

Memulai pengujian dengan Python

Memulai pengujian dengan Python menggunakan modul unittest. Pelajari konsep pengujian lain seperti pengujian fungsional dan integrasi.

Tujuan pembelajaran

Di akhir modul ini, Anda akan dapat:

- Tulis tes menggunakan unittest: Modul pengujian pustaka standar Python
- Mengidentifikasi dan memperbaiki kegagalan dengan membaca laporan kegagalan tes
- Membedakan antara tipe pengujian unit, fungsional, dan integrasi

Pengantar

Pengujian adalah praktik inti dalam teknik profesional. Memiliki pemahaman yang baik tentang pengujian dan memanfaatkan kekuatannya sangat penting untuk mengembangkan perangkat lunak yang kuat. Namun, mudah untuk kewalahan dengan menguji dirinya sendiri. Ada berbagai jenis pengujian dan beberapa alat dan perpustakaan yang mencoba mencapai tujuan yang sama.

Modul ini menjelaskan beberapa konsep di balik pengujian, terutama bagaimana mereka ada di Python. Dari unittest (modul perpustakaan standar Python) ke berbagai jenis pengujian dan kapan harus menerapkan teknik tersebut.

Skenario

Bayangkan Anda bertanggung jawab atas proyek Python dengan tes yang ditulis menggunakan unittest modul. Tidak ada banyak tes dan manajer telah meminta Anda untuk mengevaluasi dan mengusulkan strategi pengujian yang berbeda untuk meningkatkan proyek.

Apa yang akan Anda pelajari

Dalam modul ini, Anda akan memahami cara kerja modul pustaka standar Python dan apa saja jenis pengujian yang berbeda bersama dengan tantangannya. Ini akan memungkinkan Anda untuk:

- Bekerja dengan tes yang ada atau baru yang unittest ditulis dengan modul.
- Memahami perbedaan antara jenis pengujian dan kapan harus menerapkannya
- Jelaskan beberapa tantangan umum saat pengujian

Apa tujuan utamanya

Pada akhir modul ini, Anda akan merasa nyaman mendiskusikan strategi pengujian yang berbeda dan mengidentifikasi tantangan. Selain itu, Anda akan dapat bekerja dengan tes yang `unittest` ada atau menulis yang baru.

Memahami pengujian Python dengan modul `unittest`

Python memiliki modul pengujian yang disebut `unittest` di perpustakaan standarnya. Berada di perpustakaan standar berarti modul disertakan dalam Python itu sendiri, jadi tidak perlu menginstal apa pun untuk menggunakannya.

Adalah umum untuk melihat file pengujian yang mengimpor `unittest` dan menguji class yang menggunakan perpustakaan untuk menerapkan tes.

Class dan warisan adalah dasar untuk digunakan `unittest` untuk menulis tes. Jadi, tidak mungkin untuk menulis fungsi pengujian atau tes lain yang tidak menggunakan class dasar dari `unittest`.

Tulis tes menggunakan `unittest`

Menulis tes dengan `unittest` mengharuskan mengimpor modul dan membuat setidaknya satu class yang mewarisi dari `unittest.TestCase` class. Berikut ini adalah bagaimana tes contoh terlihat dalam file yang disebut `test_assertions.py`:

```
import unittest

class TestAssertions(unittest.TestCase):

    def test_equals(self):
        self.assertEqual("one string", "one string")

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

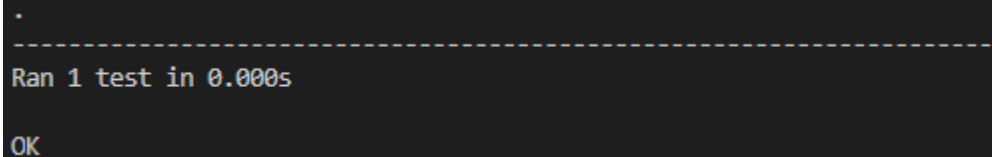
Ada beberapa item penting dalam file yang diperlukan agar tes berfungsi. Dari konvensi penamaan hingga method tertentu yang digabungkan memungkinkan file pengujian dieksekusi.

Jalankan pengujian

Ada dua cara untuk menjalankan file uji. Mari kita lihat lagi di akhir file *test_assertions.py*, di mana `unittest.main()` panggilan memungkinkan menjalankan tes dengan mengeksekusi file dengan Python:

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

Uji coba dimungkinkan karena `if` blok di akhir, di mana kondisinya hanya terpenuhi saat menjalankan skrip langsung dengan Python. Mari kita periksa output saat menjalankannya dengan cara itu:



```
.  
-----  
Ran 1 test in 0.000s  
OK
```

Konvensi penamaan

Nama class dan method mengikuti konvensi uji. Konvensinya adalah bahwa mereka perlu diawali dengan `test`. Meskipun tidak diperlukan, class tes menggunakan casing unta, dan method pengujian lebih rendah, dan kata-kata terpisah dengan garis bawah. Misalnya, berikut ini adalah bagaimana pengujian untuk akun pelanggan yang memverifikasi pembuatan dan penghapusan dapat terlihat:

```
class TestAccounts(unittest.TestCase):  
  
    def test_creation(self):  
        self.assertTrue(account.create())  
  
    def test_deletion(self):  
        self.assertTrue(account.delete())
```

Kelas uji atau metode yang tidak mengikuti konvensi ini tidak akan dijalankan. Meskipun mungkin tampak seperti masalah untuk tidak menjalankan setiap metode dalam kelas, itu dapat membantu ketika membuat kode non-tes.

Pernyataan dan metode assert

Sangat penting untuk menggunakan metode assert alih-alih fungsi bawaan Python untuk memiliki pelaporan yang assert() kaya ketika kegagalan terjadi. *Test_assertion.py* menggunakan `self.assertEqual()`, salah satu dari banyak metode khusus dari `unittest.TestCase` kelas untuk memastikan bahwa dua nilai sama:

```
self.assertEqual("one string", "one string")
```

Dalam hal ini, kedua string sama, sehingga tes berlalu. Menguji kesetaraan adalah salah satu dari banyak pernyataan berbeda yang `unittest.TestCase` ditawarkan kelas. Meskipun ada lebih dari 30 metode assert, berikut ini paling sering digunakan selain `self.assertEqual()`

- `self.assertTrue(value)`: Pastikan itu value benar.
- `self.assertFalse(value)`: Pastikan itu value salah.
- `self.assertNotEqual(a, b)`: Periksa itu a dan b tidak sama.

Kegagalan dan pelaporan

Lulus tes adalah cara yang bagus untuk memastikan kekokohan. Namun, memahami pelaporan kegagalan sangat penting untuk memperbarui dan memperbaiki kode produksi.

Dalam contoh berikutnya, saya mengubah salah satu string dari "one string" ke "other string". Kemudian, saya menjalankan tes dengan Python. Berikut adalah bagaimana output terlihat sekarang:

```
Traceback (most recent call last):
  File "f:\python\try 4.py", line 8, in <module>
    class TestAccounts(unittest.TestCase):
  File "C:\Users\Hp\AppData\Local\Programs\Python\Python39\lib\unittest\__init__.py", line 95, in __getattr__
    raise AttributeError(f"module {__name__!r} has no attribute {name!r}")
AttributeError: module 'unittest' has no attribute 'Testcase'
```

Tes menunjukkan kegagalan pernyataan karena string berbeda. Laporan ini meningkatkan kesalahan pernyataan untuk menunjukkan string tidak cocok pada beberapa karakter. Sangat membantu untuk memeriksa di mana tepatnya kesalahan terjadi karena mempersempit area di mana masalahnya. Dalam hal ini, ada dalam file *test_assertions.py* di baris enam.

Latihan - Tulis tes unit dengan modul `unittest`

Dalam latihan ini, Anda akan memanfaatkan `unittest` modul, yang termasuk dalam pustaka standar Python untuk menulis tes dan memperbaiki bug. Lulus tes memastikan bahwa kode berperilaku seperti yang diharapkan, meningkatkan kepercayaan diri bahwa perubahan baru tidak melanggar fungsi sebelumnya.

Langkah 1 - Tambahkan file untuk latihan ini

1. Menggunakan konvensi nama file Python untuk file uji, buat file uji baru. Beri nama file uji `test_exercise.py` dan tambahkan kode berikut:

```
def str_to_bool(value):
    true_values = ['y', 'yes']
    false_values = ['no', 'n']

    if value in true_values:
        return True
    if value in false_values:
        return False
```

Fungsi `str_to_bool()` mengambil string sebagai input, dan tugasnya adalah mengembalikan `True` atau `False` tergantung pada isi string. Misalnya jika string itu `y` akan kembali `True`. Demikian pula, jika string, `no` itu akan kembali `False`.

2. Dalam file yang sama, tambahkan tes untuk fungsi:

```
import unittest

class TestStrToBool(unittest.TestCase):

    def test_y_is_true(self):
        result = str_to_bool('y')
        self.assertTrue(result)

    def test_yes_is_true(self):
        result = str_to_bool('Yes')
        self.assertTrue(result)

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

Langkah 2 - Jalankan tes dan identifikasi kegagalan

Sekarang file pengujian memiliki fungsi untuk menguji dan beberapa tes untuk memverifikasi perilakunya. Sudah waktunya untuk menjalankan tes dan bekerja dengan kegagalan.

- Jalankan/running file dengan Python

Hasil running harus selesai dengan satu lulus tes dan satu kegagalan, dan output kegagalan harus mirip dengan output berikut:

```
NameError: name 'str_to_bool' is not defined

=====
ERROR: test_yes_is_true (__main__.TestStrToBool)
-----
Traceback (most recent call last):
  File "f:\python\unittest\test.py", line 10, in test_yes_is_true
    result = str_to_bool('Yes')
NameError: name 'str_to_bool' is not defined

-----
Ran 2 tests in 0.003s

FAILED (errors=2)
```

Analisa hasil eksekusi:

dari hasil eksekusinya terdapat satu kegagalan dan juga satu lulus tes. Hal itu bisa terjadi karena disaat kita memasukkan `str_to_bool` nya menggunakan `Yes` huruf besar di `Y` sedangkan `tru_values` nya `'yes'`

Sekarang kesalahan menunjuk ke input string yang dikapitalisasi `'Yes'`, bug perlu diperbaiki. Meskipun tes dapat diperbarui untuk menggunakan kata huruf kecil (`'yes'`) untuk membuatnya lulus, perbaikan akan membuat fungsi bekerja dengan casing apa pun. Itu berarti bahwa salah satu opsi ini harus berfungsi: `YES`, `Yes`, `, yes`, `yES`, `yeS`, `Y`, `y`.

1. `str_to_bool()` Perbarui fungsi untuk menetapkan `value` kembali variabel menjadi huruf kecil menggunakan `value.lower()`. Fungsi yang diperbarui akan terlihat seperti ini:

```
def str_to_bool(value):  
    value = value.lower()  
    true_values = ['y', 'yes']  
    false_values = ['no', 'n']  
  
    if value in true_values:  
        return True  
    if value in false_values:  
        return False
```

2. Jalankan program dan tampilkan hasilnya di sini

```
..  
-----  
Ran 2 tests in 0.001s  
OK
```

Analisa Anda terhadap hasil eksekusi di atas: Dikarenakan value yang diuji dijadikan lowercase, maka terjadi kesamaan antara value yang diuji dan value dalam unittest. Hasilnya unittest melewati blok program itu karena hasil dari unittest tersebut karena tidak terjadi failure.

Langkah 4 - Tambahkan kode baru dengan tes

Bagian dari gagasan bekerja dengan tes adalah mendapatkan validasi untuk perubahan. Bahkan jika perubahan memperbaiki kode yang ada atau menambahkan fungsionalitas baru, tes meningkatkan kepercayaan diri bahwa semuanya masih berfungsi seperti yang diharapkan.

Dalam hal ini, fungsi yang diuji hanya bekerja dengan string, jadi menggunakan jenis lain sebagai input akan menyebabkan pengecualian yang tidak ditangani dinaikkan.

1. Perbarui fungsi sehingga menimbulkan `AttributeError` jika nilai non-string digunakan. Ini dapat dideteksi dengan `AttributeError` menangkap saat memanggil `value.lower()` karena hanya string yang memiliki `lower()` metode:

```
def str_to_bool(value):
    try:
        value = value.lower()
    except AttributeError:
        raise AttributeError(f"{value} must be of type string")
    true_values = ['y', 'yes']
    false_values = ['no', 'n']

    if value in true_values:
        return True
    if value in false_values:
        return False
```

- Gunakan metode `assert` baru dari `unittest.TestCase` di kelas tes. Tes baru ini harus memverifikasi bahwa `AttributeError` kenaikan pada input non-string:

```
def test_invalid_input(self):
    with self.assertRaises(AttributeError):
        str_to_bool(1)
```

- Jalankan semua tes lagi dengan mengeksekusi skrip dengan Python untuk mendapatkan output berikut:



```
***
-----
Ran 3 tests in 0.001s
OK
```

Analisa Anda terhadap hasil eksekusi di atas jika dibandingkan dengan hasil pada Langkah 3 : Output dari kode program tersebut tidak menunjukkan adanya error atau failure, karena semua baris program bekerja secara benar dan tidak menunjukkan adanya kesalahan. Jika saja penulis kode tidak menggunakan tipe data string maka akan muncul pesan error seperti pada blok try-except.

Periksa pekerjaan Anda

Pada titik ini Anda harus memiliki file uji Python yang dinamai mirip dengan *test_exercise.py* dengan yang berikut:

- Sebuah `str_to_bool()` fungsi
- Blok try/except dalam `str_to_bool()` fungsi yang menangkap `AttributeError`

- Kelas `TestStrToBool()` tes yang mewarisi dari `unittest.TestCase`
- Setidaknya ada tiga metode pengujian yang menguji input pada `str_to_bool()` fungsi

Tantangan dengan pengujian

Dalam proyek perangkat lunak, pengujian bisa rumit dan terasa luar biasa. Ketika proyek yang ada tidak dibangun dengan banyak tes, kode cenderung kompleks dan kurang kuat. Memahami masalah dan potensi jebakan dengan pengujian sangat penting untuk proyek perangkat lunak yang sukses.

Kekurangan pengujian

Ketika proyek perangkat lunak dibangun tanpa tes (atau jumlah tes yang tidak memadai), kode memburuk, menjadi rapuh, dan sulit dipahami. Tes membuat kode bertanggung jawab atas perilaku mereka, dan ketika tes diperkenalkan, itu memaksa insinyur untuk membuat kode lebih mudah untuk diuji.

Ada beberapa efek samping positif dari pengujian. Ini adalah beberapa aspek yang dapat dikenali dalam fungsi dan metode:

- Panjangnya tidak lebih dari selusin baris.
- Cenderung memiliki tanggung jawab tunggal daripada melakukan banyak hal yang berbeda.
- Mereka memiliki jumlah input tunggal atau minimal, bukan banyak argumen dan nilai.

Input pendek, tanggung jawab tunggal, dan minimal membuat kode lebih mudah dipahami, dipelihara, dan diuji. Ketika pengujian tidak terlibat, adalah umum untuk melihat fungsi yang lebih kecil tumbuh menjadi beberapa ratus baris, melakukan banyak hal. Ini terjadi karena tidak ada pertanggungjawaban yang mencegah kompleksitas yang tidak perlu.

Kode warisan

Salah satu cara untuk memikirkan kode yang belum teruji adalah dengan melabelinya sebagai *warisan*. Adalah umum untuk menemukan kode yang belum teruji dalam proyek perangkat lunak. Hal ini dapat terjadi karena berbagai jenis alasan seperti kurangnya pengalaman dengan pengujian atau praktik

pengembangan yang tidak memungkinkan waktu yang cukup untuk mempertimbangkan pengujian sebagai bagian dari memproduksi perangkat lunak.

Seperti yang telah disebutkan, salah satu karakteristik kode yang belum teruji adalah sulit untuk dipahami dan seringkali bisa sangat kompleks karena alasan yang sama. Efek samping dari semua masalah ini adalah bahwa hal itu membuat kode bahkan lebih sulit untuk ditangani yang pada gilirannya menyebabkan fungsi (atau metode) untuk terus tumbuh dengan lebih banyak logika dan jalur kode yang lebih terjalin.

Semakin kecil fungsinya, semakin mudah untuk diuji.

Tes yang lambat dan tidak dapat diandalkan

Meskipun test suite yang ada sangat bagus untuk memulai, mungkin bermasalah ketika berisi tes yang lambat atau tidak dapat diandalkan. Seorang pengembang akan lebih cenderung untuk menjalankan tes sering jika ini memberikan umpan balik yang cepat. Jika Anda harus bekerja dengan test suite yang memakan waktu berjam-jam, bukan menit (atau detik), Anda mungkin akan menunggu sampai saat terakhir untuk menjalankan test suite. Ketika loop umpan balik lambat, proses pengembangan akan dikompromikan.

Demikian pula, Anda mungkin menemukan situasi di mana tes dapat gagal tanpa alasan yang jelas. Tes yang tidak dapat diandalkan kadang-kadang disebut *terkelupas*. Tanggung jawab utama dari test suite adalah untuk menunjukkan bahwa kode yang diuji adalah memenuhi harapan yang ditetapkan oleh tes itu sendiri. Jika tes tidak dapat diandalkan, Anda tidak dapat benar-benar mengetahui apakah kode perlu diubah atau jika patch telah memperkenalkan regresi.

Tes yang tidak dapat diandalkan (atau terkelupas) harus diperbaiki atau dihapus sepenuhnya dari suite. Tidak setiap tes bisa cepat, dan beberapa jenis pengujian seperti tes integrasi bisa lambat. Tetapi memiliki suite uji yang solid yang memberikan umpan balik cepat sangat penting.

Kurangnya otomatisasi

Ketika pengujian tidak terjadi dengan cara otomatis, mudah untuk melupakan apa yang harus diuji dan menjadi rawan kesalahan. Bahkan pada proyek perangkat lunak kecil mudah dilupakan jika mengubah kondisi dalam suatu fungsi dapat memiliki efek samping (negatif) di tempat lain di basis kode. Ketika pengembang perangkat lunak tidak bertugas mengingat apa dan kapan harus menguji karena semuanya terjadi secara otomatis, maka kepercayaan pada perangkat lunak, tes, dan keseluruhan sistem meningkat.

Otomatisasi adalah tentang menghapus tugas berulang dan mengurangi langkah-langkah yang diperlukan untuk suatu tindakan terjadi. Dalam pengujian, otomatisasi dapat terjadi ketika kode didorong ke repositori jarak jauh, atau bahkan sebelum digabungkan. Mencegah kode yang rusak masuk ke kode produksi tanpa pemeriksaan manual sangat penting untuk aplikasi yang kuat.

Perkakas uji

Beberapa tantangan berasal dari kurangnya pengujian, sementara yang lain terkait dengan pengalaman pengujian yang buruk dan tidak tahu teknik apa yang harus diterapkan. Kami telah menyebutkan *kode lama* dalam modul ini, dan bagaimana kode yang belum teruji dapat terus tumbuh dalam ukuran dan kompleksitas. Meskipun Anda dapat mengatakan bahwa fungsi besar mungkin belum teruji, ada alat cakupan pengujian yang dapat secara akurat mengetahui jalur kode apa yang diuji atau tidak dicakup oleh tes yang ada.

Mengandalkan perkakas dan pengujian perpustakaan adalah cara yang bagus untuk mengurangi jumlah upaya yang diperlukan untuk menghasilkan tes yang baik. Tergantung pada bahasa dan aplikasi yang sedang diuji, Anda harus menemukan solusi yang dapat membuat pengujian lebih mudah dan lebih kuat. Berikut adalah beberapa contoh tentang apa alat-alat itu seharusnya:

- Alat cakupan pengujian untuk melaporkan jalur kode yang diuji dan belum teruji
- Pelari uji, yang dapat mengumpulkan, mengeksekusi, dan menjalankan ulang tes tertentu
- Lingkungan CI/CD, yang dapat secara otomatis menjalankan pengujian dan mencegah kode yang rusak digabungkan atau disebarkan

Jenis pengujian dan cara menggunakannya

Ada berbagai jenis pengujian, yang dapat menyebabkan beberapa kebingungan ketika mencoba untuk memulai pengujian. Mengetahui apa jenis pengujian ini dan bagaimana mereka berbeda satu sama lain adalah cara yang solid untuk menerapkan strategi pengujian yang kuat.

Catatan

Perbedaan antara unit, integrasi, dan pengujian fungsional bisa menjadi kontroversial. Deskripsi ini dimaksudkan sebagai pedoman. Yang paling penting adalah konsisten dengan proyek dan pengembang lainnya.

Pengujian Unit

Fokus utama dari pengujian unit adalah menguji bagian terkecil dari logika kode yang mungkin. Efek samping dari jenis pengujian ini adalah kecepatan. Tes unit biasanya akan berjalan sangat cepat karena (idealnya) tidak ada sumber daya eksternal seperti database, situs web, atau panggilan jaringan yang diperlukan.

Menguji fungsi dan metode yang sangat panjang, dengan beberapa kondisi logis akan sulit untuk diuji unit. Pengujian unit memaksa pengembang untuk memikirkan kompleksitas dalam kode dan menjaganya seminimal mungkin. Secara umum, semakin pendek fungsi atau metodenya, semakin mudah untuk mengujinya.

Tidak ada aturan keras tentang pengujian unit, tetapi secara umum berikut ini adalah norma yang umum diterima tentang tes unit:

- Mereka menguji bagian terkecil dari logika yang mungkin.
- Fungsi, metode, atau kelas diuji terisolasi mungkin, menghindari kebutuhan untuk mengatur fungsi lain, metode, atau kelas sebagai persyaratan.
- Tidak ada layanan eksternal seperti database atau layanan jaringan yang diperlukan agar mereka dapat berjalan.

Pengujian integrasi

Ketika membahas jenis pengujian ini, biasanya tentang menguji logika yang akan berinteraksi dengan bagian logika lain dalam proyek kode. Terkadang, tes ini akan memerlukan koneksi ke database atau layanan eksternal lainnya.

Misalnya, fungsi yang memeriksa nama pengguna dan kata sandi mungkin perlu terhubung ke database untuk memverifikasi data yang ada. Tes untuk fungsi itu mungkin memerlukan database dengan informasi yang ada. Jenis tes ini akan membutuhkan pengaturan yang sedikit lebih kompleks daripada tes unit dan mungkin membutuhkan waktu lebih lama untuk dijalankan.

Ini adalah beberapa norma yang diterima secara umum tentang pengujian integrasi:

- Tidak secepat tes unit, dan mereka cenderung membutuhkan lebih banyak pengaturan di muka sebelum menjalankan
- Fungsi, metode, atau kelas diuji dengan fungsionalitas dalam fungsi, metode, atau kelas lain.
- Layanan eksternal dapat digunakan, tetapi tidak semua layanan untuk aplikasi

Pengujian fungsional

Pengujian fungsional biasanya membutuhkan menjalankan aplikasi secara keseluruhan. Untuk situs web, tes fungsional mungkin memerlukan server web, database, dan layanan lain yang diperlukan agar aplikasi dapat berjalan. Idennya adalah untuk mereplikasi aplikasi sedekat mungkin dengan lingkungan produksi.

Karena tidak selalu mungkin untuk mereplikasi lingkungan produksi secara akurat, perawatan khusus perlu dimasukkan ke dalam kelemahan pengaturan. Misalnya, jika sebuah situs web memiliki satu terabyte data dalam produksi, mungkin tidak apa-apa untuk menguji dengan subset data produksi. Namun, tidak disarankan untuk menggunakan database tertanam seperti SQLite daripada database PostgreSQL yang sama yang digunakan dalam produksi. Perbedaan versi layanan eksternal seperti database dapat menyebabkan tes lulus tetapi gagal dalam produksi.

Ini adalah beberapa aspek pengujian fungsional:

- Tidak secepat tes unit, dan mereka cenderung membutuhkan lebih banyak pengaturan di muka sebelum menjalankan
- Fungsi, metode, atau kelas diuji dengan fungsionalitas dalam fungsi, metode, atau kelas lain.
- Layanan eksternal dapat digunakan, tetapi tidak semua layanan untuk aplikasi

Karena tes fungsional membutuhkan lebih banyak layanan, reproduksi lingkungan produksi, dan pengaturan yang paling kompleks dari jenis pengujian, biasanya akan memakan waktu dan sumber daya intensif.

Tes fungsional tunggal untuk toko ritel online dapat melibatkan langkah-langkah berikut untuk diselesaikan:

1. Mendaftar untuk pengguna baru
2. Pilih produk tertentu dan tambahkan ke keranjang belanja virtual
3. Melengkapi informasi pengiriman dan penagihan
4. Pilih tombol "beli" untuk menyelesaikan pembelian
5. Verifikasi bahwa akun baru ada, informasi penagihan dan pengiriman sudah benar, dan inventaris diperbarui

Integrasi Berkelanjutan

Meskipun CI (Continuous Integration) bukan jenis pengujian, ini adalah bagian penting yang berkaitan dengan setiap jenis pengujian. Ketika rencana pengujian diberlakukan, penting untuk memiliki *sesuatu* yang akan menjalankan tes secara otomatis. Lingkungan CI adalah tempat yang tepat untuk menjalankan tes secara otomatis. Tes ini dapat berjalan sesuai jadwal, dipicu oleh peristiwa, atau dieksekusi

secara manual. Lingkungan ini biasanya akan konsisten dengan di mana kode atau proyek perlu berjalan.

Dasar untuk proses CI yang baik adalah otomatisasi. Otomatisasi adalah apa yang akan menetapkan lingkungan yang konsisten (dan terkenal). Tanpa lingkungan pengujian yang berulang dan diketahui, masalah debugging dan tes yang gagal akan memakan waktu atau bahkan tidak mungkin.

Dipicu oleh suatu peristiwa

Misalnya, jika pengembang ingin menggabungkan perubahannya ke cabang utama, sangat ideal untuk memastikan bahwa perubahan tidak akan menyebabkan kerusakan. Sistem CI dapat secara otomatis menjalankan pengujian dari cabang pengembang dan memberi tahu ketika ada kegagalan. Mencegah kode yang gagal CI untuk digabungkan adalah cara yang baik untuk meningkatkan kepercayaan diri dan kekokohan dalam sebuah proyek.

Berjalan sesuai jadwal

Seperti yang telah disebutkan, otomatisasi adalah bagian penting dari proses CI. Biasanya, dengan pengaturan jenis pengujian apa pun, dependensi harus diinstal agar pengujian berfungsi. Menggunakan jadwal untuk uji coba (misalnya lari malam) memastikan bahwa proyek dibangun dengan benar bahkan ketika dependensi berubah. Uji coba terjadwal berulang juga dapat membantu menjalankan tes yang membutuhkan waktu lama untuk diselesaikan ketika sumber daya lebih mudah tersedia. Jika proyek perangkat lunak memiliki tes fungsional yang dapat memakan waktu beberapa jam untuk diselesaikan, akan lebih efisien untuk menjalankan tes tersebut di malam hari sehingga tim dapat mengerjakan kegagalan hal pertama di pagi hari.

Pemicu manual

Akhirnya, meskipun uji coba terjadwal dan berbasis acara berguna, ada baiknya untuk dapat menjalankan test suite secara manual. Misalnya, ketika test suite diketahui memiliki masalah, pengembang dapat mencoba perbaikan dengan melakukan rerunning run yang sebelumnya gagal. Loop umpan balik ini memungkinkan perbaikan untuk diuji tanpa perlu acara atau jadwal tertentu. Selanjutnya, ci run mungkin memungkinkan perubahan konfigurasi untuk menjalankan tes, dan ini dapat diubah ad-hoc untuk memastikan kondisi tertentu terpenuhi.

Uji pengetahuan

Selesai200 XP

- 4 menit

1. Apa satu efek samping umum untuk fungsi ketika tidak ada tes yang terlibat?

- ☐ Fungsi akan berjalan lebih cepat karena tidak perlu menggunakan tes.
- ☐ Fungsi dapat tumbuh dalam kompleksitas dan ukuran.
- ☐ Fungsi akan berjalan lebih lambat dan memiliki lebih banyak argumen.

2. Apa aspek yang dapat dikenali dalam fungsi dan metode saat pengujian terlibat?

- ☐ Mereka cenderung memiliki tanggung jawab tunggal daripada melakukan banyak hal yang berbeda.
- ☐ Mereka lebih fleksibel, memungkinkan lebih banyak argumen
- ☐ Mereka tidak cenderung meningkatkan pengecualian

3. Apa doa Python yang benar untuk menggunakan penemuan otomatis uji?

- ☒ `python -m autodiscovery`
- ☐ `python -m discovery`
- ☐ `python -m unittest`