Unit Testing

همان‌قدر که نوشتن کد خوب مهم است، نوشتن تست‌های خوب نیز مهم است. بهتر است خودتان باگ‌ها را پیدا کنید تا اینکه کاربران نهایی آنها را به شما گزارش دهند!

در این بخش، با فایل‌ها خارج از نوت‌بوک کار خواهیم کرد. کد خود را در یک فایل py. ذخیره خواهیم کرد و سپس اسکریپت تست خود را در یک فایل py. دیگر ذخیره خواهیم کرد. به طور معمول، ما این فایل‌ها را با استفاده از یک ویرایشگر متن مانند Brackets یا Atom یا درون یک محیط توسعه یکپارچه مانند Spyder یا Pycharm کد می‌نویسیم. اما از آنجا که در اینجا هستیم، بیایید از Jupyter استفاده کنیم!

با استفاده از یکی از ویژگی‌های IPython می‌توانیم محتوای یک سلول را با استفاده از `%%writefile` در یک فایل ذخیره کنیم.
چیزی که تاکنون ندیده‌ایم؛ می‌توانید دستورات ترمینال را از طریق یک سلول Jupyter با استفاده از `!` اجرا کنید.

Testing tools

ده‌ها کتابخانه تست خوب وجود دارند. بیشتر آنها پکیج‌های شخص ثالثی هستند که نیاز به نصب دارند، مانند:

- pylint (https://www.pylint.org/)
- pyflakes (https://pypi.python.org/pypi/pyflakes/)
- pep8 (https://pypi.python.org/pypi/pep8)

این ابزارها ابزارهای ساده‌ای هستند که فقط به کد شما نگاه می‌کنند و اگر مشکلاتی مانند مسائل استایل یا مشکلات ساده‌ای مانند فراخوانی نام متغیرها قبل از اختصاص به آنها وجود داشته باشد، به شما اطلاع می‌دهند.

روشی بسیار بهتر برای تست کردن کد شما، نوشتن تست‌هایی است که داده‌های نمونه را به برنامه شما ارسال کنند و مقایسه کنند که چه چیزی برگردانده می‌شود و چه نتیجه مورد انتظار است.
دو ابزار از کتابخانه استاندارد در دسترس هستند:

- unittest (https://docs.python.org/3/library/unittest.html)
- doctest (https://docs.python.org/3/library/doctest.html)

بیایید ابتدا به pylint نگاهی بیندازیم، سپس با استفاده از unittest برخی تست‌های سنگین‌تر انجام دهیم.

`pylint`

`pylint` برای استایل و همچنین برخی از منطق بسیار ابتدایی برنامه تست انجام می‌دهد.

ابتدا، اگر هنوز ندارید (و احتمالاً دارید، زیرا قسمتی از توزیع آناکوندا است)، باید `pylint` را نصب کنید.
بعد از انجام این کار، می‌توانید سلول را کامنت کنید، دیگر به آن نیاز نخواهید داشت.

In [3]:

```
! pip install pylint
```

```
Defaulting to user installation because normal site-packages is not writea
ble
Requirement already satisfied: pylint in c:\programdata\anaconda3\lib\site
-packages (2.16.2)
Requirement already satisfied: astroid<=2.16.0-dev0,>=2.14.2 in c:\program
data\anaconda3\lib\site-packages (from pylint) (2.14.2)
Requirement already satisfied: tomli>=1.1.0 in c:\programdata\anaconda3\li
b\site-packages (from pylint) (2.0.1)
Requirement already satisfied: platformdirs>=2.2.0 in c:\programdata\anaco
nda3\lib\site-packages (from pylint) (2.5.2)
Requirement already satisfied: dill>=0.2 in c:\programdata\anaconda3\lib\s
ite-packages (from pylint) (0.3.6)
Requirement already satisfied: tomlkit>=0.10.1 in c:\programdata\anaconda3
\lib\site-packages (from pylint) (0.11.1)
Requirement already satisfied: isort<6,>=4.2.5 in c:\programdata\anaconda3
\lib\site-packages (from pylint) (5.9.3)
Requirement already satisfied: mccabe<0.8,>=0.6 in c:\programdata\anaconda
3\lib\site-packages (from pylint) (0.7.0)
Requirement already satisfied: colorama>=0.4.5 in c:\programdata\anaconda3
\lib\site-packages (from pylint) (0.4.6)
Requirement already satisfied: lazy-object-proxy>=1.4.0 in c:\programdata
\anaconda3\lib\site-packages (from astroid<=2.16.0-dev0,>=2.14.2->pylint)
(1.6.0)
Requirement already satisfied: wrapt<2,>=1.11 in c:\programdata\anaconda3
\lib\site-packages (from astroid<=2.16.0-dev0,>=2.14.2->pylint) (1.14.1)
Requirement already satisfied: typing-extensions>=4.0.0 in c:\programdata
\anaconda3\lib\site-packages (from astroid<=2.16.0-dev0,>=2.14.2->pylint)
(4.5.0)
```

بیایید یک اسکریپت بسیار ساده را ذخیره کنیم:

In [4]:

```
%%writefile simple1.py
a = 1
b = 2
print(a)
print(b)

```

```
Writing simple1.py
```

حالا بیایید آن را با استفاده از pylint بررسی کنیم

In [5]:

```
! pylint simple1.py
```

```
************* Module simple1
simple1.py:1:0: C0114: Missing module docstring (missing-module-docstring)
simple1.py:1:0: C0103: Constant name "a" doesn't conform to UPPER_CASE nam
ing style (invalid-name)
simple1.py:2:0: C0103: Constant name "b" doesn't conform to UPPER_CASE nam
ing style (invalid-name)

--------------------------------------------------------------------
Your code has been rated at 2.50/10 (previous run: 10.00/10, -7.50)
```

ابتدا، pylint برخی از مشکلات استایلی را لیست می‌کند - می‌خواهد در انتها یک خط جدید اضافه شود، ماژول‌ها و تعاریف تابع باید از docstring توصیفی برخوردار باشند و استفاده از کاراکترهای تکی برای نامگذاری متغیرها انتخاب ضعیفی است.

اما مهمتر از همه، pylint یک خطا در برنامه شناسایی کرده است - استفاده از متغیر قبل از اختصاص به آن. این نیاز به اصلاح دارد.

توجه کنید که pylint برنامه ما را با امتیاز منفی 12.5 از 10 امتیاز گرفته است. بیایید سعی کنیم این امتیاز را بهبود ببخشیم!

In [12]:

```
%%writefile simple1.py
"""
A very simple script.
"""

def myfunc():
    '''
    An extremely simple function
    '''
    first = 1
    second = 2
    print(first)
    print(second)

myfunc()
```

```
Overwriting simple1.py
```

In [13]:

```
! pylint simple1.py
```

```
--------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 8.33/10, +1.67)
```

بسیار بهتر شد! امتیاز ما به 10.00 از 10 ارتقا یافت. متأسفانه، خط جدید نهایی با نحوه نوشتن Jupyter در یک فایل مربوط است و در اینجا چیز زیادی که بتوانیم درباره آن انجام دهیم وجود ندارد. با این حال، pylint به ما در رفع برخی از مشکلاتمان کمک کرد. اما اگر مشکل پیچیده‌تر بود چه اتفاقی می‌افتاد؟

In [14]:

```
%%writefile simple2.py
"""
A very simple script.
"""

def myfunc():
    '''
    An extremely simple function
    '''
    first = 1
    second = 2
    print(first)
    print('second')

myfunc()
```

```
Writing simple2.py
```

In [15]:

```
! pylint simple2.py
```

```
************* Module simple2
simple2.py:10:4: W0612: Unused variable 'second' (unused-variable)

-------------------------------------------------------------------
Your code has been rated at 8.33/10 (previous run: 8.33/10, +0.00)
```

pylint به ما می‌گوید که در خط 10 یک متغیر بی‌استفاده وجود دارد، اما این نمی‌داند که ممکن است خروجی غیرمنتظره‌ای از خط 13 دریافت کنیم! برای این کار نیاز به مجموعه‌ای از ابزارهای قدرتمندتر است. به همین دلیل unittest وارد صحنه می‌شود.

unittest

unittest به شما امکان می‌دهد برنامه‌های تست خود را بنویسید. هدف این است که یک مجموعه خاصی از داده‌ها را به برنامه‌ی خود ارسال کنید و نتایج بازگشتی را با یک نتیجه مورد انتظار مقایسه کنید.

بیایید یک اسکریپت ساده بسازیم که کلمات در یک رشته داده شده را بزرگ کند. اسم آن را **cap.py** خواهیم گذاشت.

In [16]:

```
%%writefile cap.py
def cap_text(text):
    return text.capitalize()
```

```
Writing cap.py
```

In [17]:

```
def cap_text(text):
    return text.capitalize()

cap_text("monty python")
```

Out[17]:

```
'Monty python'
```

حالا یک اسکریپت تست خواهیم نوشت. می‌توانیم آن را هر چیزی که می‌خواهیم صدا بزنیم، اما **test_cap.py** به نظر گزینه‌ای منطقی می‌رسد.

در هنگام نوشتن توابع تست، بهتر است از ساده به پیچیده رفت، زیرا هر تابع به ترتیب اجرا خواهد شد. در اینجا ابتدا رشته‌های ساده و یک کلمه‌ای را تست می‌کنیم، سپس تستی برای رشته‌های چند کلمه‌ای انجام می‌دهیم.

In [2]:

```
%%writefile test_cap.py
import unittest
import cap

class TestCap(unittest.TestCase):

    def test_one_word(self):
        text = 'python'
        result = cap.cap_text(text)

        self.assertEqual(result, 'Python')

    def test_multiple_words(self):
        text = "monty python"
        result = cap.cap_text(text)

        self.assertEqual(result, 'Monty Python')

if(__name__ == '__main__'):
    unittest.main()
```

```
Overwriting test_cap.py
```

In [2]:

```
! python test_cap.py
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

چه اتفاقی افتاد؟ به نظر می‌رسد که متد `.capitalize()` تنها حرف اول کلمه اول در یک رشته را بزرگ می‌کند. با انجام یک تحقیق کوچک دربارهٔ متدهای رشته، متوجه می‌شویم که ممکن است متد `.title()` ما را به هدفمان برساند.

In [3]:

```
%%writefile cap.py
def cap_text(text):
    return text.title()
```

```
Overwriting cap.py
```

In [3]:

```
! python test_cap.py
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

عالی، هر دو تست ما پاس شدن! اما آیا همه موارد را تست کردیم؟ بیایید یک تست دیگر به **test_cap.py** اضافه کنیم تا ببینیم آیا با کلماتی که حاوی واژه‌های ایتالیک هستند مانند "don't"، به درستی برخورد می‌کند.

در یک ویرایشگر متنی این کار آسان است، اما در Jupyter ما باید از ابتدا شروع کنیم.

In [4]:

```
%%writefile test_cap.py
import unittest
import cap

class TestCap(unittest.TestCase):

    def test_one_word(self):
        text = 'python'
        result = cap.cap_text(text)

        self.assertEqual(result, 'Python')

    def test_multiple_words(self):
        text = "monty python"
        result = cap.cap_text(text)

        self.assertEqual(result, 'Monty Python')

    def test_with_apostrophes(self):
        text = "monty python's flying circus"
        result = cap.cap_text(text)

        self.assertEqual(result, "Monty Python's Flying Circus")

if(__name__ == '__main__'):
    unittest.main()
```

```
Overwriting test_cap.py
```

In [5]:

```
1 ! python test_cap.py
```

```
..F
======================================================================
FAIL: test_with_apostrophes (__main__.TestCap)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\babak\Python For Everyone\08-Errors and Exception Handlin
g\test_cap.py", line 22, in test_with_apostrophes
    self.assertEqual(result, "Monty Python's Flying Circus")
AssertionError: "Monty Python'S Flying Circus" != "Monty Python's Flying C
ircus"
- Monty Python'S Flying Circus
?              ^
+ Monty Python's Flying Circus
?              ^


----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
```

الان باید یک راه حل پیدا کنیم که اپستروف ها را مدیریت کند! یک راه وجود دارد (جستجوی `capwords` از ماژول `string`) اما ما آن را به عنوان یک تمرین برای خواننده واگذار می کنیم.

عالی! حالا باید یک درک پایه ای از تست واحد داشته باشید!