- Short-Circuiting

  expr1 and expr2

  If expr1 = False, Python does not evaluate the expr2. So, always add first the expression that most probably evaluates to False

  expr1 or expr2

  If expr1 = True, Python does not evaluate the expr2. So, always add first the expression that most probably evaluates to True

- Shallow copies work only for mutable objects

  e.g. a = (1, 2, 3)
  b = a[:]
  } → a is b → True

  Deep copies apply to both

- If a class has many attributes, it's a sign that a group of these attributes should be moved to a separate class. Better have MANY SMALL CLASSES.

- When overriding the `__lt__()` etc methods, they are used by Python in its `.sort()` method

e.g. we have a list of cards

```
card1 = Card(rank="10", suit="Spades")
card2 = Card(rank="10", suit="Clubs")
card3 = Card(rank="Ace", suit="Hearts")
cards = [card1, card2, card3]
cards.sort()
```

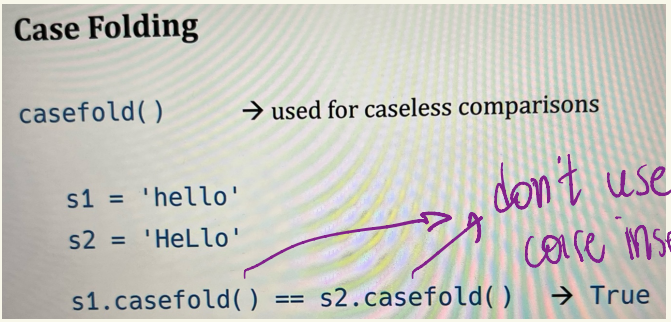
```
class Card():
    ⋮
    def __lt__(self, other)
        if self.rank == other.rank:
            return self.suit < other.suit
        return self.rank < other.rank
```

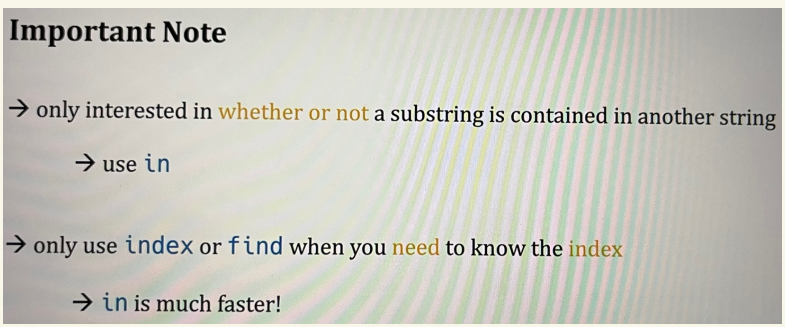
alphabetically for same rank

increasing order by rank

- ? <module> returns info about the classes / methods in the specified module

Strings





String: index vs. find

- index → raises ValueError if not found
- find → returns -1 if not found

Better use index because it is used in other sequences as well (e.g., lists), whereas find() is used only in strings.

Formatting Strings

```
bid = 1.5760
ask = 1.5763
```

```
'bid: {}, ask: {}, spread: {}'.format(bid, ask, ask-bid)
```

```
'bid: 1.576, ask: 1.5763, spread: 0.00029999999999996696'
```

```
f"bid: {bid}, ask: {ask}, spread: {ask - bid}"
```

```
'bid: 1.576, ask: 1.5763, spread: 0.00029999999999996696'
```

```
'bid: {b}, ask: {a}, spread: {spread}'.format(a=ask, spread=ask-bid, b=bid)
```

```
'bid: 1.576, ask: 1.5763, spread: 0.00029999999999996696'
```

```
format(0.1, '.10f')
```

```
'0.1000000000'
```

```
'bid: {:.4f}, ask: {:.4f}, spread: {:.4f}'.format(bid, ask, ask-bid)
```

```
'bid: 1.5760, ask: 1.5763, spread: 0.0003'
```

```
'bid: {b:.4f}, ask: {a:.4f}, spread: {spread:.4f}'.format(a=ask, spread=ask-bid, b=bid)
```

```
'bid: 1.5760, ask: 1.5763, spread: 0.0003'
```

```
f"bid: {bid:.4f}, ask: {ask:.4f}, spread: {(ask - bid):.4f}"
```

```
'bid: 1.5760, ask: 1.5763, spread: 0.0003'
```

```
f"bid: {bid:.4f}, ask: {ask:.4f}, spread: {ask - bid:.4f}"
```

```
'bid: 1.5760, ask: 1.5763, spread: 0.0003'
```

```
def process_data(data, item_sep=',', line_sep='\n'):
    output = ''

    for row in data:
        for element in row:
            output = output + str(element) + item_sep
        output = output + line_sep

    return output
```

```
print(process_data(data))
print('done')
```

```
10,20,30,
100,200,300,
1000,2000,3000,

done
```

avoid string concatenation because strings are immutable objects and thus new strings are created when str1 + str2

Instead, use join

```
def process_data(data, item_sep=',', line_sep='\n'):
    row_strings = [
        item_sep.join(str(el) for el in row)
        for row in data
    ]
    return line_sep.join(row_strings)
```

```
print(process_data(data))
```

```
10,20,30
100,200,300
1000,2000,3000
```

refactoring

```
def process_row(row, item_sep):
    return item_sep.join(str(el) for el in row)

def process_data(data, item_sep=',', line_sep='\n'):
    row_strings = [process_row(row, item_sep) for row in data]
    return line_sep.join(row_strings)
```

generator

```
process_data(data, item_sep='|')
```

```
'10,20,30\n100,200,300\n1000,2000,3000'
```

Iteration

Viewing Contents of `range` Object

```
r = range(5)
print(r)
```

→ it is a 'range' object (neither list nor tuple)

→ 'range(5)'

→ not what we wanted

→ can convert `range` object to a `list` or `tuple`

```
print(tuple(r)) → (0, 1, 2, 3, 4)
print(list(r))
```

range(start, end, step)

start: inclusive; end: exclusive

reminds us of slicing!

- If we are to add/remove elements from an iterable, do not use for-loop! Use while-loop instead!

The `else` Clause

→ Python is really confusing here...

`for` loops can have an `else` clause

→ but it has nothing to do with the `else` clause of an `if` statement

→ the `else` clause of a `for` loop executes if and only if no `break` was encountered

in my mind I read it as "else if no break"

```
for i in range(5):
    <code block 1>
else: # if no break
    <code block 2>
```

→ `<code block 2>` executes if loop terminated normally

Back to our Example

```
found = False

for el in my_list:
    if el == 'Python':
        found = True
        print('found')
        break

if not found:
    print('not found')
```

equivalently:

```
for el in my_list:
    if el == 'Python':
        print('found')
        break
else:  # if no break
    print('not found')
```

Dictionaries

Hash Maps (aka Dictionaries)

→ better implementation is the hash map (or dictionary)

→ similar to the last approach we saw

→ but a special mechanism is used to quickly find a key

→ lookup speed is not affected by size of dictionary

IMPORTANT → keys must be hashable (hence the term hash map)

→ what that means exactly is not important now

→ strings are hashable → numerics are hashable

→ tuples *may* be hashable (if all the elements are themselves hashable)

→ lists are not hashable

Python Dictionaries

→ a dictionary is a data structure that associates a value to a key

→ both value and key are Python objects

→ key must be hashable type (e.g. `str`, `int`, `bool`, `float`, ...) and unique

→ value can be any type

→ type is `dict`

→ it is a collection of key: value pairs

→ it is iterable

→ but it is not a sequence type

→ values are looked up by key, not by index

→ technically there is no ordering in a dictionary (we'll come back to this point!)

→ it is a mutable collection

Membership testing (i.e., if a key exists) is extremely fast in dictionaries

The `get()` Method

```
d = {'length': 10, 'width': 20}
```

`d.get('length', 0)` → `10`

the key exists, so the corresponding value (`10`) is returned

`d.get('height', 0)` → `0`

the key does not exist, so the default (`0`) is returned

`d.get('height')` → `None`

the key does not exist, so the default default-value (`None`) is returned

The `get()` Method

→ data in Python is often handled using dictionaries

when we work with data we often have missing values

sometimes, not only is the value missing, but the key as well

→ using `get()` allows us to simplify our code to assign a default for missing keys

```
if 'ssn' in person_dict:
    social = person_dict['ssn']
else:
    social = ''
```

→ `social = person_dict.get('ssn', '')`

Similar method for setting values is .setdefault()

Sets

think back to keys in a dictionary

- → they are unique
- → they are iterable
- → they have no particular order (well, Python 3.6 maintains insertion order)
- → keys can be added or removed (dictionary is mutable)
- → they are hashable too - but leave that aside for a moment

does that remind you of a set?

- → we can think of the keys in a dictionary as a set

→ Python's implementation of sets is essentially like a dictionary

- → but no values, only keys
- → because of this, set elements must be hashable too

Python Sets

- → type is `set`
- → sets are iterable
- → iteration order is not guaranteed (at least not yet)
- → set elements must be hashable
- → sets are mutable
  - → sets are not hashable
    - → a set cannot be an element of another set, or a key in a dictionary
    - → if you really want nested sets, use `frozenset`
      - → immutable equivalent of sets – those are hashable (if all the elements are, themselves, hashable)

Compared to dictionaries, which maintain the insertion order, sets don't maintain any order.

Applications of Sets

- → membership testing with sets is much faster than lists or tuples
- → easy to eliminate duplicate values from a collection
- → easy to find common values between two collections
- → easy to find values in one collection but not in another

Comprehensions

They are a bit faster than using for-loops
But use them only for simple cases
Readability matters!!!