Secure Software Design

UNIVERSITÀ DELLA CALABRIA

il Campus per eccellenza

Advanced tests for Python

Mario Alviano

Outline

- Type hints
- Dataclasses
- Type and value validation
- Fixtures
- Mocks and patches

For these slides you may refer Chapter 3 of
Clean Architectures in Python by *Leonardo Giordani,*

https://leanpub.com/clean-architectures-in-python

Running example

Car Dealer

We want a TUI for storing cars and motos (plate, producer, model, price)

A discount is applied as we did in Java

We limit to add and remove, and sort by producer and price

Third-party modules

- A simple approach to validate types and values can be to use IF statements, and raise exceptions
- We better reuse third-party modules and new features of Python
- Type hints can be used by a static analyser
  - But also for dynamic validation with typeguard https://typeguard.readthedocs.io/en/latest/userguide.html
- Dataclasses are convenient to define classes from annotations
  - https://docs.python.org/3/library/dataclasses.html
  - dataclass-type-validator automates type validation of dataclasses https://github.com/levii/dataclass-type-validator
- For futher validation we can use valid8
  - https://smarie.github.io/python-valid8/

domain.py

```python
13  @typechecked
14  @dataclass(frozen=True, order=True)
15  class Plate:
16      value: str
17
18      def __post_init__(self):
19          validate_dataclass(self)
20          validate('value', self.value, min_len=5, max_len=10, custom=pattern(r'[0-9A-Z]*'))
21
22      def __str__(self):
23          return self.value
```

Enforces type validation on all methods with type hints

Generates __init__, __eq__, and other methods, inhibits changes, generates __lt__ and other methods

Annotation: dataclass Plate has a field value of type str

This method is called after __init__ to add further validation

validate() is provided by valid8

Custom __str__

Custom validator for valid8 to enforce regex

Wrapper for dataclass-type-validator to make exceptions uniform

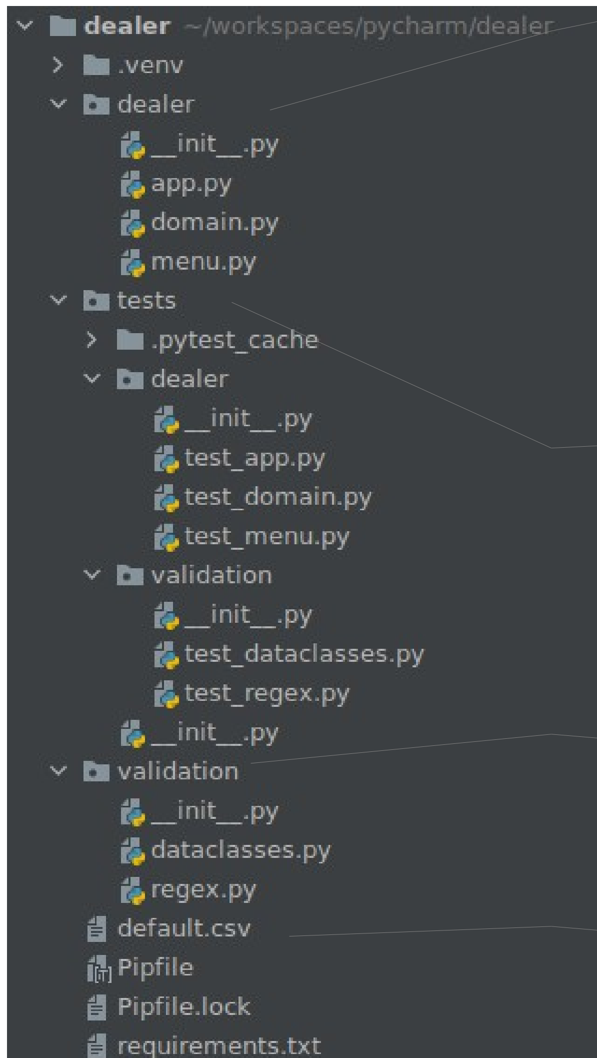
dataclasses.py

```python
4  def validate_dataclass(data):
5      try:
6          dataclass_type_validator(data)
7      except TypeValidationError as e:
8          raise TypeError(e)
```

regex.py

```python
7   @typechecked
8   def pattern(regex: str) -> Callable[[str], bool]:
9       r = re.compile(regex)
10      def res(value):
11          return bool(r.fullmatch(value))
12      res.__name__ = f'pattern({regex})'
13      return res
```

Project structure



Project modules

domain classes,
generic menu classes
(could be a third-party module),
I/O classes of the app

Tests here, the folder reflects the structure of all other folders and modules in the project

One test_* file for every module of the project

Utilities to ease validation
(could be a third-party-module)

Data are automatically loaded from and saved to this file

test_domain.py

```
7   def test_plate_format():
8       wrong_values = ['', 'abcde', 'AA000bb', 'A'*11]
9       for value in wrong_values:
10          with pytest.raises(ValidationError):
11              Plate(value)
12
13      correct_values = ['CA220NE', 'ABCDE', 'A'*10]
14      for value in correct_values:
15          assert Plate(value).value == value
16
17
18  def test_plate_str():
19      for value in ['CA220NE', 'ABCDE', 'A'*10]:
20          assert str(Plate(value)) == value
```

Test for wrong values,
we expect exceptions

Test for correct values,
we expect to read back the values

Try to cover all lines of code
with your tests

Discount in thousands

Test boundaries, string representation, and any non-trivial computation that the class must do

test_domain.py

```python
def test_discount_cannot_be_negative():
    with pytest.raises(ValidationError):
        Discount(-1)


def test_discount_cannot_be_greater_than_1000():
    with pytest.raises(ValidationError):
        Discount(1001)


def test_discount_str():
    assert str(Discount(100)) == '10.0%'


def test_discount_apply_to_int():
    assert Discount(100).apply(90) == 81
```

Use type hints as much as possible

PyCharm uses them to help you code

If we have to fail, better to fail soon

typechecked will let us fail!

domain.py

```python
@typechecked
@dataclass(frozen=True, order=True)
class Discount:
    value_in_thousands: int

    def __post_init__(self):
        validate_dataclass(self)
        validate('value_in_thousands', self.value_in_thousands, min_value=0, max_value=1000)

    def __str__(self):
        return f'{self.value_in_thousands // 10}.{self.value_in_thousands % 10}%'

    def apply(self, value: int) -> int:
        return value * (1000 - self.value_in_thousands) // 1000
```

Price, never simple

Price expressed in cents, we must be precise

Create Price from euro and possibly cents

Let’s disable the constructor to avoid confusion:
How? It cannot be private in Python!

Provide parse methods for non-string domain primitives

```
73  def test_price_no_init():
74      with pytest.raises(ValidationError):
75          Price(1)
76
77
78  def test_price_cannot_be_negative():
79      with pytest.raises(ValidationError):
80          Price.create(-1, 0)
81
82
83  def test_price_no_cents():
84      assert Price.create(1, 0) == Price.create(1)
85
86
87  def test_price_parse():
88      assert Price.parse('10.20') == Price.create(10, 20)
89
90
91  def test_price_str():
92      assert str(Price.create(9, 99)) == '9.99'
```

```
95  def test_price_euro():
96      assert Price.create(11, 22).euro == 11
97
98
99  def test_price_cents():
100     assert Price.create(11, 22).cents == 22
101
102
103 def test_price_add():
104     assert Price.create(9, 99).add(Price.create(0, 1)) == Price.create(10)
105
106
107 def test_price_apply_discount():
108     assert Price.create(100).apply_discount(Discount(100)) == Price.create(90)
```

We will use @property
to define euro and cents

Non-trivial calculation
always comes with tests

```
68  @typechecked
69  @dataclass(frozen=True, order=True)
70  class Price:
71      value_in_cents: int
72      create_key: InitVar[Any] = field(default=None)
73
74      __create_key = object()
75      __max_value = 100000000000 - 1
76      __parse_pattern = re.compile(r'(?P<euro>\d{0,11})(?:\.(?P<cents>\d{2}))?')
77
78      def __post_init__(self, create_key):
79          validate('create_key', create_key, equals=self.__create_key)
80          validate_dataclass(self)
81          validate('value_in_cents', self.value_in_cents, min_value=0, max_value=self.__max_value)
82
83      def __str__(self):
84          return f'{self.value_in_cents // 100}.{self.value_in_cents % 100:02}'
85
86      @staticmethod
87      def create(euro: int, cents: int=0) -> 'Price':
88          validate('euro', euro, min_value=0, max_value=Price.__max_value // 100)
89          validate('cents', cents, min_value=0, max_value=99)
90          return Price(euro * 100 + cents, Price.__create_key)
```

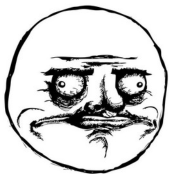
create_key is an extra argument of __init__ (and __post_init__)

__create_key is a private class variable
(actually, the name is randomized)

We pretend to have
create_key == __create_key

This is how we make
a private constructor
in Python

Since __create_key is not accessible from outsite Price,
the constructor of Price can be called only inside Price



Use a string as type hint if the type is not-yet-fully-defined

Do it for methods that return instances of the class

If you do it somewhere else,
likely you have circular dependencies

```
92      @staticmethod
93      def parse(value: str) -> 'Price':
94          m = Price.__parse_pattern.fullmatch(value)
95          validate('value', m)
96          euro = m.group('euro')
97          cents = m.group('cents') if m.group('cents') else 0
98          return Price.create(int(euro), int(cents))
99
100     @property
101     def cents(self) -> int:
102         return self.value_in_cents % 100
103
104     @property
105     def euro(self) -> int:
106         return self.value_in_cents // 100
107
108     def add(self, other: 'Price') -> 'Price':
109         return Price(self.value_in_cents + other.value_in_cents, self.__create_key)
110
111     def apply_discount(self, discount: Discount) -> 'Price':
112         return Price(discount.apply(self.value_in_cents), self.__create_key)
```

A property is a self-only method
that we want to access without parenthesis

We can also specify a setter,
but not for a frozen dataclasse

Price knows Discount, but
Discount doesn't know Price

Unless there's a valid reason,
avoid circular dependencies

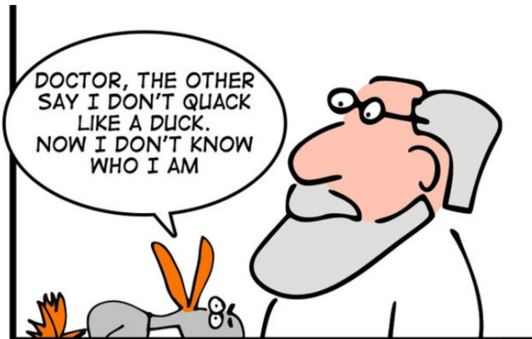
Cars and motos, should we bind them?

Car and Moto will have the same fields, and most of their logic is common

We may opt for an hierarchy, but it’d be justified more technically than from a domain point of view

As we will see, not a lot of code

KISS



Duck typing

If it walks like a duck and it quacks like a duck, then it must be a duck

We can define fixtures for objects that we want to use in many tests

```
111  @pytest.fixture
112  def cars():
113      return [
114          Car(Plate('AB123CD'), Producer('Car Producer'), Model('Model'), Price.create(100)),
115          Car(Plate('AB123CE'), Producer('Car Producer'), Model('Model'), Price.create(11000)),
116          Car(Plate('AB123CF'), Producer('Car Producer'), Model('Model'), Price.create(21000)),
117      ]
118
119
120  def test_car_type_is_car(cars):
121      assert cars[0].type == 'Car'
122
123
124  def test_car_final_price(cars):
125      assert cars[0].final_price == cars[0].price.apply_discount(Discount(50))
126      assert cars[1].final_price == cars[1].price.apply_discount(Discount(100))
127      assert cars[2].final_price == cars[2].price
```

Pass the name of the fixture as an argument to access the object returned by the fixture

As many times you like

```python
116  @typechecked
117  @dataclass(frozen=True, order=True)
118  class Car:
119      plate: Plate
120      producer: Producer
121      model: Model
122      price: Price
123
124      @property
125      def type(self) -> str:
126          return 'Car'
127
128      @property
129      def final_price(self) -> Price:
130          if self.price <= Price.create(10000):
131              return self.price.apply_discount(Discount(50))
132          if self.price <= Price.create(20000):
133              return self.price.apply_discount(Discount(100))
134          return self.price
```

It may help to save to file

We could also derive it from the name of the class, but... KISS

Price with discount accessible with a different method

Avoid ambiguities!

We can use arithmetic comparisons because Price is @dataclass(order=True)

Moto is essentially the same, at least at the moment

```python
@pytest.fixture
def motos():
    return [
        Moto(Plate('AB123CD'), Producer('Moto Producer'), Model('Model'), Price.create(100)),
        Moto(Plate('AB123CE'), Producer('Moto Producer'), Model('Model'), Price.create(8000)),
        Moto(Plate('AB123CF'), Producer('Moto Producer'), Model('Model'), Price.create(16000)),
    ]


def test_moto_type_is_moto(motos):
    assert motos[0].type == 'Moto'


def test_moto_final_price(motos):
    assert motos[0].final_price == motos[0].price.apply_discount(Discount(30))
    assert motos[1].final_price == motos[1].price.apply_discount(Discount(75))
    assert motos[2].final_price == motos[2].price
```

```python
@typechecked
@dataclass(frozen=True, order=True)
class Moto:
    plate: Plate
    producer: Producer
    model: Model
    price: Price

    @property
    def type(self) -> str:
        return 'Moto'

    @property
    def final_price(self) -> Price:
        if self.price <= Price.create(7000):
            return self.price.apply_discount(Discount(30))
        if self.price <= Price.create(15000):
            return self.price.apply_discount(Discount(75))
        return self.price
```

Don't bind concepts that can stay separated

The Dealer must provide all functionalities, but must not care about I/O operations

```python
def test_dealer_add_car(cars):
    dealer = Dealer()
    size = 0
    for car in cars:
        dealer.add_car(car)
        size += 1
        assert dealer.vehicles() == size
        assert dealer.vehicle(size - 1) == car


def test_dealer_add_motos(motos):
    dealer = Dealer()
    size = 0
    for moto in motos:
        dealer.add_moto(moto)
        size += 1
        assert dealer.vehicles() == size
        assert dealer.vehicle(size - 1) == moto
```

```python
def test_dealer_remove_vehicle(cars, motos):
    dealer = Dealer()
    for car in cars:
        dealer.add_car(car)
    for moto in motos:
        dealer.add_moto(moto)

    dealer.remove_vehicle(0)
    assert dealer.vehicle(0) == cars[1]

    with pytest.raises(ValidationError):
        dealer.remove_vehicle(-1)
    with pytest.raises(ValidationError):
        dealer.remove_vehicle(dealer.vehicles())

    while dealer.vehicles():
        dealer.remove_vehicle(0)
    assert dealer.vehicles() == 0
```

```python
def test_dealer_sort_by_producer(cars, motos):
    dealer = Dealer()
    dealer.add_moto(motos[0])
    dealer.add_car(cars[0])
    dealer.sort_by_producer()
    assert dealer.vehicle(0) == cars[0]


def test_dealer_sort_by_price(cars):
    dealer = Dealer()
    dealer.add_car(cars[1])
    dealer.add_car(cars[0])
    dealer.sort_by_price()
    assert dealer.vehicle(0) == cars[0]
```

Test all the functionalities

TDD: test first, code later

Not a religion, it’s also OK to test and code in parallel

```python
158    @typechecked
159    @dataclass(frozen=True)
160    class Dealer:
161        __vehicles: List[Union[Car, Moto]] = field(default_factory=list, init=False)
162
163        def vehicles(self) -> int:
164            return len(self.__vehicles)
165
166        def vehicle(self, index: int) -> Union[Car, Moto]:
167            validate('index', index, min_value=0, max_value=self.vehicles() - 1)
168            return self.__vehicles[index]
169
170        def add_car(self, car: Car) -> None:
171            self.__vehicles.append(car)
172
173        def add_moto(self, moto: Moto) -> None:
174            self.__vehicles.append(moto)
175
176        def remove_vehicle(self, index: int) -> None:
177            validate('index', index, min_value=0, max_value=self.vehicles() - 1)
178            del self.__vehicles[index]
179
180        def sort_by_producer(self) -> None:
181            self.__vehicles.sort(key=lambda x: x.producer)
182
183        def sort_by_price(self) -> None:
184            self.__vehicles.sort(key=lambda x: x.price)
```

Union[Car, Moto] means
either a Car or a Moto

default_factory is
a method to call
to create the default value

init=False is to exclude
this field from __init__

Nothing special
in the remainder

Menu: Description and Key

```
11  @typechecked
12  @dataclass(order=True, frozen=True)
13  class Description:
14      value: str
15
16      def __post_init__(self):
17          validate_dataclass(self)
18          validate('Description.value', self.value, min_len=1, max_len=1000, custom=pattern(r'[0-9A-Za-z ;.,_-]*'))
19
20      def __str__(self):
21          return self.value
```

```
24  @typechecked
25  @dataclass(order=True, frozen=True)
26  class Key:
27      value: str
28
29      def __post_init__(self):
30          validate_dataclass(self)
31          validate('Key.value', self.value, min_len=1, max_len=10, custom=pattern(r'[0-9A-Za-z_-]*'))
32
33      def __str__(self):
34          return self.value
```

Menu: Entry

```python
@typechecked
@dataclass(frozen=True)
class Entry:
    key: Key
    description: Description
    on_selected: Callable[[], None] = field(default=lambda: None)
    is_exit: bool = field(default=False)

    def __post_init__(self):
        validate_dataclass(self)

    @staticmethod
    def create(key: str, description: str, on_selected: Callable[[], None]=lambda: None, is_exit: bool=False) -> 'Entry':
        return Entry(Key(key), Description(description), on_selected, is_exit)
```

How can we check if on_selected work?

We have to simulate a call, and check that the call was actually completed

In Python this is usually done by a Mock

Mock is an object on which essentially all methods can be called

Calls are recorded, and can be later checked

```
51 def test_entry_on_selected():
52     mocked_on_selected = Mock()
53     entry = Entry(Key('1'), Description('Say hi'), on_selected=lambda: mocked_on_selected())
54     entry.on_selected()
55     mocked_on_selected.assert_called_once()
```

Let’s use the __call__ dunder method

We could also use mocked_on_select.foo()

Simulate a call to entry.on_selected

Check that the mocked method was actually called

We can also check call arguments

We can also mock global objects, objects declared somewhere else

For this purpose we use patches

This is the name of the patched object

```
@patch('builtins.print')
def test_entry_on_selected(mocked_print):
    entry = Entry(Key('1'), Description('Say hi'), on_selected=lambda: print('hi'))
    entry.on_selected()
    assert mocked_print.mock_calls == [call('hi')]
```

We would like to print something when selected

Let’s verify that print was indeed called with argument ‘hi’

The Menu itself

We are going to call this function after printing the description

We exclude __entries and __key2entry from __init__ because their values are implicit

```
53    @typechecked
54    @dataclass(frozen=True)
55    class Menu:
56        description: Description
57        auto_select: Callable[[], None] = field(default=lambda: None)
58        __entries: List[Entry] = field(default_factory=list, repr=False, init=False)
59        __key2entry: Dict[Key, Entry] = field(default_factory=dict, repr=False, init=False)
60        create_key: InitVar[Any] = field(default=None)
61
62        def __post_init__(self, create_key: Any):
63            validate('create_key', create_key, custom=Menu.Builder.is_valid_key)
64            validate_dataclass(self)
65
66        def _add_entry(self, value: Entry, create_key: Any) -> None:
67            validate('create_key', create_key, custom=Menu.Builder.is_valid_key)
68            validate('value.key', value.key, custom=lambda v: v not in self.__key2entry)
69            self.__entries.append(value)
70            self.__key2entry[value.key] = value
71
72        def _has_exit(self) -> bool:
73            return bool(list(filter(lambda e: e.is_exit, self.__entries)))
```

We want a builder, so let's use the create_key pattern to have a private constructor

The builder has the key

The builder will add entries with this protected method

We use create_key to make it callable only by the builder

```python
    def __print(self) -> None:
        length = len(str(self.description))
        fmt = '***{}{}{}***'
        print(fmt.format('*', '*' * length, '*'))
        print(fmt.format(' ', self.description.value, ' '))
        print(fmt.format('*', '*' * length, '*'))
        self.auto_select()
        for entry in self.__entries:
            print(f'{entry.key}:\t{entry.description}')

    def __select_from_input(self) -> bool:
        while True:
            try:
                line = input("? ")
                key = Key(line.strip())
                entry = self.__key2entry[key]
                entry.on_selected()
                return entry.is_exit
            except (KeyError, TypeError, ValueError):
                print('Invalid selection. Please, try again...')

    def run(self) -> None:
        while True:
            self.__print()
            is_exit = self.__select_from_input()
            if is_exit:
                return
```

Print description, call auto_select, and print all entries

Let the user select an entry

Keep asking until a valid choice is given

Menu loop

The Menu.Builder

```
103        @typechecked
104        @dataclass()
105        class Builder:
106            __menu: Optional['Menu']
107            __create_key = object()
108
109            def __init__(self, description: Description, auto_select: Callable[[], None]=lambda: None):
110                self.__menu = Menu(description, auto_select, self.__create_key)
111
112            @staticmethod
113            def is_valid_key(key: Any) -> bool:
114                return key == Menu.Builder.__create_key
115
116            def with_entry(self, value: Entry) -> 'Menu.Builder':
117                validate('menu', self.__menu)
118                self.__menu._add_entry(value, self.__create_key)
119                return self
120
121            def build(self) -> 'Menu':
122                validate('menu', self.__menu)
123                validate('menu.entries', self.__menu._has_exit(), equals=True)
124                res, self.__menu = self.__menu, None
125                return res
```

Do not release the key

Fluent interface

build cannot be called twice

menu must have an exit entry

Use side_effect to list return values of a patched object

```python
@patch('builtins.input', side_effect=['1', '0'])
@patch('builtins.print')
def test_menu_selection_call_on_selected(mocked_print, mocked_input):
    menu = Menu.Builder(Description('a description'))\
        .with_entry(Entry.create('1', 'first entry', on_selected=lambda: print('first entry selected')))\
        .with_entry(Entry.create('0', 'exit', is_exit=True))\
        .build()
    menu.run()
    mocked_print.assert_any_call('first entry selected')
    mocked_input.assert_called()


@patch('builtins.input', side_effect=['-1', '0'])
@patch('builtins.print')
def test_menu_selection_on_wrong_key(mocked_print, mocked_input):
    menu = Menu.Builder(Description('a description'))\
        .with_entry(Entry.create('1', 'first entry', on_selected=lambda: print('first entry selected')))\
        .with_entry(Entry.create('0', 'exit', is_exit=True))\
        .build()
    menu.run()
    mocked_print.assert_any_call('Invalid selection. Please, try again...')
    mocked_input.assert_called()
```

Check that the first entry was indeed selected

Check for mistakes… users will do many!

The App

Fix the menu on construction

Call methods to handle events

```python
class App:
    __filename = Path(__file__).parent.parent / 'default.csv'
    __delimiter = '\t'

    def __init__(self):
        self.__menu = Menu.Builder(Description('LaRusso Auto Group'), auto_select=lambda: self.__print_vehicles())\
            .with_entry(Entry.create('1', 'Add car', on_selected=lambda: self.__add_car()))\
            .with_entry(Entry.create('2', 'Add moto', on_selected=lambda: self.__add_moto()))\
            .with_entry(Entry.create('3', 'Remove vehicle', on_selected=lambda: self.__remove_vehicle()))\
            .with_entry(Entry.create('4', 'Sort by producer', on_selected=lambda: self.__sort_by_producer()))\
            .with_entry(Entry.create('5', 'Sort by price', on_selected=lambda: self.__sort_by_price()))\
            .with_entry(Entry.create('0', 'Exit', on_selected=lambda: print('Bye!'), is_exit=True))\
            .build()
        self.__dealer = Dealer()

    def __print_vehicles(self) -> None:
        print_sep = lambda: print('-' * 100)
        print_sep()
        fmt = '%3s %-10s %-30s %-30s %10s %10s'
        print(fmt % ('#', 'PLATE', 'PRODUCER', 'MODEL', 'PRICE', 'FINAL PR.'))
        print_sep()
        for index in range(self.__dealer.vehicles()):
            vehicle = self.__dealer.vehicle(index)
            print(fmt % (index + 1, vehicle.plate.value, vehicle.producer.value, vehicle.model.value, vehicle.price, vehicle.final_price))
        print_sep()
```

More handlers

```python
def __read_vehicle(self) -> Tuple[Plate, Producer, Model, Price]:
    plate = self.__read('Plate', Plate)
    producer = self.__read('Producer', Producer)
    model = self.__read('Model', Model)
    price = self.__read('Price', Price.parse)
    return plate, producer, model, price
```

```python
def __add_car(self) -> None:
    car = Car(*self.__read_vehicle())
    self.__dealer.add_car(car)
    self.__save()
    print('Car added!')

def __add_moto(self) -> None:
    moto = Moto(*self.__read_vehicle())
    self.__dealer.add_moto(moto)
    self.__save()
    print('Moto added!')
```

```python
@staticmethod
def __read(prompt: str, builder: Callable) -> Any:
    while True:
        try:
            line = input(f'{prompt}: ')
            res = builder(line.strip())
            return res
        except (TypeError, ValueError, ValidationError) as e:
            print(e)
```

```python
def __sort_by_producer(self) -> None:
    self.__dealer.sort_by_producer()
    self.__save()

def __sort_by_price(self) -> None:
    self.__dealer.sort_by_price()
    self.__save()
```

```python
def __remove_vehicle(self) -> None:
    def builder(value: str) -> int:
        validate('value', int(value), min_value=0, max_value=self.__dealer.vehicles())
        return int(value)

    index = self.__read('Index (0 to cancel)', builder)
    if index == 0:
        print('Cancelled!')
        return
    self.__dealer.remove_vehicle(index - 1)
    self.__save()
    print('Vehicle removed!')
```

Load and save

```python
84    def __load(self) -> None:
85        if not Path(self.__filename).exists():
86            return
87
88        with open(self.__filename) as file:
89            reader = csv.reader(file, delimiter=self.__delimiter)
90            for row in reader:
91                validate('row length', row, length=5)
92                typ = row[0]
93                plate = Plate(row[1])
94                producer = Producer(row[2])
95                model = Model(row[3])
96                price = Price.parse(row[4])
97                if typ == 'Car':
98                    self.__dealer.add_car(Car(plate, producer, model, price))
99                elif typ == 'Moto':
100                   self.__dealer.add_moto(Moto(plate, producer, model, price))
101               else:
102                   raise ValueError('Unknown vehicle type in default.csv')
```

```python
104    def __save(self) -> None:
105        with open(self.__filename, 'w') as file:
106            writer = csv.writer(file, delimiter=self.__delimiter, lineterminator='\n')
107            for index in range(self.__dealer.vehicles()):
108                vehicle = self.__dealer.vehicle(index)
109                writer.writerow([vehicle.type, vehicle.plate, vehicle.producer, vehicle.model, vehicle.price])
```

Ignition

```
69      def __run(self) -> None:
70          try:
71              self.__load()
72          except ValueError as e:
73              print(e)
74              print('Continuing with an empty list of vehicles...')
75
76          self.__menu.run()
77
78      def run(self) -> None:
79          try:
80              self.__run()
81          except:
82              print('Panic error!', file=sys.stderr)
```

Global exception handler

Avoid to leak sensitive data

```
129  def main(name: str):
130      if name == '__main__':
131          App().run()
132
133  main(__name__)
```

Dirty trick to reach 100% coverage

Private methods are good to ensure that a class is not used improperly,
but may also make testing more difficult:
reaching all paths may be challenging

Relying on global objects, like input() and print(), simplifies the code,
but may also make testing more difficult:
heavy need of mocks and patches

As a general rule, avoid to hardcode global objects

Better to have some way to set those objects,
and have the global objects as default values

Anyhow, let’s test the App we have now

We have often to bypass the check
on the existence of default.csv,
so let’s define a fixture

```
9   @pytest.fixture
10  def mock_path():
11      Path.exists = Mock()
12      Path.exists.return_value = True
13      return Path
14
15
16  @pytest.fixture
17  def data():
18      data = [
19          ['Car', 'CA220NE', 'Fiat', 'Punto', '199.99'],
20          ['Moto', 'CA220NI', 'Kawasaki', 'Ninja', '99.99'],
21      ]
22      return '\n'.join(['\t'.join(d) for d in data])
```

We have also to simulate
the reading of default.csv,
so let’s define another fixture

```python
@patch('builtins.input', side_effect=['0'])
@patch('builtins.print')
def test_app_main(mocked_print, mocked_input):
    with patch.object(Path, 'exists') as mocked_path_exists:
        mocked_path_exists.return_value = False
        with patch('builtins.open', mock_open()):
            main('__main__')
            mocked_print.assert_any_call('*** LaRusso Auto Group ***')
            mocked_print.assert_any_call('0:\tExit')
            mocked_print.assert_any_call('Bye!')
            mocked_input.assert_called()
```

A patch can be also applied this way

For open() we can use mock_open(), provided by unittest.mock

Simulate main execution

Check for some expected output

Test reading of file

```python
@patch('builtins.input', side_effect=['0'])
@patch('builtins.print')
def test_app_load_datafile(mocked_print, mocked_input, mock_path, data):
    with patch('builtins.open', mock_open(read_data=data)):
        App().run()
    mock_path.exists.assert_called_once()
    assert list(filter(lambda x: 'CA220NE' in str(x), mocked_print.mock_calls))
    mocked_input.assert_called()


@patch('builtins.input', side_effect=['0'])
@patch('builtins.print')
def test_app_handles_corrupted_datafile(mocked_print, mocked_input, mock_path):
    with patch('builtins.open', mock_open(read_data='xyz')):
        App().run()
    mocked_print.assert_any_call('Continuing with an empty list of vehicles...')
    mocked_input.assert_called()
```

But also stability on corrupted files

Test for correct usage

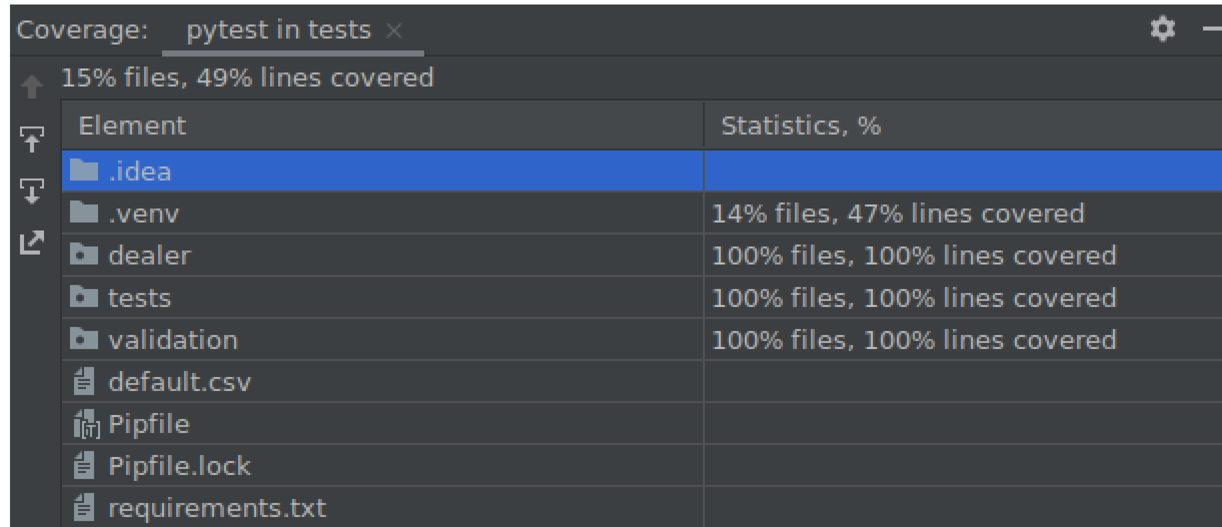
```python
@patch('builtins.input', side_effect=['1', 'CA220NE', 'Fiat', 'Punto', '199.99', '0'])
@patch('builtins.print')
def test_app_add_car(mocked_print, mocked_input, mock_path):
    with patch('builtins.open', mock_open()) as mocked_open:
        App().run()
    assert list(filter(lambda x: 'CA220NE' in str(x), mocked_print.mock_calls))

    handle = mocked_open()
    handle.write.assert_called_once_with('Car\tCA220NE\tFiat\tPunto\t199.99\n')
    mocked_input.assert_called()


@patch('builtins.input', side_effect=['1', 'ca220ne', 'CA220NE', 'Fiat', 'Punto', '199.99', '0'])
@patch('builtins.print')
def test_app_add_car_resists_to_wrong_plate(mocked_print, mocked_input, mock_path):
    with patch('builtins.open', mock_open()) as mocked_open:
        App().run()
    assert list(filter(lambda x: 'CA220NE' in str(x), mocked_print.mock_calls))
    mocked_input.assert_called()

    handle = mocked_open()
    handle.write.assert_called_once_with('Car\tCA220NE\tFiat\tPunto\t199.99\n')
```

But also for stability on mistakes

```python
146    @patch('builtins.input', side_effect=['0'])
147    @patch('builtins.print')
148    def test_app_global_exception_handler(mocked_print, mocked_input):
149        with patch.object(Path, 'exists') as mocked_path_exits:
150            mocked_path_exits.side_effect = Mock(side_effect=Exception('Test'))
151            App().run()
152        assert mocked_input.mock_calls == []
153        assert list(filter(lambda x: 'Panic error!' in str(x), mocked_print.mock_calls))
```

Test the global handler
by introducing some unexpected exception

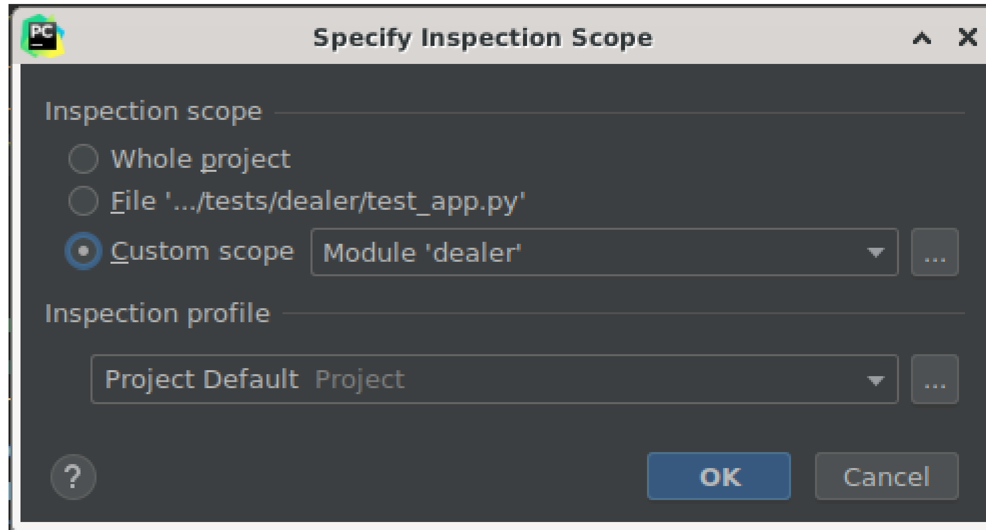
Coverage, the higher the better



Every metric can be tricked, don’t trick yourself

Use the coverage analysis to identify missing tests and unreachable code

Inspect Code



Menu **Code | Inspect Code…**
Menu **Code | Code Cleanup…**
Menu **Code | Optimize Imports**

Use them, check every warning that is reported

Enjoy the final result

```
**************************
*** LaRusso Auto Group ***
**************************
--------------------------------------------------------------------------------------------------
  # PLATE      PRODUCER                       MODEL                                PRICE  FINAL PR.
--------------------------------------------------------------------------------------------------
  1 CA220NE    Fiat                           Punto                               199.99     189.99
  2 CA220NI    Kawasaki                       Ninja                               100.00      97.00
  3 AB123CD    VW                             CC                                 2000.00    1900.00
--------------------------------------------------------------------------------------------------
1:  Add car
2:  Add moto
3:  Remove vehicle
4:  Sort by producer
5:  Sort by price
0:  Exit
?
```

Exercise: Restaurant

Write a TUI to manage a restaurant, and specifically the list of orders.

For an order we are interested in the customer, a textual description and the price. From the discussion with the expert domain we understood that the customer can be represented by strings of letters, numbers and spaces; the length of such strings doesn’t exceed 100 chars. The same restrictions apply to the description of an oder. The price must be represented in euro, with two decimal digits, and cannot be a negative quantity.

The application must allow to

- show all orders
- add and remove orders
- show the list of customers
- restrict the visualization to the orders of a given customer
- sort the orders by ascending price

Data must be saved automatically on disc in a file **default.csv** in the root of the project and loaded when the application starts. We expect tests, too… should I still say it?!?

Exercise: Music Archive

Write a TUI to manage a music archive, and specifically the list of songs.

For a song we are interested in the author, the title, the genre and the duration. From the discussion with the expert domain we understood that the author can be represented by strings of letters, numbers and spaces; the length of such strings doesn’t exceed 100 chars. The same restrictions apply to titles and genres. The duration must be shown in minutes and seconds (eg. 3:25 for 3 minutes and 25 seconds); choose the internal representation.

The application must allow to

- show all songs
- add and remove songs
- show the list of authors
- restrict the visualization to the songs of a given author
- sort songs according to several criteria (of your choice)

Data must be saved automatically on disc in a file **default.csv** in the root of the project and loaded when the application starts. We expect tests, too… should I still say it?!?

Exercise: Medical Office

Write a TUI to manage a medical office, and specifically the list of reservations.

For a reservation we are interested in the name of the patient, the scheduled time, the type of visit and the cost. From the discussion with the expert domain we understood that the patient can be represented by strings of letters, numbers and spaces; the length of such strings doesn't exceed 100 chars. The same restrictions apply to the type of visit. The scheduled time must be shown in hours and minutes (eg. 15:20 for 20 minutes past 15); choose the internal representation. The cost must be represented in euro, with two decimal digits (and definitely it's not a negative number).

The application must allow to

- show all reservations, always sorted by ascending scheduled time
- add and remove reservations
- restrict the visualization to reservations scheduled after a given time

Data must be saved automatically on disc in a file **default.csv** in the root of the project and loaded when the application starts. We expect tests, too… should I still say it?!?

Questions