Waktunya Naik Level: Menjadi Master Fungsi!

Selamat datang di pertemuan keenam! Di pelajaran sebelumnya, kita telah mempelajari fondasi dari fungsi: cara mendefinisikan, memanggil, menggunakan parameter, dan mengembalikan nilai dengan `return`. Kita juga sudah paham tentang variabel lokal dan bagaimana fungsi menjaga "ruang kerja pribadinya".

Hari ini, kita akan membawa kemampuan fungsi kita ke level selanjutnya. Kita akan belajar bagaimana membuat fungsi yang lebih fleksibel, lebih mudah dibaca, lebih informatif, dan bagaimana ia bisa berinteraksi (dengan hati-hati) dengan dunia di luarnya.

Review Singkat: Apa yang Sudah Kita Tahu

Mari kita ingat kembali anatomi dasar sebuah fungsi yang mengembalikan nilai (*fruitful function*):

```
# Definisi Fungsi
# 'parameter1' dan 'parameter2' adalah placeholder (variabel lokal)
def nama_fungsi(parameter1, parameter2):
    # 1. Menerima input melalui parameter.
    # 2. Melakukan serangkaian proses di 'body' fungsi.
    hasil_proses = parameter1 + parameter2
    # 3. Mengembalikan output menggunakan 'return'.
    return hasil_proses

# Pemanggilan Fungsi
# 1. Memberikan nilai nyata (argumen) ke fungsi.
# 2. Menangkap nilai kembaliannya dalam sebuah variabel.
hasil_akhir = nama_fungsi(10, 5) # 10 dan 5 adalah argumen
print(hasil_akhir) # Output: 15
```

Ini adalah pola yang akan terus kita gunakan. Sekarang, mari kita tingkatkan fleksibilitas dan kekuatannya.

Fleksibilitas Tingkat 1: Argumen Default (Default Arguments)

Terkadang, sebuah parameter dalam fungsi kita memiliki nilai yang "standar" atau paling sering digunakan. Misalnya, saat menghitung total belanjaan, pajak standarnya mungkin 11%, tapi kadang bisa berbeda (misal, untuk produk makanan pajaknya 5%).

Akan merepotkan jika kita harus selalu memasukkan nilai `11` setiap kali memanggil fungsi untuk kasus standar. Di sinilah **argumen default** berguna. Kita bisa memberikan nilai *default* pada parameter langsung di dalam definisi fungsi menggunakan operator assignment `=`.

```
# 'persen_pajak' memiliki nilai default 11
def hitung_total_belanja(subtotal, persen_pajak=11):
    pajak = subtotal * (persen_pajak / 100)
    total = subtotal + pajak
    return total
```

```
# Kasus 1: Memanggil fungsi TANPA memberikan argumen untuk persen_pajak.
# Python akan otomatis menggunakan nilai default yang sudah kita tentukan
(11).
belanja_standar = hitung_total_belanja(100000)
print(f"Total belanja standar (pajak 11%): Rp {belanja_standar}")

# Kasus 2: Memanggil fungsi DENGAN memberikan argumen untuk persen_pajak.
# Nilai default akan DIABAIKAN dan diganti dengan nilai yang kita berikan.
belanja_makanan = hitung_total_belanja(100000, 5) # Pajak khusus 5%
print(f"Total belanja makanan (pajak 5%): Rp {belanja_makanan}")
```

Output:

```
Total belanja standar (pajak 11%): Rp 111000.0
Total belanja makanan (pajak 5%): Rp 105000.0
```

Aturan Emas Argumen Default: Parameter dengan nilai default **HARUS** diletakkan **SETELAH** semua parameter yang tidak punya nilai default.

```
# KODE INI AKAN MENYEBABKAN SYNTAXERROR!
def fungsi_salah(nama="Budi", umur):
    print(f"{nama} berumur {umur} tahun")

# SyntaxError: non-default argument follows default argument
```

Mengapa? Karena Python akan bingung saat memproses argumen. Jika kamu memanggil `fungsi_salah(20)`, apakah `20` ini untuk `nama` atau `umur`? Untuk menghindari kebingungan ini, Python memberlakukan aturan ketat tersebut.

```
# KODE YANG BENAR
def fungsi_benar(umur, nama="Budi", kota="Jakarta"):
    print(f"{nama} berumur {umur} tahun dari kota {kota}")

fungsi_benar(25) # umur=25, nama & kota akan menggunakan default
fungsi_benar(30, nama="Ani") # umur=30, nama diubah, kota default
fungsi_benar(22, "Charlie", "Bandung") # semua diisi secara posisional
```

Fleksibilitas Tingkat 2: Argumen Kata Kunci (Keyword Arguments)

Perhatikan fungsi berikut:

```
def data_siswa(nama, umur, kelas, sekolah):
    print(f"Nama: {nama}, Umur: {umur}, Kelas: {kelas}, Sekolah:
{sekolah}")
```

Saat memanggilnya, kita harus hafal urutan parameternya. Argumen yang diberikan berdasarkan urutan/posisi ini disebut **positional arguments**.

```
# Ini disebut 'positional arguments' karena posisinya penting
data_siswa("Charlie", 16, "11 IPA 2", "SMA Harapan Bangsa")
```

Bagaimana jika kita lupa urutannya? Atau jika fungsinya punya 10 parameter? Tentu akan sangat merepotkan dan rawan kesalahan.

Untuk mengatasi ini, kita bisa menggunakan **argumen kata kunci (keyword arguments)**. Kita secara eksplisit menyebutkan nama parameter mana yang ingin kita isi saat memanggil fungsi.

```
# Dengan keyword arguments, urutan tidak lagi penting!
data_siswa(sekolah="SMA Cita-Cita", nama="Dani", kelas="12 IPS 1",
umur=18)
```

Outputnya akan tetap benar dan terurut sesuai definisi fungsi, karena Python mencocokkan argumen berdasarkan namanya, bukan posisinya.

Keuntungan Keyword Arguments:

1. **Meningkatkan Keterbacaan:** Kode menjadi sangat jelas dan mendokumentasikan dirinya sendiri. Kita tahu persis nilai `18` itu untuk parameter `umur`.
2. **Menghindari Kesalahan Urutan:** Tidak perlu lagi pusing menghafal urutan parameter.

Kita juga bisa mencampur *positional* dan *keyword arguments*. **Aturan Emas Kedua:** *Positional arguments* **HARUS** selalu ditulis **SEBELUM** *keyword arguments*.

```
# BENAR: Positional dulu, baru keyword
data_siswa("Eko", 17, sekolah="SMA Maju Jaya", kelas="11 IPS 3")

# SALAH: Keyword tidak boleh diikuti positional
# data_siswa(nama="Fani", 19, kelas="12 Bahasa", sekolah="SMA Pelita") #
Akan menyebabkan SyntaxError
```

Kekuatan Super `return`: Mengembalikan Banyak Nilai

Sejauh ini, fungsi kita hanya mengembalikan satu nilai. Tapi bagaimana jika sebuah proses menghasilkan beberapa output sekaligus? Misalnya, sebuah fungsi statistik yang menghitung nilai rata-rata, tertinggi, dan terendah dari sebuah daftar.

Di Python, ini sangat mudah. Cukup pisahkan nilai-nilai yang ingin dikembalikan dengan koma di dalam `return` statement.

```python
def kalkulasi_sederhana(a, b):
    tambah = a + b
    kurang = a - b
    kali = a * b
    # Guard clause untuk menghindari pembagian dengan nol
    if b != 0:
        bagi = a / b
    else:
        bagi = "Tidak bisa dibagi dengan nol"

    # Kembalikan SEMUA hasil perhitungan
    return tambah, kurang, kali, bagi

# Memanggil fungsi dan menangkap hasilnya
hasil_komplit = kalkulasi_sederhana(10, 2)
print("Hasil yang dikembalikan:", hasil_komplit)
print("Tipe data hasil:", type(hasil_komplit))
```

Output:

```
Hasil yang dikembalikan: (12, 8, 20, 5.0)
Tipe data hasil: <class 'tuple'>
```

Ternyata, saat kita mengembalikan banyak nilai, Python secara diam-diam membungkusnya dalam satu tipe data bernama **tuple** (sebuah collection yang *immutable*).

Yang lebih keren lagi, kita bisa langsung "membongkar" (*unpack*) tuple ini ke dalam beberapa variabel terpisah.

```python
# Unpacking hasil return ke dalam variabel-variabel terpisah
hasil_tambah, hasil_kurang, hasil_kali, hasil_bagi =
kalkulasi_sederhana(10, 2)

print(f"Hasil Penjumlahan: {hasil_tambah}")
print(f"Hasil Pengurangan: {hasil_kurang}")
print(f"Hasil Perkalian: {hasil_kali}")
print(f"Hasil Pembagian: {hasil_bagi}")
```

Ini adalah teknik yang sangat umum dan kuat di Python untuk membuat fungsi yang informatif.

Praktik Profesional: Mendokumentasikan Fungsi dengan Docstrings

Kode yang baik adalah kode yang bisa dimengerti oleh orang lain dan oleh dirimu sendiri di masa depan. Selain menggunakan nama variabel dan fungsi yang deskriptif, cara terbaik untuk mendokumentasikan fungsi adalah dengan **Docstrings**.

Docstring adalah sebuah string multi-baris (menggunakan `"""..."""`) yang diletakkan sebagai baris **pertama** di dalam sebuah fungsi. Tujuannya adalah untuk menjelaskan apa yang dilakukan fungsi tersebut, input apa yang dibutuhkannya, dan output apa yang dihasilkannya.

```
def hitung_imt(berat_kg, tinggi_m):
    """Menghitung Indeks Massa Tubuh (IMT) dan memberikan kategorinya.

    Fungsi ini menerima berat badan dalam kilogram dan tinggi badan dalam
meter,
    lalu mengembalikan nilai IMT sebagai float dan kategori berat badan
    sebagai string.

    Args:
        berat_kg (float or int): Berat badan dalam kilogram.
        tinggi_m (float or int): Tinggi badan dalam meter.

    Returns:
        tuple: Sebuah tuple berisi dua nilai:
               - imt (float): Nilai Indeks Massa Tubuh (IMT).
               - kategori (str): Kategori berat badan (e.g., "Normal",
"Obesitas").
               Mengembalikan (None, "Error") jika input tidak valid.
    """
    # Guard Clause: Cek kondisi error di awal
    if tinggi_m <= 0 or berat_kg <= 0:
        return None, "Error: Tinggi dan berat harus positif."

    imt = berat_kg / (tinggi_m ** 2)

    if imt < 18.5:
        kategori = "Kurang berat badan"
    elif imt < 25:
        kategori = "Berat badan normal"
    elif imt < 30:
        kategori = "Kelebihan berat badan"
    else:
        kategori = "Obesitas"

    return imt, kategori
```

Apa keajaiban Docstrings? IDE modern seperti VS Code bisa membacanya dan menampilkannya sebagai *tooltip* saat kamu menggunakan fungsi itu. Selain itu, kamu bisa memanggil fungsi `help()` untuk melihat dokumentasi tersebut secara langsung!

```
help(hitung_imt)
```

Outputnya akan menampilkan docstring yang sudah kita tulis dengan rapi. Ini adalah praktik yang sangat penting dalam dunia kerja.

Fungsi Anonim: `lambda` Expressions

Terkadang kita butuh fungsi yang sangat sederhana, yang hanya melakukan satu hal, dan kita hanya akan menggunakannya sekali. Membuat fungsi lengkap dengan `def` terasa berlebihan.

Untuk kasus seperti ini, Python menyediakan **lambda expression** untuk membuat fungsi kecil tanpa nama (anonim).

Strukturnya: `lambda parameter: ekspresi`

```
# Cara biasa dengan def
def tambah_lima(x):
    return x + 5

# Cara singkat dengan lambda
tambah_lima_lambda = lambda x: x + 5

# Keduanya dipanggil dengan cara yang sama
print("Hasil dari def:", tambah_lima(10))
print("Hasil dari lambda:", tambah_lima_lambda(10))
```

Keduanya menghasilkan `15`. Lambda sangat berguna saat digunakan bersama fungsi lain seperti `map()` atau `filter()`, yang akan kita pelajari nanti. Untuk sekarang, cukup ketahui bahwa ini adalah cara cepat untuk membuat fungsi satu baris.

Latihan untuk Menguji Pemahaman

Waktunya mengasah kemampuan fungsi tingkat lanjut!

Latihan 1: Fungsi dengan Argumen Default Buatlah sebuah fungsi bernama `buat_email` yang menerima dua parameter: `nama_pengguna` dan `domain` dengan nilai default `"coding.com"`. Fungsi ini harus mengembalikan alamat email lengkap dalam format lowercase.

- `buat_email("Budi")` harus mengembalikan `"budi@coding.com"`
- `buat_email("Ani", "belajar.id")` harus mengembalikan `"ani@belajar.id"`

Latihan 2: Memanggil dengan Keyword Arguments Diberikan fungsi berikut: `def kirim_paket(barang, tujuan, berat_kg, asuransi=False, express=False):`. Panggil fungsi tersebut untuk mengirim "Buku" ke "Bandung" dengan berat 2 kg. Gunakan *keyword arguments* untuk secara eksplisit menyatakan bahwa layanan `express` harus `True`, sementara `asuransi` tetap menggunakan nilai default-nya.

Latihan 3: Mengembalikan Banyak Nilai Buatlah sebuah fungsi bernama `analisis_daftar` yang menerima satu parameter berupa daftar angka (misal: `[10, 20, 30, 40, 50]`). Fungsi ini harus mengembalikan TIGA nilai:

1. Jumlah total semua angka dalam daftar.

2. Jumlah elemen dalam daftar.
3. Nilai rata-rata. *Petunjuk: Gunakan fungsi bawaan `sum()` dan `len()`*. Panggil fungsi tersebut dan "unpack" hasilnya ke dalam tiga variabel terpisah, lalu cetak.

Latihan 4: Dokumentasi Profesional Ambil fungsi `analisis_daftar` dari Latihan 3 dan tambahkan **docstring** yang lengkap. Docstring harus menjelaskan:

1. Apa tujuan fungsi tersebut.
2. Apa parameter yang diterimanya (`Args`).
3. Apa yang dikembalikannya (`Returns`). Setelah itu, panggil `help(analisis_daftar)` untuk memeriksa apakah docstring Anda muncul dengan benar.

Latihan 5: Konversi ke Lambda Diberikan fungsi `def` berikut:

```
def get_luas_lingkaran(radius):
    return 3.14159 * (radius ** 2)
```

Tulis ulang fungsi di atas sebagai sebuah *lambda expression* yang disimpan dalam variabel bernama `luas_lingkaran_lambda`. Uji dengan memanggilnya.