CS50x Iran

Unit Tests

Week note

پاییز و زمستان ۱۴۰۴

1 فهرست

- تا به حال ممکن است برای تست کردن کدها از دستورات چاپی مثل `print` استفاده کرده باشید، و یا از طرفی، ممکنه که برای تست کردن کدهایتان به CS50 متکی باشید!

- رایج‌ترین کار این است که برای تست کردن برنامه هایمان کدهایی بنویسیم که توانایی انجام این تست ها را روی برنامه‌ی ما داشته باشند.

- در پنجره کنسول `code calculator.py` را تایپ کنید. توجه کنید که ممکن است قبلاً این فایل را هفته قبلی استفاده کرده باشید.

کد خود را بصورت زیر بنویسید:

calculator.py

```
def main():
    x = int(input("What's x? "))
    print("x squared is", square(x))

def square(n):
    return n * n

if __name__ == "__main__":
    main()
```

توجه کنید که می‌توانید کد بالا را با استفاده از برخی اعداد مانند 2 تست کنید. با این حال در نظر بگیرید چرا ممکن است که بخواهید تستی ایجاد کنید که از عملکرد درست تابع اطمینان پیدا کند.

طبق روال همیشگی بیایید با اجرای دستور `code test_calculator.py` یک برنامه تست جدید ایجاد کنیم. اکنون کد خود را در فایل ایجاد شده به صورت زیر تغییر دهید:

test_calculator.py

```
from calculator import square

def main():
    test_square()

def test_square():
    if square(2) != 4:
        print("2 squared was not 4")
    if square(3) != 9:
        print("3 squared was not 9")

if __name__ == "__main__":
    main()
```

توجه کنید در خط اول، تابع `square` را که در فایل قبلی (calculator.py) نوشته بودیم در این برنامه وارد می‌کنیم تا بتوانیم در ادامه این تابع را تست کنیم.

برای تست تابع `square` تابع دیگری به نام `test_square` می‌سازیم و در آن شرایط مطلوب برای تست تابع `square` را پیاده سازی می‌کنیم، برای مثال بررسی می‌کنیم که آیا تابع `square` می‌تواند به درستی توان دو عدد 2 را بدست بیاورد؟ اگر این محاسبه به درستی رخ ندهد تابع `test_square` باید یک پیغام خطا نمایش دهد و در غیر این صورت خطایی نداشته باشد.

دقت کنید که نداشتن خطا در توابعی که وظیفه تست دارند به این معناست که تمامی تست‌های ما به درستی انجام شده و تابع مد نظر (در اینجا `square`) به درستی کار می‌کند، و یا می‌تواند به این معنا باشد که ما یکی از حالت های خاص که می‌تواند باعث بروز خطا شود را بررسی نکرده ایم و به همین دلیل، خطایی نیز برای یافتن این مشکل نداریم. پس حتما به حالت‌های خاص توجه کنید.

- داخل پنجره کنسول `python test_calculator.py` تایپ کنید، خواهید دید که هیچ خروجی برای شما چاپ نمی‌شود که به این می‌تواند به این معنا باشد که تمامی حالات درست می‌باشند یا از طرف دیگر، ممکن است تابع تست ما یکی از مواردی را که می تواند خطا ایجاد کند را، کشف نکرده باشد.

- در حال حاضر، کد ما فقط دو شرط را آزمایش می‌کند. اگر بخواهیم بسیاری از شرایط دیگر را نیز آزمایش کنیم، کد تست ممکن است بسیار شلوغ و پیچیده شود.
چگونه می‌توانیم بدون شلوغ و زیاد شدن کد تست، قابلیت‌های آن را گسترش دهیم؟

- در پایتون، دستور `assert` به ما اجازه می‌دهد تا به کامپایلر بگوییم تا درستی یک ادعا را بررسی کند. برای استفاده از این دستور در کد تستی خود، می‌توانید به شکل زیر عمل کنید:

test_calculator.py

```
from calculator import square

def main():
    test_square()

def test_square():
    assert square(2) == 4
    assert square(3) == 9

if __name__ == "__main__":
    main()
```

توجه کنید که ما در اینجا به طور مشخص باید معلوم کنیم که توابع `square(2)` و `square(3)` باید مساوی چه مقداری باشند. همان طور که می‌بینید کد ما از چهار خط به دو خط کاهش یافته است.

- ما می‌توانیم با تغییر دادن تابع `square` در calculator.py ادعاهای وارد شده را عمداً خراب کنیم تا نتایج را در صورت بروز خطا ببینیم:

calculator.py

```
def main():
    x = int(input("What's x? "))
    print("x squared is", square(x))

def square(n):
    return n + n

if __name__ == "__main__":
    main()
```

- توجه کنید که ما در تابع `square(n)`، عملگر ضرب * را به جمع + تغییر داده‌ایم .

- با اجرای دستور `python test_square.py` در پنجره کنسول، خطا `AssertionError` را مشاهده می‌کنید که نشان می‌دهد یکی از شرایط مورد نظر برآورده نشده است. چالشی که در حال حاضر با آن روبرو هستیم، این است که کد ما برای ارائه‌ی پیغام خطا بطور دقیق‌تر به کاربران، باید تعداد خطوط بیشتری را داشته باشد. برای این منظور می‌توانیم به صورتی که در صفحه‌ی بعد مشاهده می‌کنید، کد خود را ویرایش کنیم.

test_calculator.py

```
def main():
    test_square()

def test_square():
    try:
        assert square(2) == 4
    except AssertionError:
        print("2 squared is not 4")
    try:
        assert square(3) == 9
    except AssertionError:
        print("3 squared is not 9")
    try:
        assert square(-2) == 4
    except AssertionError:
        print("-2 squared is not 4")
    try:
        assert square(-3) == 9
    except AssertionError:
        print("-3 squared is not 9")
    try:
        assert square(0) == 0
    except AssertionError:
        print("0 squared is not 0")

if __name__ == "__main__":
    main()
```

- توجه کنید که اجرای کد صفحه‌ی قبل چندین خطا را مدیریت و پردازش می‌کند.

- این یک مثال خوب است که به شما کمک می‌کند تا اهمیت نوشتن تست های جامع برای یک تابع را به خوبی درک کنید. با داشتن تست های مفید و جامع، شما دقت برنامه هایی که می‌نویسید را چندین مرتبه بالاتر می‌برید.

- با این حال، کد بالا یک مشکل بزرگ دارد:
  - چگونه می‌توانیم تست کردن کد خود را بدون ده ها خط کد مانند بالا و به شکلی آسان تر پیاده سازی کنیم؟

- شما می‌توانید در مستندات assert پایتون، اطلاعات بیشتری درباره آن کسب کنید.

- pytest یک کتابخانه شخص ثالث است که به شما این امکان را می‌دهد تا برنامه خود را با روش unit testing یا تست هر واحد بیازمایید، به عبارت دیگر، می‌توانید توابع مختلفی که در برنامه‌ی خود دارید را به صورت یک به یک تست کنید تا اطمینان حاصل کنید که همه‌ی قسمت‌های کدتان به درستی کار می‌کنند.
- Unit Testing ساز و کاری است برای آزمودن درستی کارکرد واحدهای کد. واحدها یا (unit)، بلوک‌های کوچکی از کد هستند که می‌توان بصورت جداگانه آنها را مورد آزمایش قرار داد. بر اساس این ساز و کار، درستی عملکرد هر بلوک کد که یک وظیفه (task) خاصی را انجام می‌دهد یا چیز خاصی را برمی‌گرداند، ارزیابی می‌شود.
- برای استفاده از pytest، لطفاً دستور `pip install pytest` را در پنجره کنسول خود وارد و اجرا کنید تا کتابخانه pytest برای شما نصب شود.

- قبل از استفاده از pytest، تابع فایل test_calculator برنامه خود را به شکل زیر تغییر دهید:

test_calculator.py

```python
from calculator import square


def test_assert():
    assert square(2) == 4
    assert square(3) == 9
    assert square(-2) == 4
    assert square(-3) == 9
    assert square(0) == 0
```

- توجه داشته باشید که کد بالا تمامی شرایطی را که برای انجام تست نیاز داریم را، تأیید می‌کند.
- pytest به ما این امکان را می‌دهد که برنامه خود را مستقیماً از طریق آن اجرا کنیم و به راحتی نتایج شرایط تست (Test Conditions) خود را ببینیم.

- برای اجرای برنامه، در پنجره ترمینال دستور `python test_calculator.py` را تایپ و اجرا کنید. خواهید دید که بلافاصله خروجی نمایش داده می‌شود. توجه کنید که حرف F قرمز در نزدیکی بالای خروجی، نشانگر وجود خطا در کد شما است. همچنین توجه کنید که حرف E قرمز نیز، راهنمایی‌هایی درباره خطاهای برنامه calculator.py شما ارائه می‌دهد. با توجه به این خروجی، می‌توانید حالتی را تصور کنید که در آن 3 * 3 به جای 9، عدد 6 را نمایش داده است. بر اساس نتایج این تست، می‌توانیم کدهای برنامه calculator.py خود را به صورت زیر اصلاح کنیم. توجه کنید که بجای عملگر + (جمع) در تابع `square`، عملگر * (ضرب) را قرار دادیم تا کد ما به درستی کار کند.

calculator.py

```
def main():
    x = int(input("What's x? "))
    print("x squared is", square(x))

def square(n):
    return n * n

if __name__ == "__main__":
    main()
```

- پس از اصلاح کد، با اجرای دوباره دستور `python test_calculator.py` متوجه می‌شوید که هیچ خطایی نمایش داده نخواهد شد. تبریک می‌گوییم! کد شما به درستی کار می‌کند!
- اما در حال حاضر، اینکه pytest بعد از اولین خطا در انجام تست متوقف خواهد شد، مطلوب نیست. پس بار دیگر، برنامه calculator.py خود را به شکل اشتباه و خطادار برگردانید:

calculator.py

```
def main():
    x = int(input("What's x? "))
    print("x squared is", square(x))

def square(n):
    return n + n

if __name__ == "__main__":
    main()
```

- توجه کنید که در تابع `square`، عملگر * را به عملگر + تغییر داده‌ایم و تابع اشتباه بازگشته است.

- برای بهبود کد تست خود، کد های فایل test_calculator.py را به چند گروه تست مختلف تقسیم کنید:

test_calculator.py

```python
from calculator import square

def test_positive():
    assert square(2) == 4
    assert square(3) == 9

def test_negative():
    assert square(-2) == 4
    assert square(-3) == 9

def test_zero():
    assert square(0) == 0
```

توجه کنید که ما همان پنج شرط قبلی تست را به سه تابع مختلف تقسیم کرده‌ایم. فریمورک های تست مانند pytest، تمام توابع را حتی اگر خطایی در یکی از آنها رخ دهد، اجرا می‌کنند. با اجرای مجدد `pytest test_calculator.py`، متوجه خواهید شد که نتیجه ای بسیار بیشتر از یک خطا نمایش داده می‌شود. نمایش خطاهای بیشتر به ما این امکان را می‌دهد تا بیشتر و بهتر مشکلاتی که در کدمان وجود دارد را بیابیم و بر روی آنها تمرکز کنیم.

- بعد از بهبود دادن کد تست، برنامه calculator.py را به حالت کاملاً کارآمد و سالم آن بازگردانید:

calculator.py

```
def main():
    x = int(input("What's x? "))
    print("x squared is", square(x))

def square(n):
    return n * n

if __name__ == "__main__":
    main()
```

- با اجرای مجدد `pytest test_calculator.py`، متوجه خواهید شد که هیچ خطایی پیدا نمی‌شود.
- به طور خلاصه، تعریف و تعیین شرایط تست به عهده خود برنامه‌نویس است و شما به عنوان یک برنامه‌نویس باید مشخص کنید که چه تعداد شرط تست مورد نیاز است!
- برای به دست آوردن اطلاعات بیشتر کار با کتابخانه pytest می‌توانید به مستندات pytest رجوع کنید.

- بیایید مروری به قبل داشته باشیم. کد hello.py زیر را در نظر داشته باشید:

```
hello.py

def main():
    name = input("What's your name? ")
    hello(name)

def hello(to="world"):
    print("hello,", to)

if __name__ == "__main__":
    main()
```

- در اینجا هدف ما آزمایش کردن نتیجه‌ی تابع `hello` خواهد بود.

- قطعه کد test_hello.py زیر را در نظر داشته باشید:

```
test_hello.py

from hello import hello


def test_hello():
    assert hello("David") == "hello, David"
    assert hello() == "hello, world"
```

به این کد دقت کنید، آیا به نظر شما این رویکرد به درستی عمل خواهد کرد؟ اگر جواب شما به این سوال منفی است، چه دلایلی برای آن دارید؟

مشکل این است که تابع `hello` شما در hello.py به کمک دستور `print` خروجی را چاپ می‌کند و چیزی برای شما برنمی‌گرداند، به همین دلیل نمی‌توانید از تابع انتظار خروجی داشته باشید و نوشتار کد به صورت قسمت بالا، به ارور منتهی می‌شود.

- برای حل این مشکل ما می‌توانید تابع `hello` را به صورت زیر تغییر دهیم:

```hello.py
def main():
    name = input("What's your name? ")
    print(hello(name))

def hello(to="world"):
    return f"hello, {to}"

if __name__ == "__main__":
    main()
```

- بعد از انجام این تغییرات تابع `hello` برای ما یک string بر می‌گرداند و این بدان معناست که می‌توانیم از pytest برای تست کردن تابع `hello` استفاده کنیم.
- با اجرا کردن `pytest test_hello.py` خواهیم دید که تست ها به درستی پاس می‌شوند.

- با روشی که کمی قبل تر به آن اشاره کردیم می‌توانید حالت های مختلف تست کیس را در توابع متفاوت پیاده سازی کنید، برای مثال در اینجا از `test_default` و `test_argument` به عنوان توابع پیاده سازی تست استفاده کردیم.

test_hello.py

```
from hello import hello


def test_default():
    assert hello() == "hello, world"


def test_argument():
    assert hello("David") == "hello, David"
```

- استفاده از Unit test (تست هر واحد) با استفاده از چندین فایل تست در یک پوشه روشی بسیار رایج است. این روش به شما کمک می‌کند تا با استفاده از یک دستور بتوانید تمامی تست هایی که در یک پوشه دارید را به یکباره اجرا کنید.
- ابتدا، در پنجره ترمینال، دستور `mkdir test` را اجرا کنید تا پوشه ای به نام test ایجاد شود.
- سپس، برای ایجاد یک فایل تست در این پوشه، در پنجره ترمینال دستور `code test/hello.py` را وارد کنید. توجه کنید که `test/` به ترمینال می‌گوید که hello.py را در پوشه‌ی test ایجاد کند.
- در پنجره ویرایشگر متن، فایل را به گونه ای ویرایش کنید که شامل کد زیر باشد. توجه کنید که ما همانند قبل در حال ایجاد یک تست هستیم:

hello.py

```
from hello import hello

def test_default():
    assert hello() == "hello, world"

def test_argument():
    assert hello("David") == "hello, David"
```

- دقت داشته باشید که pytest به ما اجازه‌ی اجرای تست‌ها به صورت یک پوشه و فقط با استفاده از این فایل (یا یک سری از فایل ها) را نمی‌دهد، مگر با وجود یک فایل خاص با نام `__init__.py`
- این فایل باعث میشود تا pytest بتواند فایل های شما در یک پوشه را شناسایی کند و در صورت نبود این فایل، توانایی اجرای پوشه شما را نخواهد داشت.
- در پنجره ترمینال خود، با تایپ دستور `code test/__init__.py` این فایل را ایجاد کنید. توجه کنید (__) underscores در هر دو طرف کلمه `init` وجود داشته باشد.
- حتی اگر فایل `__init__.py` را خالی بگذارید، pytest می‌داند کل پوشه ای که شامل `__init__.py` هست دارای تست‌هایی می‌باشد که قابل اجرا هستند.
- حال با تایپ `pytest test` در ترمینال، می‌توانید کل پوشه تست کد را اجرا کنید.
- شما می‌توانید با مطالعه مستندات pytest درباره مکانیزم های `import`، بیشتر در رابطه با آن یاد بگیرید.

در این هفته مباحث زیر را مورد بررسی قرار دادیم:

- Unit Tests ✓
- `assert` ✓
- pytest ✓
- Testing Strings ✓
- Organizing Tests into Folders ✓

حالا یک توانایی به لیست رو به افزایش توانایی های شما در زبان پایتون اضافه شد. این رشد ادامه داره...

CS50x Iran

Harvard's Computer Science 50x Iran