**Pengantar Unit Testing**

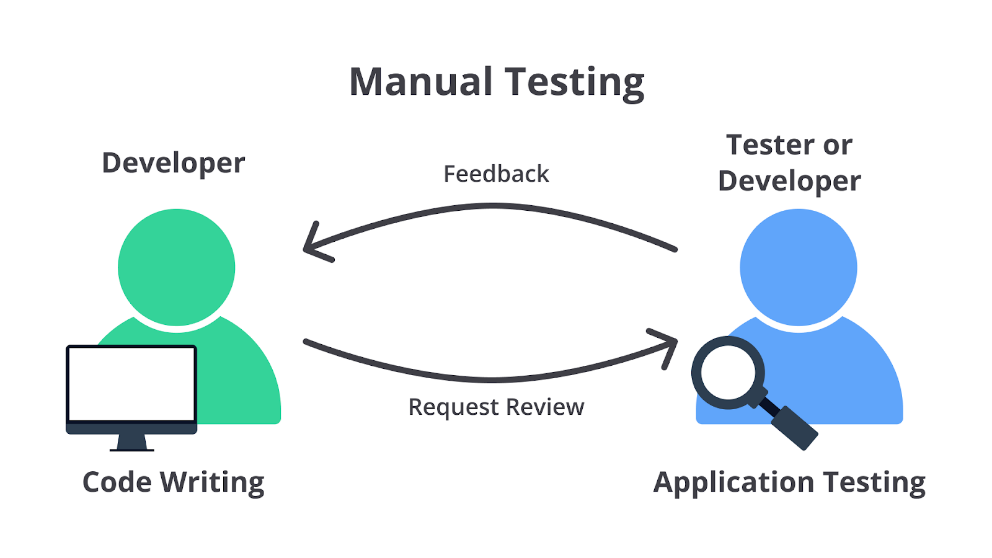
Sampai pada tahap ini, sudah banyak modul pemrograman Python yang telah dipelajari. Kita telah mengenal operasi-operasi dasar dalam Python, seperti perulangan, fungsi, hingga OOP pada Python.

Saat Anda membangun aplikasi menggunakan Python dan aplikasi yang dikembangkan semakin kompleks, dependensi akan muncul. Artinya, satu atau lebih fungsi digunakan oleh fungsi lain. Bahkan, ketika kita mulai membangun aplikasi dengan rekan kita, kita membuat fungsi yang digunakan oleh rekan, ataupun sebaliknya.

Pada saat membuat fungsi baru ataupun mengubah fungsi yang sudah ada, tentunya perlu dipastikan bahwa fungsionalitas aplikasi yang sebelumnya tidak terganggu dengan adanya perubahan baru tersebut. Bagaimana jika fungsionalitas bukan hanya lima atau sepuluh, tetapi lebih dari itu? Tentu menyulitkan sekali untuk mengeceknya satu per satu setiap kita melakukan perubahan.

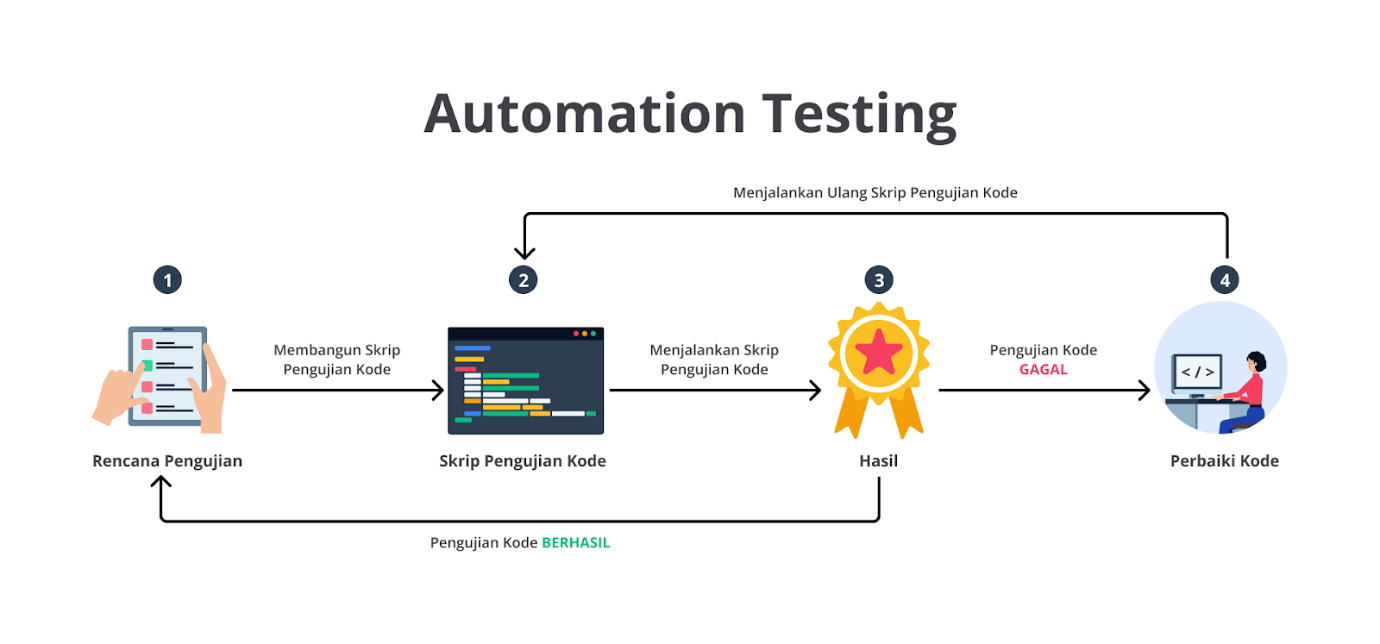
Di sinilah kita butuh pengujian (*test*) untuk fungsi-fungsi tersebut. Pengujian sebenarnya dapat dibedakan menjadi dua tipe, yaitu **manual** dan **otomatis**.

1. Manual testing adalah proses pengujian yang dilakukan oleh seseorang yang ditunjuk sebagai tester atau bahkan developer lainnya.



Tanpa sadar, sebenarnya ketika Anda menjalankan program pertama kali lalu mengecek bahwa output-nya sesuai atau tidak, itu merupakan pengujian manual.

1. Testing otomatis merupakan hal yang sebaliknya. Ini adalah pengujian yang dilakukan secara otomatis terhadap kode-kode yang kita bangun berdasarkan rencana pengujian (*test plan*).



Rencana pengujian terdiri dari bagian aplikasi yang ingin diuji, urutannya, dan tanggapan atau output yang diharapkan. Alur pengujian otomatis secara umum dimulai dari menyusun rencana pengujian, lalu membangun kode tes dan menjalankan kode tes tersebut. Jika kode tes gagal, kita perlu memperbarui kode; jika kode tes berhasil, kita melaju ke pengujian selanjutnya hingga selesai.

Tidak hanya sekadar manual dan otomatis, dalam dunia testing yang begitu luas, Anda akan menemui berbagai jenis testing. Beberapa di antaranya adalah **unit testing** dan **integration testing**.

Integration testing adalah pengujian yang bertujuan untuk menguji suatu sistem sebagai satu kesatuan. Bayangkan Anda sedang mengecek lampu motor, tentu hal pertama yang dilakukan adalah menyalakan motor. Lalu, Anda melihat lampu motor tersebut yang sempat menyala, tetapi perlahan mati. Anda kemudian bertanya, mengapa lampunya mati?

Kejadian yang dijelaskan di atas erat dengan konsep integration testing karena dengan menyalakan motor, kita dapat menguji seluruh bagian motor lain, seperti lampunya.

Unit testing adalah pengujian yang lebih spesifik dan fokus terhadap bagian-bagian kecil. Bayangkan ketika mengecek lampu motor dan ternyata ia tidak menyala, Anda perlu mengecek lampu tersebut; apakah rusak? Atau ada kabel dari lampu tersebut yang tidak berfungsi? Hal-hal yang lebih spesifik tersebut adalah unit testing. Dalam pemrograman, bagian-bagian kecil tersebut berupa fungsi, kelas, dan sebagainya.

Pada materi ini, kita akan mempelajari unit testing menggunakan salah satu library bawaan Python, yaitu **unittest**. Unit test adalah proses pengujian perangkat lunak yang memastikan setiap unit/fungsi dari program teruji. Jika fungsionalitas dari aplikasi yang kita bangun terdiri dari prosedur-prosedur dan fungsi-fungsi yang kita tulis, kita perlu melakukan unit test untuk setiap prosedur atau fungsi yang ada.

Layaknya sebuah framework pengujian, library unittest mendukung beberapa hal esensial berikut.

* Pengujian secara otomatis.
* Kode awal proses (setup) dan akhir proses (shutdown) yang dapat digunakan ulang.
* Penyatuan sejumlah pengujian dalam sebuah koleksi.
* Terpisahnya framework pengujian dari framework pelaporan (*reporting*).

Library unittest mendukung sejumlah konsep penting yang berorientasi objek, antara lain berikut.

* *Test fixture* merepresentasikan **persiapan**yang dibutuhkan untuk melakukan satu pengujian atau lebih serta proses pembersihannya (*cleanup*). Beberapa contohnya adalah menyiapkan basis data pengujian, direktori pengujian, atau mengaktifkan sebuah proses server.
* *Test case* adalah sebuah unit dari pengujian ketika ia mengecek sejumlah respons dari sebagian kelompok masukan. Library unittest menyediakan basis class dan TestCase yang akan digunakan untuk membuat kasus pengujian baru.
* *Test suite* adalah sebuah koleksi dari kasus-kasus pengujian, koleksi dari test suite itu sendiri, atau gabungan keduanya. Hal ini berguna untuk mengumpulkan pengujian-pengujian yang akan dieksekusi bersama.

*Test runner* adalah komponen yang akan mengatur (*orchestrates*) eksekusi dari pengujian-pengujian dan menyediakan keluaran untuk pengguna. Dalam hal ini, runner dapat menggunakan tampilan grafis, tampilan tekstual, atau mengembalikan nilai spesial yang menyatakan hasil dari pengujian.

**Penerapan Unit Test dengan Library unittest**

Sekarang mari kita pelajari cara implementasinya, materi kali ini akan menggunakan PyCharm sebagai IDE-nya. Silakan ikuti sembari menggunakan IDE Anda, seperti Visual Studio Code atau PyCharm.

Tulis kode ini pada IDE PyCharm dan simpan kode ini dalam format .py. Lalu, jalankan pada Command Prompt di perangkat Anda.

|  |
| --- |
| import unittest    class TestStringMethods(unittest.TestCase):      # Ini adalah test case pertama (1)      def test\_strip(self):          self.assertEqual('www.dicoding.com'.strip('c.mow'), 'dicoding')        # Test case kedua (2)      def test\_isalnum(self):          self.assertTrue('c0d1ng'.isalnum())          self.assertFalse('c0d!ng'.isalnum())        # Test case ketiga (3)      def test\_index(self):          s = 'dicoding'          self.assertEqual(s.index('coding'), 2)          # cek s.index gagal ketika tidak ditemukan          with self.assertRaises(ValueError):              s.index('decode')    if \_\_name\_\_ == '\_\_main\_\_':      # Test Runner      unittest.main() |

Mari kita bahas satu per satu dari kode di atas.

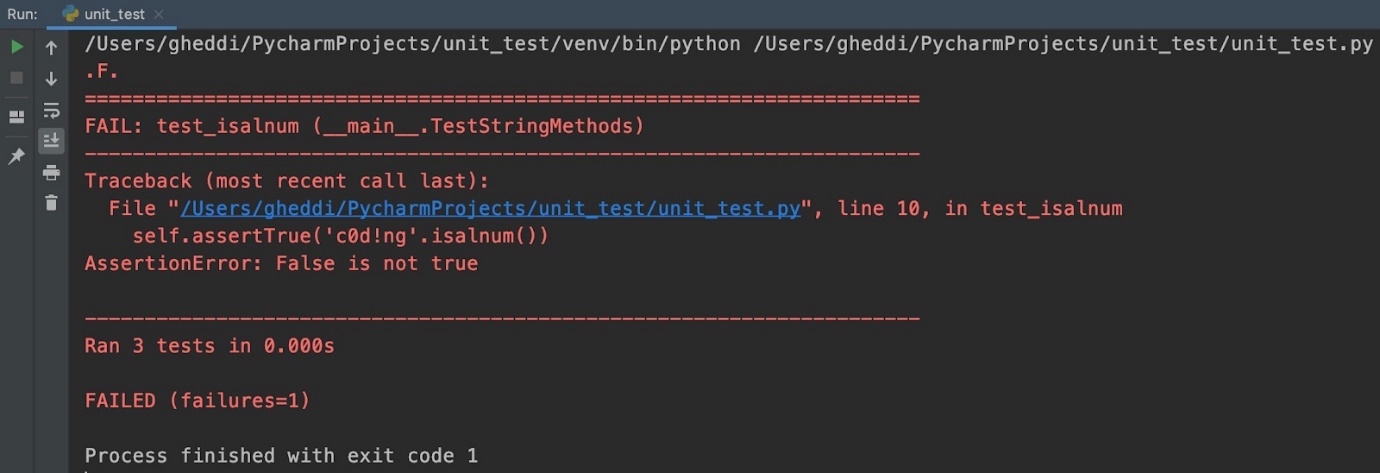
* Kelas TestStringMethods adalah sebuah kelas yang merupakan turunan (*subclass*) dari class unittest.TestCase sehingga proses tes dapat dilangsungkan tanpa banyak implementasi lain.
* Ada tiga metode pada class tersebut yang semua namanya diawali dengan kata test. Hal ini merupakan konvensi (aturan) yang wajib diikuti untuk menginformasikan ke test runner bahwa sejumlah metode tersebut merepresentasikan tes yang akan dioperasikan.
* Pada setiap metode, pengujian dilakukan dengan pemanggilan assert. Pada metode test\_strip pengecekan kesamaan menggunakan assertEqual dilakukan untuk memastikan bahwa 'www.dicoding.com'.strip('c.mow') sama dengan ‘dicoding’.
* Pada metode test\_isalnum pengecekan bahwa fungsi bernilai benar (True) dilakukan dengan assertTrue untuk memastikan jika 'c0d1ng'.isalnum() bernilai benar, yakni ‘cOd1ng’ adalah betul bertipe alfanumerik. Kemudian juga ada pengecekan bahwa fungsi bernilai salah (False) dengan assertFalse untuk memastikan jika 'c0d!ng'.isalnum() betul bernilai salah karena ada karakter yang bukan alfanumerik yaitu ‘!’.
* Pada metode test\_index pengecekan kesamaan dilakukan seperti sebelumnya dengan menggunakan assertEqual bahwa pencarian substring coding menempati index sama dengan 2. Kemudian juga ada pengecekan bahwa ValueError akan dibangkitkan dengan menggunakan assertRaises(ValueError), jika pencarian index tidak berhasil ditemukan pada string yang sudah ditentukan.
* Pada bagian terakhir kode, ada pemanggilan unittest.main() untuk mulai menjalankan test.

Selanjutnya, kita akan membahas hasil keluarannya. Tampak pada keluaran bahwa ada tiga tanda titik (...) yang menyatakan bahwa ketiga fungsi yang dites berhasil melewati tes. Dirangkum juga waktu pemrosesan dari total tiga tes tersebut yang berlangsung selama 0.00 detik serta di baris paling akhir adalah rangkuman bahwa semua test berlangsung sukses (OK).

Anda bisa mencoba melihat keluaran lain dengan membuat gagal salah satu test. Misalnya pada metode test\_isalnum, keduanya akan diubah menggunakan assertTrue sehingga salah satu fungsi akan gagal. Kodenya bisa Anda lihat di bawah.

|  |
| --- |
| def test\_isalnum(self):  self.assertTrue('c0d1ng'.isalnum()) # ini akan berhasil  self.assertTrue('c0d!ng'.isalnum()) # ini akan gagal |

Kemudian jalankan kembali program Anda. Keluarannya akan seperti berikut.



Berikut penjelasannya.

* Layaknya yang sudah Anda duga bahwa akan ada pengujian yang gagal sehingga tertulis .F. yang menggambarkan bahwa pengujian metode kedua gagal (FAIL).
* Berikutnya dijelaskan bahwa kegagalan ada dalam metode test\_isalnum, yaitu sebuah metode dari class \_\_main\_\_.TestStringMethods.
* Lebih jauh, diinformasikan bahwa test\_isalnum yang gagal berada pada baris ke-10 pada kode Anda, yakni pada pengecekan self.assertTrue('c0d!ng'.isalnum()) yang memang tadi kita ubah dari assertFalse. Sistem pengujian juga melaporkan bahwa pembandingannya tidak sesuai, yakni False tidak bernilai benar seperti yang diharapkan dengan adanya pengujian assertTrue.
* Rekap totalnya ada tiga tes yang dilakukan dalam 0.01 detik. Kemudian secara umum tes menghasilkan satu buah kegagalan (*failure*).

Cukup mudah dimengerti, bukan? Kita dapat melihat sendiri bahwa pengujian hasilnya gagal. Namun, kali ini gagalnya memang sesuai dengan ekspektasi kita.

Sekarang kita coba pengujian dengan contoh yang lebih nyata, misalnya kita memiliki class User dan kita akan menguji aktif atau tidaknya user dengan melihat bahwa dia terkoneksi ke basis data (db) atau tidak.

Untuk menyederhanakan kodenya dan lebih fokus pada pengujiannya, tulis simulasinya dalam satu file kode sebagai berikut.

|  |
| --- |
| import unittest    def koneksi\_ke\_db():  print("[terhubung ke db]")    def putus\_koneksi\_db(db):  print("[tidak terhubung ke db {}]".format(db))    class User:  username = ""  aktif = False    def \_\_init\_\_(self, db, username): # using db sample  self.username = username    def set\_aktif(self):  self.aktif = True    class TestUser(unittest.TestCase):  # Test Case 1  def test\_user\_default\_not\_active(self):  db = koneksi\_ke\_db()  dicoding = User(db, "dicoding")  self.assertFalse(dicoding.aktif) # tidak aktif secara default  putus\_koneksi\_db(db)    # Test Case 2  def test\_user\_is\_active(self):  db = koneksi\_ke\_db()  dicoding = User(db, "dicoding")  dicoding.set\_aktif() # aktifkan user baru  self.assertTrue(dicoding.aktif)  putus\_koneksi\_db(db)    if \_\_name\_\_ == "\_\_main\_\_":  # Test Runner  unittest.main() |

Sama seperti sebelumnya, kita akan membuat sebuah class TestUser yang merupakan turunan dari class unittest.TestCase, kemudian menulis dua metode untuk pengujian kali ini.

Jika Anda perhatikan kembali semua kode di atas, kita melakukan beberapa tindakan yang berulang ketika kita menjalankan fungsi koneksi ke basis data dan membuat User dicoding setiap kali proses tes. Hal ini karena setiap tes itu dioperasikan secara terpisah sehingga perlu melakukan tindakan yang disebutkan secara berulang. Tindakan ini juga dianggap bukan praktik yang baik karena memakan lebih banyak memori apalagi jika program yang kita uji berukuran besar.

Lantas, kita bisa memperbaikinya dengan menerapkan konsep *test fixture*. Konsep ini merepresentasikan persiapan yang dibutuhkan untuk melakukan satu pengujian atau lebih serta proses pembersihannya (cleanup).

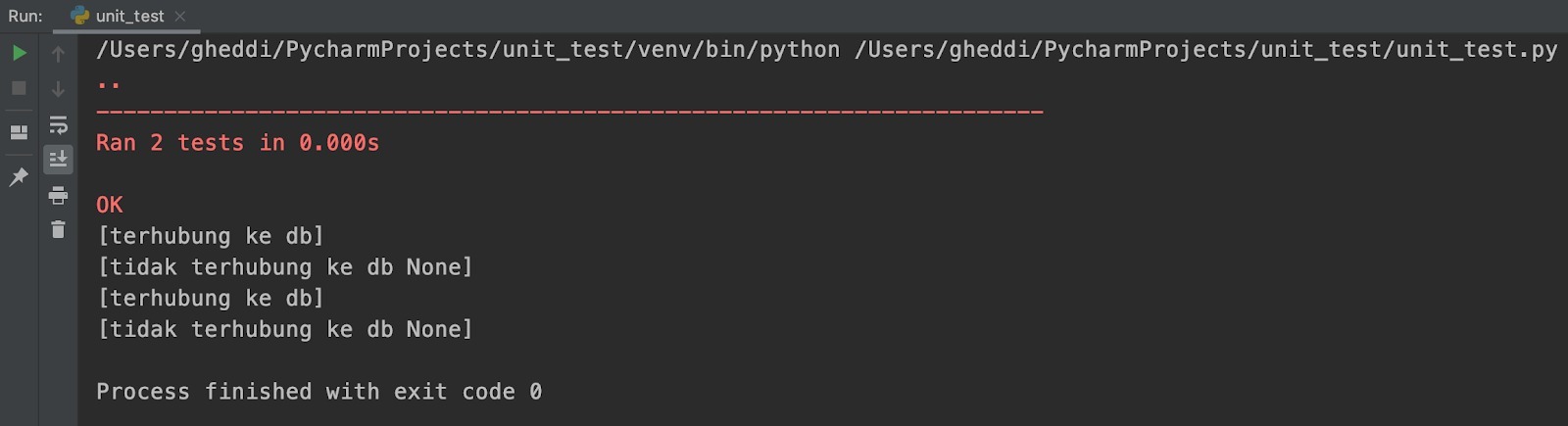
Kita akan menggunakan metode bawaan dari class TestCase, yakni metode setUp() dan tearDown().

* Metode setUp(), sesuai namanya, bertujuan untuk mempersiapkan pengujian. Metode ini akan dipanggil untuk menyiapkan tes sehingga pemanggilannya akan dilakukan tiap sebelum metode tes dilaksanakan.
* Metode tearDown() akan dipanggil setiap metode tes selesai dilaksanakan dan bertindak untuk membersihkan, meskipun terjadi kesalahan (*exception*) pada proses tes.

Kode sebelumnya akan kita ubah dengan implementasi kedua metode setUp() dan tearDown(). Kita cukup melakukan perubahan pada class TestUser saja seperti di bawah.

|  |
| --- |
| class TestUser(unittest.TestCase):  # Test Fixture  def setUp(self):  self.db = koneksi\_ke\_db()  self.dicoding = User(self.db, "dicoding")    def tearDown(self):  putus\_koneksi\_db(self.db)    # Test Case 1  def test\_user\_default\_not\_active(self):  self.assertFalse(self.dicoding.aktif) # tidak aktif secara default  # Test Case 2  def test\_user\_is\_active(self):  self.dicoding.set\_aktif() # aktifkan user baru  self.assertTrue(self.dicoding.aktif) |

Kemudian jalankan kembali program Anda. Keluarannya seperti berikut.



Terlihat bahwa setiap kali melakukan pengujian, metode setUp() dipanggil. Begitu juga setelah selesai pengujian, metode tearDown() dipanggil.

Dengan kemampuan pengujian ini, aplikasi yang Anda buat jadi lebih teruji atau biasa disebut dengan istilah lebih tahan banting (*robust*).