

1. Парадигми програмування
2. Класи та об'єкти
1. Інкапсуляція
2. Наслідування
3. Поліморфізм
3. Ще один приклад

Буквально все в python є об'єктами. Тому це повністю об'єктно-орієнтована мова програмування.

Стандартні типи даних є класами і містять готові методи для обробки своїх даних: зміну регістра рядка, пошук підрядків, зміна порядку елементів у списку та ін. Обидва цих типи є послідовностями і мають спільні риси поведінки -- це досягається за рахунок існування абстрактного класу "послідовність", від якого наслідуються і рядки, і списки. За необхідності ви також можете створювати і власні класи-типи, наслідуючи їх від вже існуючих і додаючи до них нові функції. Така побудова програмного продукту дозволяє зручно структурувати і в подальшому розширювати код.

**Класи** -- це по суті складні типи даних, визначені програмістом. Як і інші складні типи, клас дозволяє зберігати в змінній елементи інших типів. Поля класу -- це змінні, оголошені всередині класу для збереження даних. Методи класу -- це функції, оголошені для їх обробки та взаємодії з основною програмою та іншими даними.

**Об'єкти** -- це окремі екземпляри класу, по суті змінні цього типу. Оголосивши клас із полями та методами, ви можете створювати скільки завгодно об'єктів, кожен з яких міститиме оголошений в класі набір полів та методів. При цьому дані, що зберігаються всередині, у кожного об'єкту можуть бути свої.

Наприклад, нехай у нас є магазин із різними видами товарів. У кожного виду товарів є назва, ціна та кількість в наявності:

```
class ProductInStore(object):
    price = 9.99
    name = 'new product'
    qty_in_stock = 0

abn911 = ProductInStore()
abn911.name = 'Lego Mindstorm v1'
abn911.price = 199.99
abn911.qty_in_stock = 5
print abn911.name, abn911.price, abn911.qty_in_stock

abn912 = ProductInStore()
abn912.name = 'Anakin action figure 921'
```

```
abn912.price = 56.11
abn912.qty_in_stock = 11
print abn912.name, abn912.price, abn912.qty_in_stock
```

```
D:\ProgrammingCourse>python test.py
Lego Mindstorm v1 199.99 5
Anakin action figure 921 56.11 11
```

Класи оголошуються з ключовим словом `class`, ім'я класу за стандартами python записується наступним чином: всі слова разом, кожне з великої літери (так, приклади у відеолекції називаються неправильно). Для створення екземпляру клас викликається як функція з круглими дужками, повертаючи новий об'єкт, який ви можете зберегти у змінній.

Після оголошення класу, вам (або іншому програмісту) не обов'язково розуміти всю внутрішню будову для його подальшого використання. Головне - знати, які дані в ньому зберігаються, і які його методи можна викликати. В цьому класи подібні до модулів, так як приховують внутрішню будову, залишаючи на поверхні лише зовнішній "інтерфейс" для використання.

Це поєднання даних та функцій всередині однієї сутності, разом із прихованням внутрішньої будови, називається **інкапсуляцією** і є головним принципом ООП.

Цікавий прийом: якщо метод не повинен повертати конкретного значення, часто його пишуть так, щоб повертати сам об'єкт. Це дозволяє за необхідності будувати довгі "ланцюги" методів. Зверніть увагу: зворотній слеш на кінці рядків в коді програми сигналізує інтерпретатору, що інструкція незавершена і її продовження слід шукати на наступному рядку (звичайно його можна використовувати не лише для об'єктів, але для гарного форматування будь-яких команд, наприклад довгих умов в структурі розгалуження).

```
class ProductInStore(object):
    price = 9.99
    name = 'new product'
    qty_in_stock = 0

    def set_price(self, price):
        self.price = price
        return self

    def set_name(self, name):
        self.name = name
        return self
```

```

    def set_stock(self, qty):
        self.qty_in_stock = qty
        return self

abn911 = ProductInStore()
abn911.set_name('Lego Mindstorm v1') \
    .set_price(199.99) \
    .set_stock(5)
print abn911.name, abn911.price, abn911.qty_in_stock

```

### замість

```

abn911 = ProductInStore()
abn911.set_name('Lego Mindstorm v1')
abn911.set_price(199.99)
abn911.set_stock(5)
print abn911.name, abn911.price, abn911.qty_in_stock

```

Серед програмістів є як прихильники, так і противники такого запису. Я вважаю за корисне розповісти про нього, щоб ви не лякалися, коли таке побачите.

При оголошенні класу в дужках можуть бути записані (одне або декілька) імена вже існуючих класів -- це називається **наслідуванням**. В такому випадку новий клас (який називається дочірнім) успадковує всі поля та методи класів перелічених в дужках (які називаються батьківськими).

Додамо 2 види товарів, які унаслідують від початкового. Це "звичайний" товар, який нічим не відрізняється від базового класу, та "товар із опціями", у якого є додаткова опція, яку можна вибрати при замовленні -- нехай вона може впливати на ціну товару.

```

class ProductInStore(object):
    ...
    def __init__(self, name, price):
        self.name = name
        self.price = price
    ...
class SimpleProductInStore(ProductInStore):
    pass

class ConfigurableProductInStore(ProductInStore):
    option_prices = None

    def __init__(self, name, basic_price, add_prices):
        self.name = name
        self.price = basic_price
        self.option_prices = add_prices

```

```

abn911 = ConfigurableProductInStore('Lego Mindstorm v1', 199.99,
{'v1': 0, 'v1.1': 0, 'v2': 21.0, 'v3': 50.5})
print abn911.name, abn911.price, abn911.qty_in_stock, \
      abn911.option_prices
abn912 = SimpleProductInStore('Anakin action figure 921', 56.11)
print abn912.name, abn912.price, abn912.qty_in_stock

```

В даному прикладі клас SimpleProductInStore взагалі немає нічого свого, але ми можемо створювати його екземпляри -- він повністю наслідує поведінку класу ProductInStore. Клас ConfigurableProductInStore також наслідує ProductInStore, отримавши поля name, price, qty\_in\_stock, але до них своє власне option\_prices.

Також вони можуть мати власні методи з однаковими іменами (це називається **поліморфізмом**). Можна оголосити їх окремо в кожному класі:

```

class SimpleProductInStore(ProductInStore):
    def get_price(self):
        return self.price

class ConfigurableProductInStore(ProductInStore):
    # ...
    def get_price(self, option = None):
        if option != None and option in self.option_prices:
            return self.price + self.option_prices[option]
        return self.price

```

Але, якщо цей метод характерний для всіх товарів краще оголосити його одразу в батьківському класі ProductInStore, щоб він точно існував у всіх дочірніх класах, а в дочірніх -- за необхідності його можна буде перевизначити.

```

class ProductInStore(object):
    price = 9.99
    name = 'new product'
    qty_in_stock = 0

    def __init__(self, name, price):
        self.name = name
        self.price = price

    def get_price(self):
        return self.price

class SimpleProductInStore(ProductInStore):
    pass

class ConfigurableProductInStore(ProductInStore):
    option_prices = None

    def get_price(self, option = None):
        if option != None and option in self.option_prices:

```

```

        return self.price + self.option_prices[option]
    return self.price

    def __init__(self, name, basic_price, add_prices):
        self.name = name
        self.price = basic_price
        self.option_prices = add_prices

```

Тепер ми будемо впевнені, що при додаванні будь-яких нових типів товарів метод `get_price()` буде для них визначений і при обробці товару не станеться помилки.

Метод для базового товару може бути складнішим, ніж повернути ціну. Тому замість повторення дії в методі `get_price()` класу `ConfigurableProductInStore` краще викликати батьківський метод:

```

class SimpleProductInStore(ProductInStore):
    def get_price(self, option = None):
        return super(SimpleProductInStore, self).get_price()

class ConfigurableProductInStore(ProductInStore):
    option_prices = None

    def get_price(self, option = None):
        price = super(ConfigurableProductInStore,
self).get_price()
        if option != None and option in self.option_prices:
            price += self.option_prices[option]
        return price

```

Це дозволяє нам повторно використати існуючий код і -- найголовніше -- в такому разі, якщо батьківський клас буде змінено, ми будемо впевнені, що дочірній клас "підхопить" зміни і його код не доведеться зайвий раз переписувати.

**Кінцевий приклад:**

```

class ProductInStore(object):
    price = 9.99
    name = 'new product'
    qty_in_stock = 0

    def __init__(self, name, price):
        self.name = name
        self.price = price

    def set_price(self, price):
        self.price = price
        return self

```

```

def set_name(self, name):
    self.name = name
    return self

def set_stock(self, qty):
    self.qty_in_stock = qty
    return self

def get_price(self):
    return self.price

def report(self):
    return "%s of '%s' are left" % (self.qty_in_stock,
self.name)

class SimpleProductInStore(ProductInStore):
    def get_price(self, option = None):
        return super(SimpleProductInStore, self).get_price()

    def report(self):
        return "-----\nsimple product report\n-----\n%s of '%s'
are left \ncurrent price is %s" % (self.qty_in_stock, self.name,
self.price)

class ConfigurableProductInStore(ProductInStore):
    option_prices = None

    def get_price(self, option = None):
        price = super(ConfigurableProductInStore,
self).get_price()
        if option != None and option in self.option_prices:
            price += self.option_prices[option]
        return price

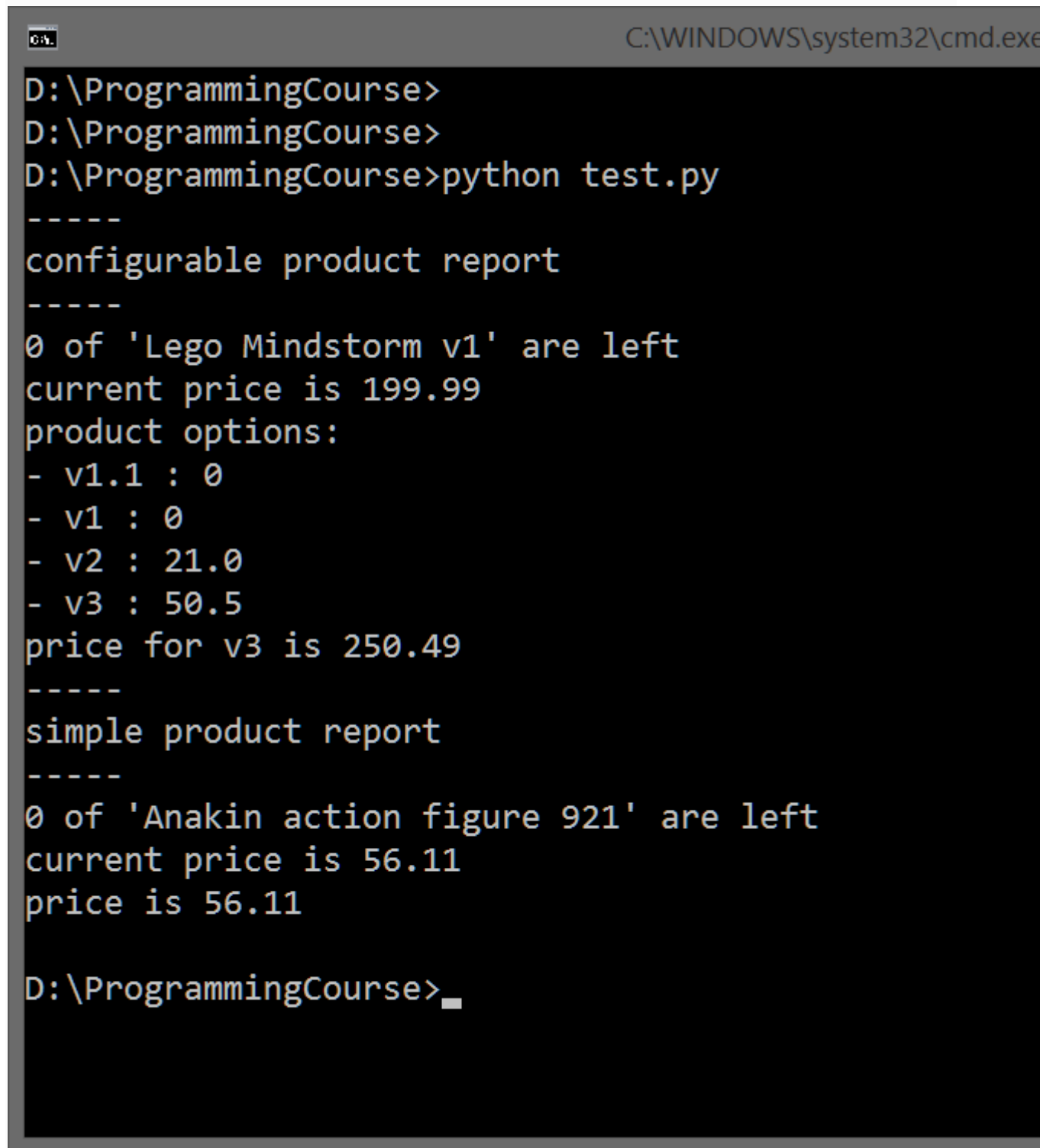
    def __init__(self, name, basic_price, add_prices):
        self.name = name
        self.price = basic_price
        self.option_prices = add_prices

    def report(self):
        report = "-----\nconfigurable product report\n-----\n%s
of '%s' are left \ncurrent price is %s" % (self.qty_in_stock,
self.name, self.price)
        if len(self.option_prices):
            report += "\nproduct options:"
            for i in self.option_prices:
                report += "\n- %s : %s" % (i,
self.option_prices[i])
        return report

abn911 = ConfigurableProductInStore('Lego Mindstorm v1', 199.99,
{'v1': 0, 'v1.1': 0, 'v2': 21.0, 'v3': 50.5})
print abn911.report()
print 'price for v3 is %s' % (abn911.get_price('v3'))

```

```
abn912 = SimpleProductInStore('Anakin action figure 921', 56.11)
print abn912.report()
print 'price is %s' % (abn912.get_price())
```



The screenshot shows a Windows command prompt window with the title bar "C:\WINDOWS\system32\cmd.exe". The prompt is at "D:\ProgrammingCourse>". The user has entered "python test.py". The output of the script is as follows:

```
D:\ProgrammingCourse>
D:\ProgrammingCourse>
D:\ProgrammingCourse>python test.py
-----
configurable product report
-----
0 of 'Lego Mindstorm v1' are left
current price is 199.99
product options:
- v1.1 : 0
- v1 : 0
- v2 : 21.0
- v3 : 50.5
price for v3 is 250.49
-----
simple product report
-----
0 of 'Anakin action figure 921' are left
current price is 56.11
price is 56.11

D:\ProgrammingCourse>_
```

Містить 5 класів:

- `datetime.datetime` -- для представлення одночасно дати і часу.
- `datetime.date` -- для представлення лише дати. Містить методи, аналогічні методам `datetime` для роботи з датами.

- `datetime.time` -- для представлення лише часу. Містить методи, аналогічні методам `datetime` для роботи з часом.
- `datetime.timedelta` -- для представлення різниці у часі, використовується для проведення арифметичних дій над датами і часом
- `datetime.tzinfo` -- для представлення інформації про часову зону (часовий пояс).

[Повна документація](#) до модуля доступна на сайті python (англійською мовою).

## КЛАС DATETIME.DATETIME

Для створення об'єктів класу `datetime.datetime` використовується як звичайний конструктор, так і ряд методів, які належать самому класу та не потребують створення об'єктів для свого виклику.

`x = datetime.datetime(year, month, day[, hour[, minute[, second[, microsecond[, tzinfo]]]])` -- конструктор класу. Перші 3 параметри (рік, місяць та день) є обов'язковими, інші необов'язкові і можуть задаватися як іменовані аргументи.

```
dt = datetime.datetime(2014, 1, 16)
dt2 = datetime.datetime(2014, 1, 16, minute=11)
```

`x = datetime.datetime.today()` -- метод класу, використовується для створення об'єкту сьогоднішньої дати (як альтернатива конструктору).

`x = datetime.datetime.now([tz])` -- те ж саме, але з можливістю задати часову зону.

```
dt3 = datetime.datetime.today()
```

`x = datetime.datetime.combine(date, time)` -- метод класу, створює об'єкт `datetime` з пари об'єктів `date` та `time`

`x = datetime.datetime.strptime(date_string, format)` -- метод класу, створює об'єкт `datetime` з рядка `date_string`, вважаючи, що він містить дату в форматі `format`.

Поля об'єктів класу доступні лише для читання:

`x.year` -- рік (ціле число).

`x.month` -- місяць (ціле число від 1 до 12).



**x.day** -- число (ціле число від 1 до кількості днів у місяці).

**x.hour** -- години (ціле число від 0 до 23).

**x.minute** -- хвилини (ціле число від 0 до 59).

**x.second** -- секунди (ціле число від 0 до 59).

**x.microsecond** -- мікросекунди (ціле число від 0 до 1000000).

**x.tzinfo** -- часова зона (об'єкт класу `datetime.tzinfo` або `None`).

Методи об'єктів класу.

**x.date()** -- повертає об'єкт `datetime.date` з аналогічними даними про дату (роком, місяцем та числом).

**x.time()** -- повертає об'єкт `datetime.time` з аналогічними даними про час.

**x.replace([year[, month[, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]]]]])** -- повертає новий об'єкт `datetime` із зміненими значеннями вказаних полів відносно об'єкту `x`. Для вказання конкретних полів необхідно передати їх назву як іменовані параметри.

```
print dt.replace(year=2013,hour=12) # datetime.datetime(2013, 1, 16, 12, 0)
print dt # datetime.datetime(2014, 1, 16, 0, 0)
```

**x.weekday()** -- повертає порядковий номер дня тижня (0 -- понеділок, 6 -- неділя).

```
print dt.weekday() # 3
```

**x.isoweekday()** -- повертає порядковий номер дня тижня в ISO-форматі (1 -- понеділок, 7 -- неділя).

```
print dt.isoweekday() # 4
```

**x.isocalendar()** -- повертає тьюпл, що містить рік, номер тижня та номер дня тижня в ISO-форматі.

```
print dt.isocalendar() # (2014, 3, 4)
```

**x.isoformat([sep])** -- повертає рядок, що містить дату, відформатовану у форматі ISO. В якості роздільника між датою та часом використовується `sep`. Якщо `sep` не заданий, то `'T'`.

```
print x.isoformat() # 2014-01-16T00:00:00
```

`x.strftime(format)` -- повертає рядок, що містить дату у форматі заданому рядком `format`. Точний перелік позначень для всіх можливих елементів дати можна переглянути в [документації](#).

```
dt.strftime("%A, %d. %B %Y %I:%M%p")    #'Thursday, 16. January 2014 12:00AM'
```

## КЛАС DATETIME.TIMEDELTA

`x = datetime.timedelta([days[, seconds[, microseconds[, milliseconds[, minutes[, hours[, weeks]]]]]])` -- конструктор класу. Всі аргументи є необов'язковими і можуть задаватися в іменованому вигляді.

```
td = datetime.timedelta(hours=48)
td2 = datetime.timedelta()
```

Об'єкти класу містять наступні поля, доступні лише для читання:

**x.days** -- кількість днів у проміжку часу (ціле число від -999999999 до 999999999 включно).

**x.seconds** -- кількість секунд у проміжку часу (ціле число від 0 до 86399 включно).

**x.microseconds** -- кількість мікросекунд у проміжку часу.

Об'єкти класу мають один метод

**x.total\_seconds()** -- повертає повну кількість секунд у проміжку часу.

```
print td.total_seconds()    # 172800.0
```

Крім того, об'єкти класу `datetime.timedelta` можна порівнювати, додавати та віднімати між собою, множити та ділити на ціле число, додавати то віднімати до дат (об'єктів `datetime.datetime`, `datetime.date`, `datetime.time`). Також об'єкти `datetime.timedelta` виникають при відніманні дат одна від одної.

```
C:\WINDOWS\system32\cmd.exe

>>>
>>> import datetime
>>> dt1 = datetime.datetime(2004, 05, 06, 12)
>>> print dt1
2004-05-06 12:00:00
>>> td_2years = datetime.timedelta(days=365*2)
>>> print td_2years
730 days, 0:00:00
>>> dt2 = dt1 + td_2years / 2
>>> print dt2
2005-05-06 12:00:00
>>>
```

Якщо під час виконання програми щось іде не так, інтерпретатор python генерує об'єкт виключення одного із стандартних класів, залежно від того, що саме сталося. Всі ці стандартні класи містяться у модулі exception, який не вимагає додаткового підключення.

Виникнення таких виключень може бути перехоплене в коді програми за допомогою конструкції

```
try:
    # дії, які потенційно можуть викликати помилку
except ExceptionClass:
    # дії, які мають виконатися у випадку помилки
```

Замість ExceptionClass вказується назва конкретного класу виключень, які повинні перехоплюватися. Виключення інших класів не будуть перехоплені у випадку їх появи. Найпростіший спосіб визначити необхідний клас виключення -- це відтворити необхідну помилку та подивитися назву класу її виключення у стандартному повідомленні інтерпретатора:

```
x = int(raw_input('input number:'))
```

```
C:\WINDOWS\system32\cmd.exe

D:\ProgrammingCourse>python test.py
input number:t
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    x = int(raw_input('input number:'))
ValueError: invalid literal for int() with base 10: 't'
```

В нашому прикладі -- це ValueError.

Якщо необхідно перехоплювати одночасно декілька класів виключень, вони перераховуються у тьюплі:

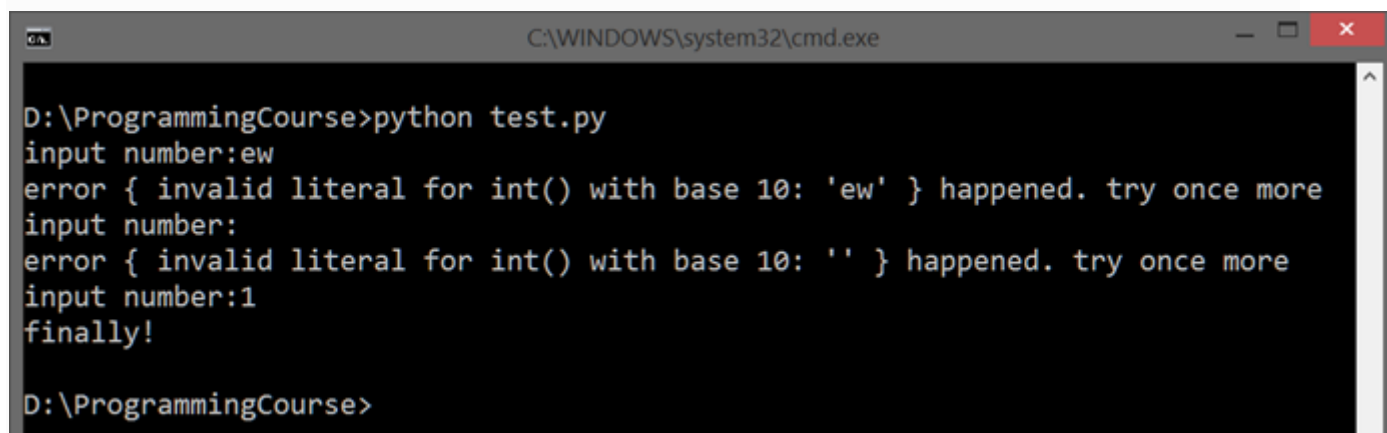
```
x = None
while not x:
    try:
        x = int(raw_input('input number:'))
    except (TypeError, ValueError):
        print 'error happened. try once more'
print 'finally!'
```

Різні виключення можуть мати різні поля з даними, але загальним для всіх є поле message, в якому зберігається текстове повідомлення, що описує помилку. Для отримання докладних відомостей про помилку перехоплене виключення можна зберегти як змінну:

```
x = None
while not x:
    try:
        x = int(raw_input('input number:'))
    except (TypeError, ValueError) as e:
        print 'error { %s } happened. try once more' % (e.message)
print 'finally!'
```

Стандартні виключення наслідуються одне від одного, утворюючи складну ієрархію. Вказуючи в ехсепт виключення батьківських класів, ви відловлюватимете також дочірні для них виключення:

```
x = None
while not x:
    try:
        x = int(raw_input('input number:'))
    except StandardError as e:
        print 'error { %s } happened. try once more' % (e.message)
print 'finally!'
```



```
C:\WINDOWS\system32\cmd.exe

D:\ProgrammingCourse>python test.py
input number:ew
error { invalid literal for int() with base 10: 'ew' } happened. try once more
input number:
error { invalid literal for int() with base 10: '' } happened. try once more
input number:1
finally!

D:\ProgrammingCourse>
```

Як це працює? При виникненні помилки робота поточної функції припиняється і генерується об'єкт виключення, який передається з функції, де виникла помилка, на рівень вище -- у функцію, яка її викликала, також припиняючи її роботу, потім ще на рівень вище, і так далі до тіла головної програми. Якщо на цьому шляху виключення не було перехоплене (жодна

з цих функцій не знаходилася всередині блоку try..except), робота всієї програми буде таким чином зупинена і на екран буде виведене стандартне повідомлення про помилку із вказанням точки в коді, де помилка відбулася.

Повна ієрархія стандартних виключень виглядає наступним чином:

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StandardError
        |   +-- BufferError
        |   +-- ArithmeticError
        |       |   +-- FloatingPointError
        |       |   +-- OverflowError
        |       |   +-- ZeroDivisionError
        |   +-- AssertionError
        |   +-- AttributeError
        |   +-- EnvironmentError
        |       |   +-- IOError
        |       |   +-- OSError
        |           |   +-- WindowsError (Windows)
        |           |   +-- VMSError (VMS)
        |   +-- EOFError
        |   +-- ImportError
        |   +-- LookupError
        |       |   +-- IndexError
        |       |   +-- KeyError
        |   +-- MemoryError
        |   +-- NameError
        |       |   +-- UnboundLocalError
        |   +-- ReferenceError
        |   +-- RuntimeError
        |       |   +-- NotImplementedError
        |   +-- SyntaxError
        |       |   +-- IndentationError
        |       |   +-- TabError
        |   +-- SystemError
        |   +-- TypeError
        |   +-- ValueError
        |       |   +-- UnicodeError
        |           |   +-- UnicodeDecodeError
        |           |   +-- UnicodeEncodeError
        |           |   +-- UnicodeTranslateError
    +-- Warning
        +-- DeprecationWarning
        +-- PendingDeprecationWarning
        +-- RuntimeWarning
        +-- SyntaxWarning
        +-- UserWarning
        +-- FutureWarning
+-- ImportWarning
+-- UnicodeWarning
+-- BytesWarning
```