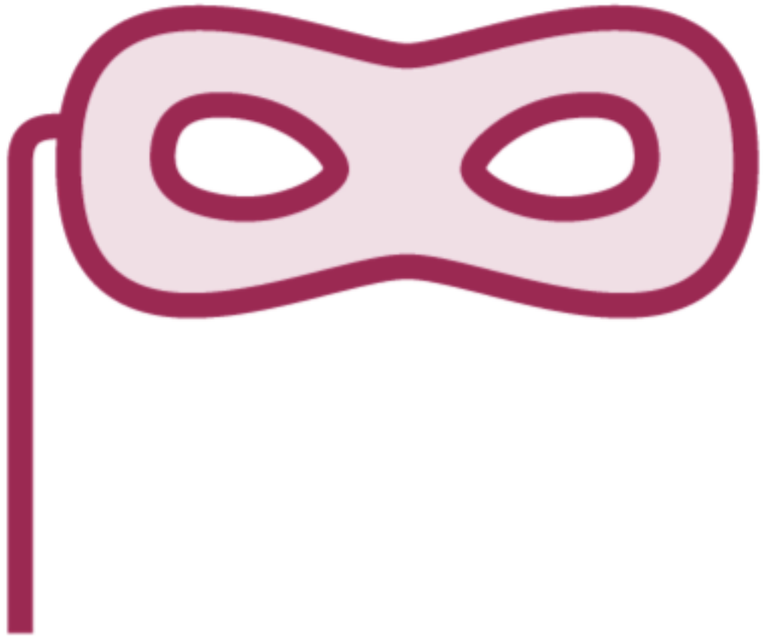
```
#A0522D
>>> for key in colors.keys():
...     print(key)
...
aquamarine
burlywood
chartreuse
cornflower
firebrick
honeydew
maroon
sienna
>>> for key, value in colors.items():
...     print(f"{key} => {value}")
...
aquamarine => #7FFFD4
burlywood => #DEB887
chartreuse => #7FFF00
cornflower => #6495ED
firebrick => #B22222
honeydew => #F0FFF0
maroon => #B03060
sienna => #A0522D
>>>
```

`dict.items()`

Iterates over keys and values in tandem.

Yields a `(key, value)` tuple on each iteration.

```
>>> del z['Fy']
>>> z
{'H': 1, 'Tc': 43, 'Xe': 54, 'Rf': 104, 'Fm': 100}
>>> m = {'H': [1, 2, 3],
...      'He': [3, 4],
...      'Li': [6, 7],
...      'Be': [7, 9, 10],
...      'B': [10, 11],
...      'C': [11, 12, 13, 14]}
>>> m['H'] += [4, 5, 6, 7]
>>> m
{'H': [1, 2, 3, 4, 5, 6, 7], 'He': [3, 4], 'Li': [6, 7], 'Be': [7, 9, 10], 'B':
[10, 11], 'C': [11, 12, 13, 14]}
>>> m['N'] = [13, 14, 15]
>>> from pprint import pprint as pp
>>> pp(m)
{'B': [10, 11],
 'Be': [7, 9, 10],
 'C': [11, 12, 13, 14],
 'H': [1, 2, 3, 4, 5, 6, 7],
 'He': [3, 4],
 'Li': [6, 7],
 'N': [13, 14, 15]}
>>>
```



Without the "`as pp`", the `pprint` function would mask the `pprint` module.

This kind of duplicate naming is probably best avoided in your own APIs.

set

Unordered collection of unique elements.

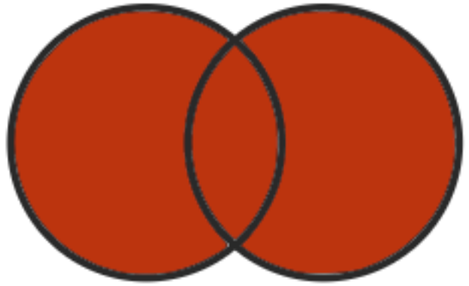
Sets are mutable.

Elements in a set must be immutable.

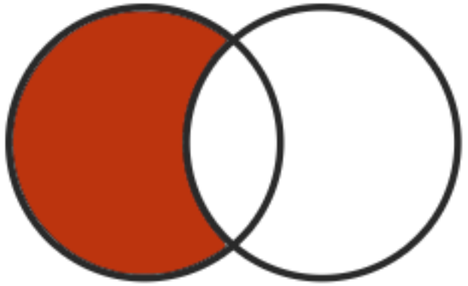
```
>>> p = {6, 28, 496, 8128, 33550336}
>>> p
{33550336, 8128, 6, 496, 28}
>>> type(p)
<class 'set'>
>>> d = {}
>>> type(d)
<class 'dict'>
>>> e = set()
>>> e
set()
>>> s = set([2, 4, 16, 64, 4096, 65536, 262144])
>>> s
{4096, 64, 2, 65536, 4, 262144, 16}
>>> t = [1, 4, 2, 1, 7, 9, 9]
>>> set(t)
{1, 2, 4, 7, 9}
>>>
```

```
>>> k.add(12)
>>> k
{81, 108, 12, 54}
>>> k.add(108)
>>> k.update([37, 128, 97])
>>> k
{128, 97, 37, 108, 12, 81, 54}
>>> k.remove(97)
>>> k
{128, 37, 108, 12, 81, 54}
>>> k.remove(98)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 98
>>> k.discard(98)
>>> k
{128, 37, 108, 12, 81, 54}
>>> j = k.copy()
>>> j
{128, 81, 37, 54, 108, 12}
>>> m = set(j)
>>> m
{128, 81, 37, 54, 108, 12}
>>>
```

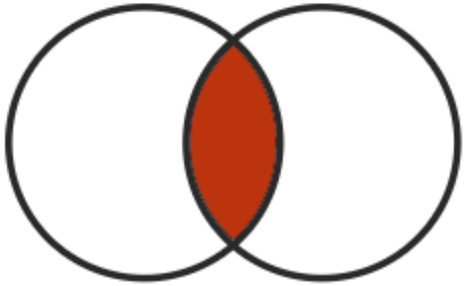
Set Algebra



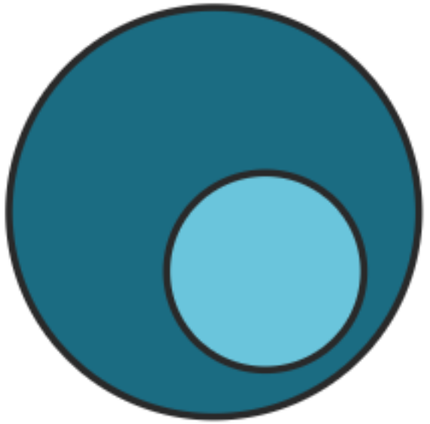
union



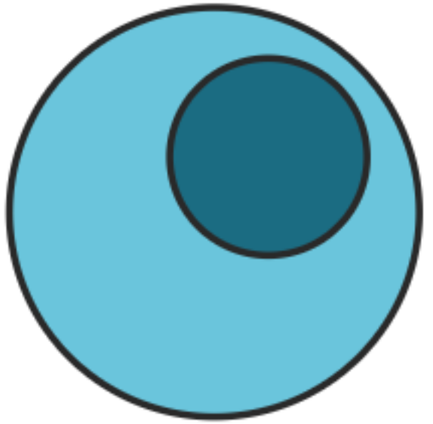
difference



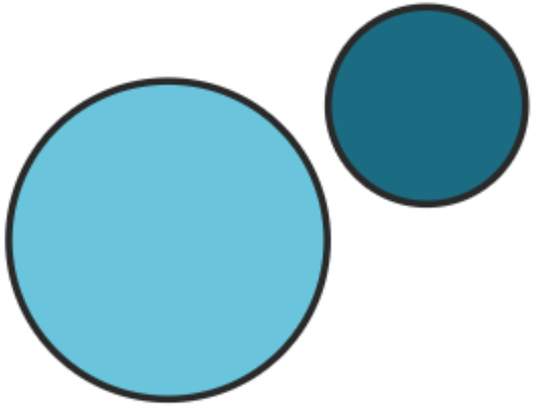
intersection



subset



superset



disjoint

```
>>> blue_eyes.union(blond_hair)
{'Olivia', 'Jack', 'Amelia', 'Lily', 'Harry', 'Joshua', 'Mia'}
>>> blue_eyes.union(blond_hair) == blond_hair.union(blue_eyes)
True
>>> blue_eyes.intersection(blond_hair)
{'Jack', 'Amelia', 'Harry'}
>>> blue_eyes.intersection(blond_hair) == blond_hair.intersection(blue_eyes)
True
>>> blond_hair.difference(blue_eyes)
{'Joshua', 'Mia'}
>>> blond_hair.difference(blue_eyes) == blue_eyes.difference(blond_hair)
False
>>> blond_hair.symmetric_difference(blue_eyes)
{'Olivia', 'Joshua', 'Mia', 'Lily'}
>>> blond_hair.symmetric_difference(blue_eyes) == blue_eyes.symmetric_difference
(blond_hair)
True
>>> smell_hcn.issubset(blond_hair)
True
>>> taste_ptc.issuperset(smell_hcn)
True
>>> a_blood.isdisjoint(o_blood)
True
>>>
```

Protocols

A set of operations that a type must support to implement the protocol.

Do not need to be defined as interfaces or base classes.

Types only need to provide functioning implementations.

Protocols

Protocol	Implementing collections
Container	`str, list, dict, range, tuple, set, bytes`
Sized	`str, list, dict, range, tuple, set, bytes`
Iterable	`str, list, dict, range, tuple, set, bytes`
Sequence	`str, list, range, tuple, bytes`
Mutable Sequence	`list`
Mutable Set	`set`
Mutable Mapping	`dict`

Protocols: Container

Protocol	Implementing collections
Container	`str, list, dict, range, tuple, set, bytes`
Sized	`str, list, dict, range, tuple, set, bytes`
Iterable	`str, list, dict, range, tuple, set, bytes`
Sequence	`str, list, range, tuple, bytes`
Mutable Sequence	`list`
Mutable Set	`set`
Mutable Mapping	`dict`

```
item in container
item not in container
```

Protocols: Sized

Protocol	Implementing collections
Container	`str, list, dict, range, tuple, set, bytes`
Sized	`str, list, dict, range, tuple, set, bytes`
Iterable	`str, list, dict, range, tuple, set, bytes`
Sequence	`str, list, range, tuple, bytes`
Mutable Sequence	`list`
Mutable Set	`set`
Mutable Mapping	`dict`

```
len(container)
```

Protocols: Iterable

Protocol	Implementing collections
Container	`str, list, dict, range, tuple, set, bytes`
Sized	`str, list, dict, range, tuple, set, bytes`
Iterable	`str, list, dict, range, tuple, set, bytes`
Sequence	`str, list, range, tuple, bytes`
Mutable Sequence	`list`
Mutable Set	`set`
Mutable Mapping	`dict`

Yield items one by one as they are requested.

```
for item in iterable:
    print(item)
```

Protocols: Sequence

Protocol	Implementing collections
Container	`str, list, dict, range, tuple, set, bytes`
Sized	`str, list, dict, range, tuple, set, bytes`
Iterable	`str, list, dict, range, tuple, set, bytes`
Sequence	`str, list, range, tuple, bytes`
Mutable Sequence	`list`
Mutable Set	`set`
Mutable Mapping	`dict`

```
item = sequence[index]
i = sequence.index(item)
num = sequence.count(item)
r = reversed(sequence)
```

iterable, sized, and container

Protocols: Others

Protocol	Implementing collections
Container	`str, list, dict, range, tuple, set, bytes`
Sized	`str, list, dict, range, tuple, set, bytes`
Iterable	`str, list, dict, range, tuple, set, bytes`
Sequence	`str, list, range, tuple, bytes`
Mutable Sequence	`list`
Mutable Set	`set`
Mutable Mapping	`dict`

Summary

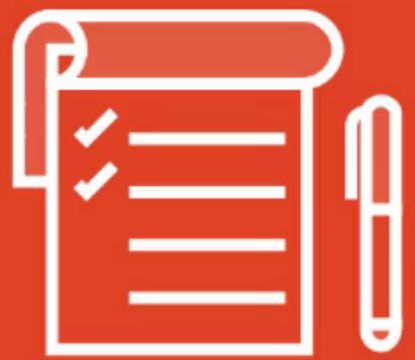
Tuples are immutable sequences

Tuple literals are optional parentheses around comma-separated items

Use a trailing comma for single-element tuples

Tuple unpacking is useful for multiple return values and swapping

Summary



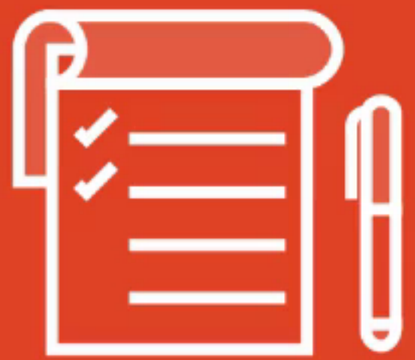
Use `str.join()` for efficient string concatenation

Use `str.partition()` for certain simple string parsing operations

`str.format()` is a powerful string interpolation technique

f-strings are a kind of string literal for performing interpolation

Summary



`range` objects represent arithmetic progressions of integers

`range()` can be called with one, two, or three arguments: start, stop, and step

`enumerate` is often better than `range` for making loop counters

Summary

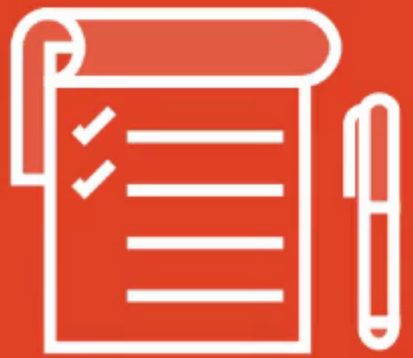
List supports indexing from the end with negative indices

Slicing copies all or part of a list

The full slice is a common idiom for copying lists

Use `list.index()` and `list.count()` to look for elements in a list

Summary



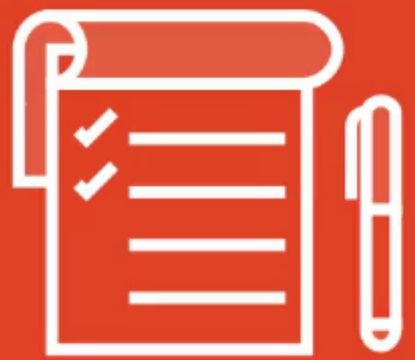
Use the `del` keyword to remove elements from a list

Sort and reverse lists in-place with `list.sort()` and `list.reverse()`

`sorted()` and `reversed()` sort and reverse any iterable

List copies are shallow, only copying the references

Summary



Dictionaries map from keys to values

Iteration and membership in dictionaries are over keys

Do not assume any order when iterating dictionary keys

`dict.keys()`, `dict.values()`, and `dict.items()` are iterable views into dictionaries

Copy dictionaries with `dict.copy()` or the `dict` constructor

Use `dict.update()` to extend one dictionary with another

Summary

Sets are unordered collections of unique elements

Sets support powerful set-algebra operations and predicates

Built-in collections can be organized by protocols

Underscore often represents unused values

The `pprint` module support pretty printing of compound data structures