Python: Advanced stuff

Table of contents

1. Files

1.1. Filenames and paths

Files are organized into directories

Every running program has a current working directory, which is the default directory for most operations

```
>>> import os
>>> os.getcwd()
'/home/fglmtt/github.com/admin'
```

Note that

- `os` provides functions for working with files and directories
- `getcwd` returns the current working directory
- `/home/fglmtt/github.com/admin` is a path

Absolute v. relative paths

- `/home/fglmtt/github.com/admin` is an absolute path
- `data/words.txt` is a relative path

Function	Description
`listdir`	List the content of a directory
`path.exists`	Check whether a file or directory exists
`path.isdir`	Check whether a path refers to a directory
`path.isfile`	Check whether a path refers to a file
`path.join`	Join directories and filenames into a path

`path` is a submodule of `os`

```
>>> import os
>>> os.path.exists('code')
True
>>> os.path.isfile('code')
```

```
False
>>> os.path.isdir('code')
True
>>> os.listdir('code')
['text_inspector.py', 'spelling_bee.py', 'uses_any.py', 'data']
>>> os.path.join(os.getcwd(), 'code', 'data', 'words.txt')
'/home/fglmtt/github.com/admin/code/data/words.txt'
```

1.2. F-strings

Strings that

- Have the letter `f` before the opening quotation mark
- Contain one or more expressions in curly braces
    - Expressions are automatically converted to `str`

```
>>> d = {'one': 1}
>>> l = [1, 2, 3]
>>> f'dict: {d}, list: {l}, sum list: {sum(l)}'
"dict: {'one': 1}, list: [1, 2, 3], sum list: 6"
```

```
>>> d = {'one': 1}
>>> l = [1, 2, 3]
>>> writer = open('deleteme.txt', 'w')
>>> writer.write(f'dict: {d}, list: {l}\n')
34
>>> writer.write(f'sum list: {sum(l)}\n')
12
>>> writer.close()
>>> print(open('deleteme.txt').read())
dict: {'one': 1}, list: [1, 2, 3]
sum list: 6
```

```
>>> d = {'one': 1}
>>> l = [1, 2, 3]
>>> writer = open('deleteme.txt', 'w')
>>> writer.write("dict: " + d + " ,list: " + l + "\n")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "dict") to str
```

```
>>> writer.write("dict: " + str(d) + " ,list: " + str(l) + "\n")
34
```

1.3. Serializing and deserializing files

Programs read and write files to store configuration data

Configuration data

- Specifies what a program should do and how
- Is typically written either in YAML or JSON

`yaml` and `json` provide functions to (de)serialize data accordingly

---

```
>>> writer = open('config.json', 'w')
>>> config = {'config_1': 1, 'config_2': 2}
>>> writer.write(json.dumps(config))
30
>>> writer.close()
>>> json.loads(open('config.json').read())
{'config_1': 1, 'config_2': 2}
```

```
>>> writer = open('config.json', 'w')
>>> config = {'config_1': 1, 'config_2': 2}
>>> writer.write(str(config))
30
>>> writer.close()
>>> json.loads(open('config.json').read())
Traceback (most recent call last):
...
json.decoder.JSONDecodeError: Expecting property name enclosed in double
quotes: line 1 column 2 (char 1)
```

```
>>> print(open('config.json').read())
{'config_1': 1, 'config_2': 2}
```

Given that

- `str(config)` does not serialize `config` correctly
- `json.loads` expects `"` but gets `'`

`json.loads` raises `json.decoder.JSONDecodeError`

```
>>> writer = open('config.json', 'w')
>>> config = {'config_1': 1, 'config_2': 2}
>>> writer.write(json.dumps(config))
30
>>> writer.close()
>>> print(open('config.json').read())
{"config_1": 1, "config_2": 2}
```

1.4. Walking directories

`walk.py` (see here):

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print(path)
        elif os.path.isdir(path):
            walk(path)
```

The output should be equivalent to

```
$ ls -R
```

2. Classes

2.1. Class definitions

A class is a programmer-defined type

```
class Time:
    """Represent a time of day"""
```

- The header indicates that the new class is called `Time`
- The body is just a docstring

A class definition creates a class object, which is like a factory for creating objects

To create a `Time` object, just call `Time` as if it were a function

```
>>> class Time:
...     """Represent a time of day"""
...
>>> lunch = Time()
>>> type(lunch)
<class '__main__.Time'>
>>> print(lunch)
<__main__.Time object at 0x104f78650>
```

`lunch` is an instance of the class `Time`

2.2. Copying objects

Objects are passed to functions as aliases

The `copy` module provides the `copy` function that can duplicate any object

```
>>> import copy
>>> class Time:
...     """Represent a time of day"""
...
>>> lunch = Time()
>>> dinner = copy.copy(lunch)
>>> lunch is dinner
False
```

```
>>> import copy
>>> class Time:
...     """Represent a time of day"""
...
>>> lunch = Time()
>>> dinner = copy.copy(lunch)
>>> lunch is dinner
False
>>> lunch == dinner
False
```

For programmer-defined classes, the default behavior of `==` is the same as `is`, i.e., `==` checks identity, not equivalence

2.2.1. Shallow copy v. deep copy

The `copy` module provides two functions for copying objects

Function	Description
`copy`	Copy the object but not the objects it contains (aka shallow copy)
`deepcopy`	Copy the object, the objects it refers to, the objects they refer to, and so on (aka deep copy)

2.3. Pure functions

Functions

- Whose return values are identical for identical arguments
- Have no side effects (e.g., modification of mutable reference arguments)

Anything that can be done with impure functions can also be done with pure functions

As a rule of thumb

- Write pure functions whenever it is reasonable
- Resort to impure functions only if there is an advantage

The following is **not** idiomatic code and serves only as an example

`custom_time_v1.py` (see here):

```
import copy

class Time:
    """Represent a time of day"""

def print_time(time):
    s = (
        f"{time.hour:02d}:"
        f"{time.minute:02d}:"
        f"{time.second:02d}"
    )
    print(s)

def make_time(hour, minute, second):
    time = Time()
    time.hour = hour
    time.minute = minute
    time.second = second
    return time

def increment_time(time, hours, minutes, seconds):
```

```
    time.hour += hours
    time.minute += minutes
    time.second += seconds

    carry, time.second = divmod(time.second, 60)
    carry, time.minute = divmod(
        time.minute + carry, 60
    )
    carry, time.hour = divmod(time.hour + carry, 60)

def add_time(time, hours, minutes, seconds):
    total = copy.copy(time)
    increment_time(
        total,
        hours,
        minutes,
        seconds
    )
    return total
```

```
>>> import custom_time_v1 as custom_time
>>> start = custom_time.make_time(9, 20, 0)
>>> custom_time.print_time(start)
09:20:00
>>> custom_time.increment_time(start, 1, 0, 0)
>>> custom_time.print_time(start)
10:20:00
```

`increment_time` is an impure function

```
>>> import custom_time_v1 as custom_time
>>> start = custom_time.make_time(9, 20, 0)
>>> end = custom_time.add_time(start, 1, 0, 0)
>>> custom_time.print_time(start)
09:20:00
>>> custom_time.print_time(end)
10:20:00
>>> start is end
False
```

`add_time` is a pure function

Built-in function	Description
`type(obj)`	Return the type of `obj`
`isinstance(obj, class)`	Check whether `obj` is instance of `class`
`hasattr(obj, str)`	Check whether `str` is an attribute of `obj`
`vars(obj)`	Return the attributes of `obj`

```
>>> import custom_time_v1 as custom_time
>>> type(custom_time.Time)
<class 'type'>
>>> start = custom_time.make_time(9, 20, 0)
>>> type(start)
<class 'custom_time_v1.Time'>
>>> isinstance(start, custom_time.Time)
True
>>> hasattr(start, 'duration')
False
>>> vars(start)
{'hour': 9, 'minute': 20, 'second': 0}
```

3. Methods

3.1. Method definitions

A method is a function that is defined inside a class definition

```
class Time:
    """Represents the time of day"""

    def print_time(self):
        s = (
            f"{self.hour:02d}:"
            f"{self.minute:02d}:"
            f"{self.second:02d}"
        )
        print(s)

def make_time(hour, minute, second):
    time = Time()
    time.hour = hour
    time.minute = minute
    time.second = second
    return time
```

- `print_time` is an instance method
- `make_time` is a function

3.2. Instance methods

Methods that must be invoked with an object as receiver

Two ways to call `print_time`

Suppose `start` is an instance of `Time`

Function syntax (less common):

```
Time.print_time(start)
```

- `start` is the receiver and passed as a parameter

Method syntax (idiomatic):

```
start.print_time()
```

- `start` is the object the method is invoked on (receiver)
- The receiver is assigned to the first parameter
    - `self` refers to `start` inside the method `print_time`

3.3. Static methods

Methods that can be invoked without an object as receiver

```
class Time:
    """Represents the time of day"""

    def print_time(self):
        s = (
            f"{self.hour:02d}:"
            f"{self.minute:02d}:"
            f"{self.second:02d}"
        )
        print(s)

    def int_to_time(seconds):
        minute, second = divmod(seconds, 60)
        hour, minute = divmod(minute, 60)
        return make_time(hour, minute, second)
```

```
def make_time(hour, minute, second):
    time = Time()
    time.hour = hour
    time.minute = minute
    time.second = second
    return time
```

- `print_time` is an instance method
- `int_to_time` is a static method
- `make_time` is a function

Static methods do not have `self` as a parameter

Static methods are invoked on the class object

```
start = Time.int_to_time(34800)
```

3.4. Special methods

Change the way operators and some functions work with an object

Special method	Description
`__init__`	Initialize the attribute(s) of an object. The arguments are those passed to the class object
`__str__`	Return a printable representation of an object. The return value must be a `str`. Called by `str()` and `print()`
`__add__`	Implement the `+` operator
`__eq__`	Implement the `==` operator

3.4.1. Initializing objects

`custom_time_v2.py` (see here, this is idiomatic code):

```
class Time:
    """Represents the time of day"""

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

```
>>> import custom_time_v2 as custom_time
>>> start = custom_time.Time(9, 20, 0)
>>> start
<custom_time_v2.Time object at 0x100bc7110>
>>> start = custom_time.Time()
>>> start
<custom_time_v2.Time object at 0x100bf43e0>
```

3.4.2. Printing objects

```
class Time:
    """Represents the time of day"""

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def __str__(self):
        return (
            f"{self.hour:02d}:"
            f"{self.minute:02d}:"
            f"{self.second:02d}"
        )
```

```
>>> import custom_time_v2 as custom_time
>>> start = custom_time.Time(9, 20, 0)
>>> custom_time.Time.__str__(start)
'09:20:00'
>>> start.__str__()
'09:20:00'
```

```
>>> import custom_time_v2 as custom_time
>>> start = custom_time.Time(9, 20, 0)
>>> str(start)
'09:20:00'
>>> print(start)
09:20:00
```

3.4.3. Overloading operators

Operator overloading is the process of using special methods to change the way operators work with programmer-defined types

```
class Time:
    """Represents the time of day"""

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def __str__(self):
        return (
            f"{self.hour:02d}:"
            f"{self.minute:02d}:"
            f"{self.second:02d}"
        )

    def int_to_time(seconds):
        minute, second = divmod(seconds, 60)
        hour, minute = divmod(minute, 60)
        return Time(hour, minute, second)

    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds

    def __add__(self, other):
        seconds = (
            self.time_to_int() +
            other.time_to_int()
        )
        return Time.int_to_time(seconds)

    def __eq__(self, other):
        return (
            self.hour == other.hour
            and self.minute == other.minute
            and self.second == other.second
        )
```

```
import custom_time_v2 as custom_time
>>> start = custom_time.Time(hour=1)
>>> end = custom_time.Time(hour=1, minute=30)
>>> str(end)
```

```
'01:30:00'
>>> print(start + end)
02:30:00
```

```
import custom_time_v2 as custom_time
>>> start = custom_time.Time(hour=1)
>>> end = custom_time.Time(hour=1, minute=30)
>>> t = custom_time.Time(1, 0, 0)
>>> start == t
True
>>> start is t
False
```

4. Inheritance

4.1. Class variables

Variables defined inside a class definition, but not inside any method

`poker.py` (see here):

```
class Card:
    """Represent a standard playing card"""

    suits = [
        'Clubs', 'Diamonds', 'Hearts', 'Spades'
    ]
    ranks = [
        None, 'Ace', '2', '3', '4', '5',
        '6', '7','8', '9', '10',
        'Jack', 'Queen', 'King', 'Ace'
    ]
```

Class variables are associated with the class, rather than instances

```
>>> import poker
>>> poker.Card.suits
['Clubs', 'Diamonds', 'Hearts', 'Spades']
>>> poker.Card.ranks[11:]
['Jack', 'Queen', 'King', 'Ace']
```

```
class Card:
    """Represent a standard playing card"""

    suits = [
        'Clubs', 'Diamonds', 'Hearts', 'Spades'
    ]
    ranks = [
        None, 'Ace', '2', '3', '4', '5',
        '6', '7','8', '9', '10',
        'Jack', 'Queen', 'King', 'Ace'
    ]

    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank
```

- `suits` and `ranks` are class variables
- `suit` and `rank` are instance attributes

```
>>> import poker
>>> queen = poker.Card(1, 12)
>>> queen.suit, queen.rank
(1, 12)
>>> queen.suits
['Clubs', 'Diamonds', 'Hearts', 'Spades']
```

It is also legal to use the instance to access class variables

4.2. Totally ordered set of objects

A set of objects is totally ordered if

- Any two elements can be compared
- The results are consistent

Special method	Description
`__eq__`	Implement the `==` operator
`__ne__`	Implement the `!=` operator. If `__ne__` does not exist, the interpreter invokes `__eq__` and inverts the result
`__lt__`	Implement the `<` operator
`__gt__`	Implement the `>` operator. If `__gt__` does not exist, the interpreter

Special method	Description
	invokes `__lt__` and inverts the result
`__le__`	Implement the `<=` operator
`__ge__`	Implement the `>=` operator. If `__ge__` does not exist, the interpreter invokes `__le__` and inverts the result

---

The following methods make `Card` objects totally ordered

```
class Card:
    """Represent a standard playing card"""

    suits = [
        'Clubs', 'Diamonds', 'Hearts', 'Spades'
    ]
    ranks = [
        None, 'Ace', '2', '3', '4', '5',
        '6', '7','8', '9', '10',
        'Jack', 'Queen', 'King', 'Ace'
    ]

    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank

    def __eq__(self, other):
        return (
            self.suit == other.suit and
            self.rank == other.rank
        )

    def to_tuple(self):
        return (self.suit, self.rank)

    def __lt__(self, other):
        # Suit is more important than rank
        # If suits are equal, compare ranks
        return self.to_tuple() < other.to_tuple()

    def __le__(self, other):
        return self.to_tuple() <= other.to_tuple()
```

4.3. Delegation

A method passing responsibility to another one

`poker.py` (see here):

```
import random

class Deck:
    """Represent a deck of cards"""

    def __init__(self, cards):
        self.cards = cards

    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)

    def make_cards():
        cards = []
        for suit in range(4):
            # Aces outrank kings
            for rank in range(2, 15):
                card = Card(suit, rank)
                cards.append(card)
        return cards

    def shuffle(self):
        random.shuffle(self.cards)

    def sort(self):
        self.cards.sort()

    def take_card(self):
        return self.cards.pop()

    def put_card(self, card):
        self.cards.append(card)

    def move_cards(self, other, num):
        for _ in range(num):
            card = self.take_card()
            other.put_card(card)
```

```
>>> import poker
>>> cards = poker.Deck.make_cards()
```

```
>>> deck = poker.Deck(cards)
>>> type(deck)
<class 'poker.Deck'>
>>> type(deck.cards)
<class 'list'>
>>> len(deck.cards)
52
```

```
>>> import poker
>>> cards = poker.Deck.make_cards()
>>> deck = poker.Deck(cards)
>>> deck.shuffle()
>>> for card in deck.cards[:3]:
...     print(card)
...
Ace of Hearts
King of Diamonds
8 of Hearts
>>> deck.sort()
>>> for card in deck.cards[:3]:
...     print(card)
...
2 of Clubs
3 of Clubs
4 of Clubs
```

- `Deck.sort` does not do any work except passing responsibility
- `list.sort` uses the `__lt__` method to sort `Card` objects

4.4. Parents and children

Inheritance is the ability to define a new class that is a modified version of a previously defined class

`poker.py` (see here):

```
class Hand(Deck):
    """Represent a hand of a player"""

    def __init__(self, label=''):
        self.label = label
        self.cards = []
```

`Hand` inherits from `Deck`

- `Hand` is a child class of `Deck`, which is the parent class
- `Hand` objects can access methods defined in `Deck`

`Hand` overrides `Deck.__init__`

- The interpreter invokes `Hand.__init__` for `Hand` objects

```
>>> import poker
>>> cards = poker.Deck.make_cards()
>>> deck = poker.Deck(cards)
>>> card = deck.take_card()
>>> hand = poker.Hand('player 1')
>>> hand.put_card(card)
>>> print(hand)
Ace of Spades
```

4.4.1. Polymorphism

A method or operator that works with multiple types of objects

```
>>> import poker
>>> cards = poker.Deck.make_cards()
>>> deck = poker.Deck(cards)
>>> hand = poker.Hand('player 1')
>>> deck.move_cards(hand, 2)
>>> print(hand)
Ace of Spades
King of Spades
```

`Deck.move_card` is polymorphic as it works with `Deck` and `Hand`

4.4.2. Liskov substitution principle

> 1. Instances of the child class should have all of the attributes of the parent class, but they can have additional ones
> 2. The child class should have all of the methods of the parent class, but it can have additional ones
> 3. If a child class overrides a method from the parent class, the new one should take the same parameters and return a compatible result
>
> -- Barbara Liskov

If you follow these rules, any method designed to work with instances of a parent class will also work with children instances

5. Extras

5.1. Sets

Collections of unique elements

```
>>> s = set()
>>> type(s)
<class 'set'>
>>> s = set('hello')
>>> s
{'h', 'e', 'l', 'o'}
```

5.2. Counter

Like sets, except that a `Counter` keeps track of how many times elements appear

```
>>> import collections
>>> c = collections.Counter('hello')
>>> c
Counter({'l': 2, 'h': 1, 'e': 1, 'o': 1})
>>> c['l']
2
```

Keys must be hashable

5.3. Defaultdict

Like dictionaries, except that a `defaultdict` automatically generates keys that do not exist

```
>>> import collections
>>> d = collections.defaultdict(list)
>>> d
defaultdict(<class 'list'>, {})
>>> d['zero']
[]
>>> d
defaultdict(<class 'list'>, {'zero': []})
```

`list` works as a factory

5.4. Conditional expressions

Expressions that use conditionals to select one of two values

`factorial` implemented with a conditional statement:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

`factorial` implemented with a conditional expression:

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)
```

In general, conditional statements can be replaced with conditional expressions if both branches contain

- A single expression
- No statements

5.5. List comprehensions

Concise ways to loop through a sequence and create a list

Traditional loop:

```
def capitalize_title(title):
    t = []
    for word in title.split():
        t.append(word.capitalize())
    return ' '.join(t)
```

With a list comprehension:

```
def capitalize_title(title):
    t = [
        word.capitalize() for word in title.split()
    ]
    return ' '.join(t)
```

- Brackets indicate that the result is a list
- The expression inside the brackets specifies the list elements

- `for` indicates what sequence is to loop through

Glossary

Term	Meaning
Absolute path	A path that does not depend on the current directory
Child class	A class that inherits from another class
Class	A programmer-defined type. A class definition creates a new class object
Class object	An object that represents a class. A class object is the result of a class definition
Class variable	A variable defined inside a class definition, but not inside any method
Configuration data	Data, often stored in a file, that specifies what a program should do and how
Conditional expression	An expression that uses a conditional to select one of two values
Current working directory	The default directory used by a program unless another directory is specified
Deep copy	A copy operation that also copies nested objects
Delegation	When one method passes responsibility to another method to do most or all of the work
Deserialization	Converting a string to an object
Directory	A collection of files and other directories
Factory function	A function used to create objects, often passed as a parameter to a function
F-string	A string that has the letter `f` before the opening quotation mark, and contains one or more expressions in curly braces
Inheritance	The ability to define a new class that is a modified version of a previously defined class
Instance	An object that belongs to a class
Instance method	A method that must be invoked with an object as receiver
List comprehension	A concise way to loop through a sequence and create a list
Method	A function that is defined inside a class definition
Operator overloading	The process of using special methods to change the way operators work with programmer-defined types
Parent class	A class that is inherited from
Path	A string that specifies a sequence of directories, often leading to a file

Term	Meaning
Polimorphism	The ability of a method or operator to work with multiple types of objects
Pure function	A function (i) whose return values are identical for identical arguments and (ii) has no side effects
Receiver	The object a method is invoked on
Relative path	A path that starts from the current working directory, or some other specified directory
Serialization	Converting an object to a string
Shallow copy	A copy operation that does not copy nested objects
Special method	A method that changes the way operators and some functions work with an object
Static method	A method that can be invoked without an object as receiver
Totally ordered	A set of objects is totally ordered if any two elements can be compared and the results are consistent

Bibliography

Author	Title	Year
Downey, A.	Think Python	2024

Licenses