learn-python

September 16, 2021

1 Strings

Strings are immutable, ordered representations of text. *Note:* since strings are immutable `+=` is very expensive.

1.1 String Formatting Functions

Formatted strings work similar to template strings in ES6.

[1]:

```
msg = "Hello, world!"
f'{msg}'
```

[1]:

```
'Hello, world!'
```

`.strip()` removes leading and trailing whitespace.

[2]:

```
msg = "   Hello, world!   "
msg.strip()
```

[2]:

```
'Hello, world!'
```

`.replace(old, new)` returns a new string where all occurences of `old` are replaced with `new`.

[3]:

```
msg.replace('l', 'L')
```

[3]:

```
'   HeLLo, worLd!   '
```

`.upper()` converts every character to upper-case. `.lower()` converts every character to lower. `.title()` capitalizes every word. `.capitalize()` makes the first word capital.

[4]:

```
msg = "HeLlO, wOrLd!"

print(f"upper: {msg.upper()}")
print(f"lower: {msg.lower()}")
print(f"title: {msg.title()}")
print(f"capitalize: {msg.capitalize()}")
```

```
upper: HELLO, WORLD!
lower: hello, world!
title: Hello, World!
capitalize: Hello, world!
```

1.2 Accesing

`.index()` returns the index of the first occurence of the string passed. If the string isn't found, a `ValueError` is thrown.

[5]:
```
msg = 'Hello, world!'
msg.index('w')
```

[5]:
```
7
```

Similar to `.index()`, `.find()` returns the index of the first occurence of the string passeed. However, if the string isn't found it returns `-1`.

[6]:
```
msg.find('a')
```

[6]:
```
-1
```

Brackets work the same as usual for indexing. Brackets can also use negative indexing to get the end of a list. Brackets can also be used to return substrings, *[begin, end, step = 1)*. If `begin` or `end` is omitted, it means until limit.

[7]:
```
print(msg[:3]) # beginning up until 3rd index
print(msg[:]) # copies a whole string
print(msg[-1]) # last character
print(msg[::-1]) # reverses a string
```

```
Hel
Hello, world!
!
!dlrow ,olleH
```

1.3 General Functions

`.count(str)` returns the number of times the passed string appears in the caller string.

[8]:
```
msg.count('l')
```

[8]:
```
3
```

Since strings are iterables, `in` can be used to check for a char and loop through a string.

[9]:
```
print('e' in msg)
```

```
True
```

1.4 List-like Functions

Strings can be split into a list using `.split(delim = ' ')`. The default delimiter is `' '`.

[10]:
```
words = 'hi my name is andy'.split()
print(words)
```

```
['hi', 'my', 'name', 'is', 'andy']
```

`.join(list)` is used to concatenate a list of strings into a single string. The caller string is used as a delimiter.

[11]:
```
'-'.join(words)
```

[11]:
```
'hi-my-name-is-andy'
```

2 Working with Numbers

2.1 % and // special cases

2.1.1 The 'funkiness'

// does integer division and returns the `floor()` of the result. Without it, we would get a float.

[12]:
```
12 // 5
```

[12]:
```
2
```

Returning the floor of the result can lead to some funky behavior since the answer isn't truncated towards 0 as it normally is in other languages. For example:

[13]:
```
# 5/2 = 2.5 -> truncate towards 0 -> 2
print(5 // 2)
# -5/2 = -2.5 -> floor(-2.5) -> -3
print(-5 // 2)
```

```
2
-3
```

This same funkiness shows up when using % with negative numbers.

[14]:
```
# 1 % 10 = 1 - 10 * int(1/10) -> 1
print(1 % 10)
# -1/10 = 1 - 10 * floor(1/10) -> 9
print(-1 % 10)
```

```
1
9
```

2.1.2 Workaround

When dealing with potentially negative numbers for integer division, opt to use float division and cast to `int`.

[15]:
```
print(-5 // 2)
print(int(-5 / 2))
```

```
-3
-2
```

When dealing with potentially negative numbers in modulo, opt to use `math.fmod(a, b)` or `math.remainder(a, b)`.

[16]:
```
from math import fmod, remainder
print(-1 % 10)
print(fmod(-1, 10))
print(remainder(-1, 10))
```

```
9
-1.0
-1.0
```

2.1.3 Explanation

`//` returns the floor of the result to maintain the mathematical relationship of integer division `a/b = q + r`. `%` is derived from this relationship by solving for `r`: `r = a - b * q`. Since this is integer division, `q = int(a/b)`. In both equations `r` is bounded by the interval `[0, b)` since it's the remainder.

However, to make `//` and `%` work with negative numbers, Python opts for `q = floor(a/b)` instead of `q = int(a/b)` like other languages. The reason behind this decision and more about `%` and `//` funkiness can be found here.

2.2 General Math Operators

`**` does exponents and is faster than `pow()`.

[17]:
```
2**3
```

[17]:
```
8
```

`abs()` gives the absolute value of a num.

[18]:
```
abs(-5)
```

[18]:
```
5
```

`max(a, b)` returns the max of two numbers. `min(a, b)` returns the min.

[19]:
```
print(max(10, 20))
print(min(10, 20))
```

```
20
10
```

`round()` rounds a float to the nearest int.

[20]:
```
round(1.3)
```

[20]:
```
1
```

2.3 math Module

The following functions need to be imported from the math module.

ceil() always rounds a float up.

[21]:
```
from math import ceil
ceil(3.2)
```

[21]:
```
4
```

floor() always rounds a float down.

[22]:
```
from math import floor
floor(3.7)
```

[22]:
```
3
```

sqrt() returns the square root of a number as a float.

[23]:
```
from math import sqrt
sqrt(4)
```

[23]:
```
2.0
```

3 Lists

Lists can be ordered, are mutable, and allow for duplicate elements.

3.1 Creating

To initialize a list, you can use list comprehension in various ways.

[24]:
```
empty = [0] * 5
print(empty)
```

```
[0, 0, 0, 0, 0]
```

[25]:
```
numbers = [i for i in range(1,6)]
print(numbers)

squares = [x * x for x in numbers]
print(squares)
```

```
[1, 2, 3, 4, 5]
[1, 4, 9, 16, 25]
```

3.2 Accessing

Brackets work the same as usual for indexing. Brackets can also use negative indexing to get the end of a list. Brackets can also be used to return subarrays, *[begin, end, step = 1)*. If begin or end is omitted, it means until limit.

[26]:

```
fruits = ['banana', 'apple', 'orange']
fruits[:2]
```

[26]:

```
['banana', 'apple']
```

Brackets have an optional step index as well which decides which increment to use for indices.

[27]:

```
fruits[::2]
```

[27]:

```
['banana', 'orange']
```

Similar to strings, `.index(item)` returns the index of the first occurence of *item* in the list. If the *item* isn't found, a `ValueError` is thrown.

[28]:

```
fruits.index('apple')
```

[28]:

```
1
```

To check for an item in a list in a conditional you can use `if ... in`.

[29]:

```
if 'apple' in fruits:
    print('Apple is in the list.')
```

```
Apple is in the list.
```

3.3 Inserting

`.extend(list)` can append one list onto another. The `+` operator can also be used to achieve the same effect.

[30]:

```
colors = ['yellow', 'red', 'orange']
fruits.extend(colors)
print(fruits)

print(fruits + ["green"])
```

```
['banana', 'apple', 'orange', 'yellow', 'red', 'orange']
['banana', 'apple', 'orange', 'yellow', 'red', 'orange', 'green']
```

`.append(item)` is used to add an item at the end.

[31]:

```
fruits = ['banana', 'apple', 'orange']
fruits.append('grapes')
print(fruits)
```

```
['banana', 'apple', 'orange', 'grapes']
```

`.insert(idx, item)` is used to insert an item at the specified index.

[32]:

```
fruits.insert(1, 'pear')
print(fruits)
```

```
['banana', 'pear', 'apple', 'orange', 'grapes']
```

3.4 Removing

`.remove(item)` is used to remove an item from the list. If the element DNE, a `ValueError` is thrown.

[33]:
```python
fruits.remove('grapes')
print(fruits)
```

```
['banana', 'pear', 'apple', 'orange']
```

`.pop()` removes the last item from the list and returns it.

[34]:
```python
fruits.pop()
```

[34]:
```
'orange'
```

`.clear()` empties and entire list.

[35]:
```python
fruits.clear()
print(fruits)
```

```
[]
```

3.5 General Functions

`.count(item)` returns the number of times *item* appears in the list.

[36]:
```python
fruits = ['banana', 'apple', 'orange']
fruits.append('apple')
fruits.count('apple')
```

[36]:
```
2
```

`.sort()` sorts the items of a list. It mutates the list. To avoid mutation and return a new list used the built-in `sorted()`.

[37]:
```python
fruits_copy = fruits.copy()
fruits.sort()
print(fruits)
print(fruits_copy)
```

```
['apple', 'apple', 'banana', 'orange']
['banana', 'apple', 'orange', 'apple']
```

`.reverse()` reverses a list.

[38]:
```python
fruits.reverse()
print(fruits)
```

```
['orange', 'banana', 'apple', 'apple']
```

Lists can be shallow copied with `.copy()`. They can also be deep copied with `.deepcopy()` from the `copy` module.

[39]:

```
from copy import copy, deepcopy

shallow = fruits.copy()
deep = deepcopy(fruits)
```

4 Tuples

Tuples are immutable, allow duplicate elements, and remain in their inserted order. They are generally used for data that's not going to be changed. They are defined with `()`.

[40]:

```
coordinates = (3, 4, 5)
print(coordinates[:1])
```

```
(3,)
```

Tuples can be unpacked and even converted into a list with the `*` operator.

[41]:

```
person = ("Andy", "Richard", "Carmen", "Mina")
first, *middle, last = person

print(first)
print(middle)
print(last)
```

```
Andy
['Richard', 'Carmen']
Mina
```

Since tuples are immutable, they take up less memory than lists.

[42]:

```
import sys
nums_list = [1, 2, 3]
nums_tup = (1, 2, 3)

print(f"list size: {sys.getsizeof(nums_list)} bytes")
print(f"tup size: {sys.getsizeof(nums_tup)} bytes")
```

```
list size: 120 bytes
tup size: 64 bytes
```

5 Dictionaries

Data structure with `(key, value)` pairs similar to a hash-map. They do not allow duplicates.

5.1 Creating

Dictionaries can be created with `{}`. They can also be constructed with `dict(key = value, ...)`. Dictionaries can also be constructed using dictionary comprehension.

[43]:

```
months = {
    "Jan": "January",
    "Feb": "February",
    "Mar": "March",
    "Apr": "April"
}
print(months)
```

```
{'Jan': 'January', 'Feb': 'February', 'Mar': 'March', 'Apr': 'April'}
```

5.2 Accessing

If the key doesn't exists, a `KeyError` will be raised.

[44]:

```
try:
    val = months["May"]
except KeyError as err:
    print(err)
```

```
'May'
```

To check if the key exists in the dictionary, use `if ... in`.

[45]:

```
if "May" not in months:
    print("May DNE.")
```

```
May DNE.
```

5.3 Iterating

`.keys()` returns a list of all of the keys in the dictionary that can be mutated. Updating the reference to `.keys()` will mutate the dictionary.

[46]:

```
keys = months.keys()
months["May"] = "May"
print(keys)
```

```
dict_keys(['Jan', 'Feb', 'Mar', 'Apr', 'May'])
```

`.values()` returns a list of all of the values in the dictionary that can be mutated. Updating the reference to `.values()` will mutate the dictionary.

[47]:

```
values = months.values()
months["June"] = "June"
print(values)
```

```
dict_values(['January', 'February', 'March', 'April', 'May', 'June'])
```

`.items()` will return an immutable list of (key, value) tuples in the dictionary. This is typically how to loop through a dictionary.

[48]:

```python
for (k, v) in months.items():
    print(k, v)
```

```
Jan January
Feb February
Mar March
Apr April
May May
June June
```

5.4 Adding

Dictionaries can be merged with another dictionary using `.update(dict)`. Merging can also be achieved with `a | b` (or in-place by using `=`).

[49]:

```python
more_months = {
    "July": "July",
    "Aug": "August",
    "Sept": "September",
    "Oct": "October",
    "Nov": "November",
    "Dec": "December"
}
months.update(more_months)
print(months.keys())
```

```
dict_keys(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'June', 'July', 'Aug', 'Sept',
'Oct', 'Nov', 'Dec'])
```

5.5 Removing

Items can be removed from the dictionary in three ways:

1. `del` - removes the item by reference
2. `.pop(key)` - removes the item by key and returns the value that was removed)
3. `.popitem()` - removes the last inserted item and returns the tuple that was removed)

[50]:

```python
keys = months.keys()
months["Andy"] = "Mina"
print(keys)

del months["Andy"]
print(keys)
```

```
dict_keys(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'June', 'July', 'Aug', 'Sept',
'Oct', 'Nov', 'Dec', 'Andy'])
dict_keys(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'June', 'July', 'Aug', 'Sept',
'Oct', 'Nov', 'Dec'])
```

[51]:

```
keys = months.keys()
months["Andy"] = "Mina"
print(keys)

removed = months.pop("Andy")
print(keys)
print(f"{removed} was removed")
```

```
dict_keys(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'June', 'July', 'Aug', 'Sept',
'Oct', 'Nov', 'Dec', 'Andy'])
dict_keys(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'June', 'July', 'Aug', 'Sept',
'Oct', 'Nov', 'Dec'])
Mina was removed
```

[52]:

```
keys = months.keys()
months["Andy"] = "Mina"
print(keys)

removed = months.popitem()
print(keys)
print(f"{removed} was removed")
```

```
dict_keys(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'June', 'July', 'Aug', 'Sept',
'Oct', 'Nov', 'Dec', 'Andy'])
dict_keys(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'June', 'July', 'Aug', 'Sept',
'Oct', 'Nov', 'Dec'])
('Andy', 'Mina') was removed
```

Dictionaries can be cleared with `.clear()`.

[53]:

```
months.clear()
print(months)
```

```
{}
```

6 Sets

Sets are unordered, mutable, and don't allow for duplicates.

6.1 Creating

They are made with `{items}`. Simply using `{}` to create an empty will create a dictionary. They can be constructued from an iterable with `set(iter)`. Sets can also be constructed using set comprehension.

[54]:

```
nums = set([1, 2, 3])
print(nums)
```

```
{1, 2, 3}
```

6.2 Accessing

Just like dictionaries, checking if an item is in the set uses `in`.

[55]:
```
print(3 in nums)
```

```
True
```

6.3 Adding and Removing

`.add(item)` adds the item to the set.

[56]:
```
nums.add(4)
print(nums)
```

```
{1, 2, 3, 4}
```

Elements can be removed with `.remove(item)`. If the item DNE, a KeyError will be raised.

[57]:
```
nums.remove(3)
print(nums)
```

```
{1, 2, 4}
```

`.discard(item)` will remove the element if it exists. If it DNE, it does nothing.

[58]:
```
nums.discard(3)
print(nums)
```

```
{1, 2, 4}
```

Similar to dictionaries, sets can be merged with `.update(set)`. Multiple sets can be merged with `a | b | c ...` (or in-place using `|=` as the first operator). `.update(iter)` also accepts iterables.

[59]:
```
nums.update([3, 4, 5, 5])
print(nums)
```

```
{1, 2, 3, 4, 5}
```

Sets can be cleared with `.clear()`.

[60]:
```
nums.clear()
print(nums)
```

```
set()
```

6.4 General Functions

`.union(set)` returns a new set which merges the two sets. The `|` operator is syntactic sugar.

[61]:
```
nums = set([i for i in range(1, 6)])

lucky_numbers = {1, 3, 5}
nums.union(lucky_numbers)
```

[61]:

```
{1, 2, 3, 4, 5}
```

`.intersection(set)` returnsa new set containing the intersection of two sets. The `&` operator is syntactic sugar.

[62]:

```
setA = {1, 2, 3, 4, 5}
setB = {1, 2, 3, 6, 7, 8, 9}

setA & setB
```

[62]:

```
{1, 2, 3}
```

`.difference(set)` returns a new set of the elements that appear in the first set, but not the second set. The `-` operator is syntactic sugar.

[63]:

```
setA - setB
```

[63]:

```
{4, 5}
```

`.symmetric_difference(set)` returns a new set containing all of the non-shared elements between both sets. The logical equivalent is `a - b  b - a`. The `^` operator is syntactic sugar.

[64]:

```
setA.symmetric_difference(setB)
```

[64]:

```
{4, 5, 6, 7, 8, 9}
```

`.intersection()`, `.difference()`, and `.symmetric_difference()` can all appended with `_update` (namely `.difference_update()`) to mutate the caller set. This can also be achieved by using their respective operators with `=`, similar to `+=`.

[65]:

```
setA -= setB
print(setA)
```

```
{4, 5}
```

6.5 Set Comparison

`.disjoint(set)` checks if the caller set and the set passed don't share any elements.

[66]:

```
setA = {1, 2, 3}
setB = {4, 5, 6}

setA.isdisjoint(setB)
```

[66]:

```
True
```

`.issubset(set)` checks if the caller set is a subset of the passed set. `<=` is syntactic sugar for `.issubset(set)`. `<` checks for a **proper** subset (`a <= b and a != b`).

[67]:

```
setA = {1, 2, 3, 4, 5}
setB = {1, 2, 3}
```

```
setB.issubset(setA)
```

[67]:
```
True
```

`.issuperset(set)` checks if the caller set is a superset of the passed set. `>=` is syntactic sugar for `.issuperset(set)`. `>` checks for a **proper** subset (`a >= b and a != b`).

[68]:
```
setA.issuperset(setB)
```

[68]:
```
True
```

6.6 Frozen Sets

Frozen sets can be constructued from an iterable similar to sets. The difference between the two is that frozen sets can't be modified.

[69]:
```
frozen = frozenset([1, 2, 3])
```

7 `collections` Module

7.1 `Counter`

The counter is data structure that stores the counts of keys as the value (key, frequency). They inherit `.keys()`, `.values()`, and `.items()` from dictionaries.

7.1.1 Creating

Counters can be created by passing an iterable to `Counter(iter)`.

[70]:
```
from collections import Counter
my_counter = Counter("aaabbc")
```

7.1.2 Accessing

Counters are accessed the same way as dictionaries. They also offer a `.most_common(n)` functions that returns a list of the `n` highest counts pairs as tuples.

[71]:
```
my_counter.most_common(2)
```

[71]:
```
[('a', 3), ('b', 2)]
```

7.2 `namedtuple`

namedtuples are most similar to structs in C++.

7.2.1 Creating

namedtuples are created using `namedtuple('name', 'props')` where props are separated by commas or spaces.

[72]:

```
from collections import namedtuple
Point = namedtuple('Point', 'x y')
p = Point(-1, 2)
print(p)
```

```
Point(x=-1, y=2)
```

7.2.2 Accessing

Props of a named tuple can be accessed by their name.

[73]:

```
print(f"x: {p.x}, y: {p.y}")
```

```
x: -1, y: 2
```

7.3 deque

Similar to a double-ended queue from other languages.

7.3.1 Creating

Deques can be created using `deque(iter)` where the passed iterable is optional. *Note:* The iterable is inserted in queue order so the last element in the iterable will be the first item in the new deque.

[74]:

```
from collections import deque
d = deque([1, 2, 3])
print(d)
```

```
deque([1, 2, 3])
```

7.3.2 Adding and Removing

Adding can be achieved through `.append(item)` or `.appendleft(item)`.

[75]:

```
d.append(4)
d.appendleft(0)
print(d)
```

```
deque([0, 1, 2, 3, 4])
```

Iterables can be added with `.extend(iter)` or `.extendleft(iter)`. *Note:* With `.extendleft()`, items are inserted in queue order so the last item will be the first item in the deque.

[76]:

```
d.extend([5, 6])
d.extendleft([-1, -2])
print(d)
```

```
deque([-2, -1, 0, 1, 2, 3, 4, 5, 6])
```

`.insert(idx, item)` is used to insert an item at a specified index. `O(n)` runtime.

[77]:

```
d.insert(0, -3)
print(d)
```

```
deque([-3, -2, -1, 0, 1, 2, 3, 4, 5, 6])
```

Removing is done through `.pop()` or `.popleft()`. Both functions return the popped item.

[78]:

```
first = d.pop()
last = d.popleft()
print(f"{first} used to be first.")
print(f"{last} used to be last.")
print(d)
```

```
6 used to be first.
-3 used to be last.
deque([-2, -1, 0, 1, 2, 3, 4, 5])
```

`.remove(item)` is used to remove the first occurrence of item. If item DNE, a `ValueError` is raised. `O(n)` runtime.

[79]:

```
d.remove(4)
print(d)
```

```
deque([-2, -1, 0, 1, 2, 3, 5])
```

The deque can be cleared with `.clear()`.

[80]:

```
d.clear()
```

7.3.3 Accessing

Deques support random access with `[]`. Random access is `O(n)`, but front/back access is `O(1)`.

`.index(item)` can be used to get the index of the specified item. `O(n)` runtime.

[81]:

```
d = deque([1, 2, 3])
d.index(2)
```

[81]:

```
1
```

7.3.4 General Functions

`.count(item)` counts the number of `items` in the deque.

[82]:

```
d.append(3)
d.count(3)
```

[82]:

```
2
```

`.reverse()` reverses the deque in-place and returns `None`.

[83]:

```
d.reverse()
print(d)
```

```
deque([3, 3, 2, 1])
```

`.rotate(n = 1)` rotates the deque by `n` to the right. If `n` is negative, the deque is rotated left. Rotating one step is `O(1)` since it's equivalent to `d.append(d.pop())`. Otherwise, rotations are `O(n)` where `n` is the parameter.

[84]:

```
print(d)
d.rotate()
print(d)
```

```
deque([3, 3, 2, 1])
deque([1, 3, 3, 2])
```

8 itertools module

8.1 product

`product(*iters)` returns the Cartesian product of the input iterables as tuples when casted to a list.

[85]:

```
from itertools import product
a = [1, 2]
b = [3, 4]
prod = list(product(a, b))
print(prod)
```

```
[(1, 3), (1, 4), (2, 3), (2, 4)]
```

8.2 permutations

`permutations(iter, r = None)` returns all possible permutations of an iterable as tuples when casted to a list. `r` represents the size limit of generated permutations.

[86]:

```
from itertools import permutations
a = [1, 2, 3]
perm = permutations(a)
print(list(perm))
```

```
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
```

8.3 combinations

`combinations(iter, r)` returns all combinations of an iterable as tuples when casted to a list. `r` represents the size limit of generated combinations.

[87]:

```
from itertools import combinations as comb
a = [1, 2, 3, 4]
c = comb(a, 2)
```

```
print(list(c))
```

```
[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
```

8.4 combinations_with_replacement

combinations_with_replacement(iter, r) retuns all possible combinations of an iterable with replacement as tuples when casted to a list. r represents the size limit of generated combinations.

[88]:

```
from itertools import combinations_with_replacement as combr
a = [1, 2, 3]
c_with_r = combr(a, 2)
print(list(c_with_r))
```

```
[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```

9 For loops

For loops in Python are a lot more flexible than they are in other languages. They can be used as a for ... in loop for many things.

[89]:

```
for char in "Andy Mina":
    print(char)
```

```
A
n
d
y
 
M
i
n
a
```

[90]:

```
for fruits in ["banana", "apple", "grapes"]:
    print(fruits)
```

```
banana
apple
grapes
```

The range() function can be used to generate a sequences of numbers to loop with. It takes three parameters: range(start = 0, stop, step = 1). The integer passed as stop is not included in the range.

[91]:

```
for i in range(5):
    print(i)
```

```
0
1
```

```
2
3
4
```

10 Try/Except Blocks

Works similar to how it does in other languages. `except` statements can be chained together like a `switch` block to catch different errors.

[92]:

```python
try:
    value = 1/0
    print(value)
except ZeroDivisionError:
    print('Undefined')
```

```
Undefined
```

11 File Streams

Files can be opened with `open(path, option = "r")`. You can specify what to do with the open file by passing the correct option string:

- "r": Read a file; errors if the file DNE
- "w": Write to a file; creates the file if it DNE
- "r+": Read and write a file; creates the file if it DNE
- "a": Appends to a file; creates the file if it DNE
- "x": Create a file; errors if the file exists already
- "t": Parse the file as text
- "b": Parse the file as binary

Never forget to close the file with `.close()`.

[93]:

```python
sample = open('sample.txt')
sample.close()
```

Files can be read with `.read()` to read the whole file into a string or `.readline()` to only read one line. `.readline()` moves the cursor to the next line which affects subsequent read commands.

[94]:

```python
sample = open('sample.txt', "r+")

single = sample.readline()
whole = sample.read()

print(single)
print(whole)
```

```
Hi, my name is Andy Mina.

My girlfriend's name is Sabina Kubayeva.
Goodbye.
```

The cursor can be moved by using `.seek()` with one of three parameters: 1. `0` which moves the pointer to the beginning of the file 2. `1` which moves the pointer relative to the current position 3. `2` which moves the pointer to the end of the file

[95]:

```
sample.seek(0)
```

[95]:

```
0
```

`.readlines()` reads all of the lines into a list which can then be used in a `for ... in` loop.

[96]:

```
lines = sample.readlines()
print(lines)
```

```
['Hi, my name is Andy Mina.\n', "My girlfriend's name is Sabina Kubayeva.\n",
'Goodbye.']
```

`.write(line)` is used to add a line onto a file.

[97]:

```
sample.write('Welcome back!\n')
```

[97]:

```
14
```

12 Classes and Objects

Classes inherit from parents classes by being passed as a parameter in the signature. Children classes do not need an `__init__` constructor.

[98]:

```
class Animal:
    def __init__(self):
        self.planet = 'Earth'

    def eat(self):
        print(f'I eat food on {self.planet}.')

class Dog(Animal):
    def speak(self):
        print('Bark!')

sunny = Dog()
sunny.speak()
```

```
Bark!
```

12.1 Overloading

12.1.1 Operator Overloading

Mathematical operators can be overloaded. Prepending any of the math overload signatures with `i` such as `a.__iadd__(b)` is an in-place specifier to be used in scenarios like `a += b`.

Operator	Expression	Overload signature
Addition	a + b	a.__add__(b)
Subtraction	a - b	a.__sub__(b)
Multiplication	a * b	a.__mul__(b)
Power	a ** b	a.__pow__(b)
Division	a / b	a.__truediv__(b)
Int Division	a // b	a.__floordiv__(b)
Modulo	a % b	a.__mod__(b)
Bitwise Left Shift	a << b	a.__lshift__(b)
Bitwise Right Shift	a >> b	a.__rshift__(b)
Bitwise NOT	~a	a.__not__()
Bitwise AND	a & b	a.__and__(b)
Bitwise OR	a | b	a.__or__(b)
Bitwise XOR	a ˆ b	a.__xor__(b)
Bracket Evaluation	c = a[b]	a.__getitem(key)__
Bracket Assignment	a[b] = c	a.__setitem(key, value)__

Comparison operators can also be overloaded.

Operator	Expression	Overload signature
Equal	a == b	a.__eq__(b)
Not equal	a != b	a.__ne__(b)
Less than	a < b	a.__lt__(b)
Less than or equal to	a <= b	a.__le__(b)
Greater than	a > b	a.__gt__(b)
Greater than or equal to	a >= b	a.__ge__(b)

12.1.2 Function Overloading in Classes

All of the following functions take `self` as the first parameter since they are implemented within classes.

`__new__()`, `__init__()`, and `__del__()` `__new__` is called to create a instance of a class. `__init__` is invoked `__new__` and is commonly used as the constructor. `__del__` is the "clean up" functions for a class and is improperly referred to as a destructor. `__del__` is called when an instance is *about* to be destroyed. `del x` decreases the reference count of `x` by 1 and `__del__` is called when the reference count of `x` is 0.

`__str__()` `__str__` is called for the built-in functions `print()` and `format()`, which is called for formatted string literals.

`__hash__()` `__hash__` is called by the built-in `hash()` and should return an integer. `__eq__` and `__hash__` should be implemented together; `__hash__` will not work without `__eq__`.

`__getitem__(key)` and `__setitem__(key, value)` `__getitem(key)__` defines the bracket evaluation operator for a class, namely `self[key]`. `__setitem(key, value)__` defines the bracket assignment operator for a class, namely `self[key] = x`.

- If `key` is the wrong type, raise `TypeError`.
- If `key` provides an index out of bounds, raise `IndexError`.
- If `key` is not in the container, raise `KeyError`.