Python advanced class

Module 3, Complex data structures

- Designing data structures
- Comprehensions
- Enum

Designing data structures

Lists vs. Tuples

- Lists and tuples are very alike, but there are important differences in use:
  - Lists are typically used for sequences of objects of comparable types, like a list of town names
  - Tuples are used for record of fields like longitude, latitude, elevation, and town name
- This also manifests in mutablility:
  - List are mutable and may be sorted, added to, removed from, and inserted into
  - Tuples are immutable and if the data needs change, a new tuple is created instead

In [1]:
```
persons = [
    ('Andrew', 34, 1.82),
    ('Ben', 38, 1.76),
    ('Charlie', 35, 1.85),
]
for name, age, height in persons:
    print(name, age, height)
```

```
Andrew 34 1.82
Ben 38 1.76
Charlie 35 1.85
```

Dicts vs. Named Tuples

- Dictionaries are ...
  - Free-formed and may have any immutable type as key, and any type as value
  - They are mutable are easily added to and removed from
  - There is no inforcement of a particular structure
- Named tuples are ...
  - Has a specific form as it is defined, and attributes must be strings
  - The named tuple is immutable, but can consist of mutable types
  - New attributes can't be introduced in the instance of a named tuple

Named Tuples

- Named tuples are created as a class

In [2]:
```
import typing as t
class Person(t.NamedTuple):
    name: str
    age: int
    height: float
persons = [
    Person('Andrew', 34, 1.82),
    Person('Ben', 38, 1.76),
    Person('Charlie', 35, 1.85),
]
for p in persons:
    print(p.name, p.age, p.height)
```

```
Andrew 34 1.82
Ben 38 1.76
Charlie 35 1.85
```

When to use classes instead

- Named tuples can serve as a replacement for classes, if the object is static
- But it is better to define a class if:
    - The object should be mutable
    - If there are non-trivial operations that would be more convinient defined as a method
- Dataclasses are just as convinient to define as named tuples

Common structures

- List of lists: Pseudo multi-dimensional arrays
- List of (named) tuples: Like tables in a database
- List of dictionaries: Open-ended representation of keyword based data
- Dictionary of (named) tuples: Indexed data table
- Named tuple of named tuples: A dataset of composed data

Comprehensions

List comprehension

- A list comprehension is a list expression that would otherwise require at least three to four lines of code
- The format is a bit terse, but the alternative is often a map with a lambda function (which is often very unreadable)
- The format is `[` *expr* `for` *var* `in` *iterable* `]`

In [3]:
```
squares = [ x**2 for x in range(1, 10+1) ]
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

In [4]:
```
squares = list(map(lambda x: x**2, range(1, 10+1)))
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Conditionals

- A list comprehension may additionally have a condition after the `for` / `in` part

In [5]:
```
squares = [ x for x in range(1, 100+1) if int(x**0.5) == float(x**0.5)
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Nested for loops

- A list comprehension can have more than one for loop
- The leftmost runs slowest, the rightmost runs fastest

In [6]:
```python
chessboard = [ letter + str(digit) for letter in list("ABCDEFGH") for di
print(chessboard)
```

```
['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7', 'A8', 'B1', 'B2', 'B
3', 'B4', 'B5', 'B6', 'B7', 'B8', 'C1', 'C2', 'C3', 'C4', 'C5',
'C6', 'C7', 'C8', 'D1', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7', 'D
8', 'E1', 'E2', 'E3', 'E4', 'E5', 'E6', 'E7', 'E8', 'F1', 'F2',
'F3', 'F4', 'F5', 'F6', 'F7', 'F8', 'G1', 'G2', 'G3', 'G4', 'G
5', 'G6', 'G7', 'G8', 'H1', 'H2', 'H3', 'H4', 'H5', 'H6', 'H7',
'H8']
```

Nested Comprehensions

- A list comprehension may have a list comprehension inside

In [7]:
```python
from pprint import pprint
chessboard = [ [ letter + str(digit) for letter in list("ABCDEFGH") ] fo
pprint(chessboard)
```

```
[['A1', 'B1', 'C1', 'D1', 'E1', 'F1', 'G1', 'H1'],
 ['A2', 'B2', 'C2', 'D2', 'E2', 'F2', 'G2', 'H2'],
 ['A3', 'B3', 'C3', 'D3', 'E3', 'F3', 'G3', 'H3'],
 ['A4', 'B4', 'C4', 'D4', 'E4', 'F4', 'G4', 'H4'],
 ['A5', 'B5', 'C5', 'D5', 'E5', 'F5', 'G5', 'H5'],
 ['A6', 'B6', 'C6', 'D6', 'E6', 'F6', 'G6', 'H6'],
 ['A7', 'B7', 'C7', 'D7', 'E7', 'F7', 'G7', 'H7'],
 ['A8', 'B8', 'C8', 'D8', 'E8', 'F8', 'G8', 'H8']]
```

Dict comprehension

- A dict comprehension is like a list comprehension
- It is written in curly brances and the expression must be a pair

In [8]:

```
names = ["andrew", "ben", "charlie", "dennis", "eric", "fred", "george"
name_lengths = {name.title(): len(name) for name in names}
print(name_lengths)
```

```
{'Andrew': 6, 'Ben': 3, 'Charlie': 7, 'Dennis': 6, 'Eric': 4, 'F
red': 4, 'George': 6, 'Hector': 6, 'Irving': 6, 'John': 4}
```

Generator comprehension

- If the comprehension is only suppose to be used once, then just write a generator comprehension

In [9]:

```
for n in (x**2 for x in range(1, 10+1)):
    print(n, end=", ")
```

```
1, 4, 9, 16, 25, 36, 49, 64, 81, 100,
```

Enum

Avoiding magic numbers or strings

- When dealing with data there are often categories that are numbered
- These numbered categories often belong to different domains
- It can be hard to guess that a particular number is a category designator

Defining an Enum type

- Defining an Enum type looks a lot like defining a named tuple

In [10]:
```
from enum import Enum

class MaritalStatus(Enum):
    UNMARRIED = 1
    ENGAGED = 2
    MARRIED = 3
    DIVORCED = 4
    WIDOWED = 5
    UNKNOWN = 6
```

Using Enum values

- When defining a categorial type use the enum type

In [11]:
```
import typing as t

class Person(t.NamedTuple):
    name: str
    age: int
    status: MaritalStatus

andrew = Person("Andrew Anderson", 34, MaritalStatus.DIVORCED)
print(andrew)
```

```
Person(name='Andrew Anderson', age=34, status=<MaritalStatus.DIV
ORCED: 4>)
```

Flags

- At times we need an enum type that can describe more than one status
- For this we have the flag type

In [12]:
```python
from enum import Flag

class CivilStatus(Flag):
    MARRIED = 1
    CHILDREN = 2
    HOMEOWNER = 4
    DOG = 8
    CITIZEN = 16

status = CivilStatus.MARRIED | CivilStatus.DOG | CivilStatus.CITIZEN
print(status)
if CivilStatus.DOG in status:
    print("Dog owner")
```

```
CivilStatus.MARRIED|DOG|CITIZEN
Dog owner
```