!DOCTYPE html>

Penjelasan Kode Program Kasir Python

Lengkap dengan LOGIKA dan ALASAN Penggunaan

Daftar Isi

---

1. Struktur Data Produk

```
produk = {
   1: {"nama": "Nasi Goreng", "harga": 15000},
   2: {"nama": "Mie Ayam", "harga": 12000},
   ...
}
```

Apa yang dilakukan:

- `produk` adalah DICTIONARY (kamus) yang menyimpan data produk
- KEY (kunci): angka 1, 2, 3, dst = kode produk
- VALUE (nilai): dictionary lagi berisi `nama` dan `harga`

MENGAPA pakai Dictionary?

Masalah: Kita perlu menyimpan data produk (nama + harga) dan bisa diakses dengan kode.

Alternatif lain yang TIDAK efisien:

```
# Cara buruk: pakai banyak variabel terpisah
nama1 = "Nasi Goreng"
harga1 = 15000
nama2 = "Mie Ayam"
harga2 = 12000
```

```
# Susah kalau mau loop, susah kalau mau tambah produk baru
```

Kenapa Dictionary lebih baik: 1. **Akses cepat** - Cukup tulis `produk[1]` untuk dapat semua info 2. **Mudah di-loop** - Bisa pakai `for` untuk tampilkan semua produk 3. **Mudah ditambah** - Tinggal tambah baris baru 4. **Data terorganisir** - Nama dan harga terhubung dalam satu tempat

MENGAPA pakai Nested Dictionary (dictionary dalam dictionary)?

Masalah: Setiap produk punya 2 info: nama DAN harga

Alternatif lain:

```
# Cara 1: Pakai 2 dictionary terpisah (RIBET)
nama_produk = {1: "Nasi Goreng", 2: "Mie Ayam"}
harga_produk = {1: 15000, 2: 12000}
# Harus akses 2 tempat berbeda, rawan tidak sinkron

# Cara 2: Pakai list (MEMBINGUNGKAN)
produk = {1: ["Nasi Goreng", 15000]}
# produk[1][0] = nama, produk[1][1] = harga
# Harus ingat urutan, tidak jelas mana nama mana harga
```

Kenapa Nested Dictionary lebih baik:

```
produk[1]["nama"]   # Jelas ini mengambil NAMA
produk[1]["harga"]  # Jelas ini mengambil HARGA
# Kode lebih mudah dibaca dan dipahami
```

Cara Akses:

```
produk[1]              # {"nama": "Nasi Goreng", "harga": 15000}
produk[1]["nama"]      # "Nasi Goreng"
produk[1]["harga"]     # 15000
```

---

2. Fungsi tampilkan_menu()

```
def tampilkan_menu():
    print()
    print("=" * 40)
    print("          DAFTAR MENU")
    print("=" * 40)
    print(f"{'Kode':<6} {'Nama Produk':<20} {'Harga':>10}")
    print("-" * 40)
    for kode, item in produk.items():
        harga_str = f"Rp {item['harga']:>7,}".replace(",", ".")
        print(f"{kode:<6} {item['nama']:<20} {harga_str}")
    print("=" * 40)
```

MENGAPA dibuat sebagai Fungsi?

Masalah: Kode untuk tampilkan menu mungkin dipanggil berkali-kali.

Tanpa fungsi (BURUK):

```
# Harus copy-paste kode yang sama setiap kali mau tampilkan menu
print("=" * 40)
print("          DAFTAR MENU")
# ... dst (banyak baris)

# Nanti di tempat lain copy-paste lagi
```

```
print("=" * 40)
print("          DAFTAR MENU")
# ... dst
```

Dengan fungsi (BAIK):

```
def tampilkan_menu():
   # Semua kode di sini

# Panggil kapan saja dengan 1 baris
tampilkan_menu()
tampilkan_menu()  # Mudah dipanggil berulang
```

Keuntungan pakai Fungsi: 1. **Tidak perlu copy-paste** - Tulis sekali, pakai berkali-kali 2. **Mudah diubah** - Ubah di satu tempat, berubah di semua 3. **Kode lebih rapi** - Fungsi main() jadi lebih pendek dan mudah dibaca

MENGAPA pakai `print("=" * 40)`?

Masalah: Mau cetak garis pembatas yang panjangnya konsisten.

Cara manual (RIBET):

```
print("========================================")
# Harus hitung manual, rawan salah jumlah
```

Cara pintar:

```
print("=" * 40)  # Python otomatis ulang "=" sebanyak 40 kali
# Mudah diubah, tinggal ganti angkanya
```

MENGAPA pakai `for kode, item in produk.items()`?

Masalah: Mau tampilkan SEMUA produk tanpa tulis satu per satu.

Tanpa loop (BURUK):

```
print(f"1   Nasi Goreng   Rp 15.000")
print(f"2   Mie Ayam      Rp 12.000")
print(f"3   Bakso         Rp 13.000")
# Harus tulis manual setiap produk, kalau tambah produk harus tambah baris
```

Dengan loop (BAIK):

```
for kode, item in produk.items():
   print(f"{kode}   {item['nama']}   Rp {item['harga']}")
# Otomatis tampilkan semua, mau 10 atau 100 produk tetap 2 baris kode
```

Penjelasan `produk.items()`:

```
produk = {1: {"nama": "Nasi Goreng", "harga": 15000}, 2: {...}}

# .items() menghasilkan pasangan (key, value)
# Loop pertama: kode=1, item={"nama": "Nasi Goreng", "harga": 15000}
# Loop kedua:   kode=2, item={"nama": "Mie Ayam", "harga": 12000}
```

MENGAPA pakai `.replace(",", ".")`?

Masalah: Python pakai format angka internasional (koma untuk ribuan), tapi Indonesia pakai titik.

```
harga = 15000
print(f"{harga:,}")      # Output: 15,000 (format internasional)
print(f"{harga:,}".replace(",", "."))  # Output: 15.000 (format Indonesia)
```

MENGAPA pakai Format String seperti `{kode:<6}`?

Masalah: Mau tampilan tabel yang rapi dan sejajar.

Tanpa format (BERANTAKAN):

```
1 Nasi Goreng Rp 15.000
2 Mie Ayam Rp 12.000
10 Air Mineral Rp 4.000
```

Dengan format (RAPI):

```
1      Nasi Goreng          Rp  15.000
2      Mie Ayam             Rp  12.000
10     Air Mineral          Rp   4.000
```

Arti format: - `{kode:<6}` = rata kiri, minimal 6 karakter (tambah spasi di kanan) - `{nama:<20}` = rata kiri, minimal 20 karakter - `{harga:>10}` = rata kanan, minimal 10 karakter (tambah spasi di kiri)

---

3. Fungsi hitung_total()

```
def hitung_total(keranjang):
   total = 0
   for kode, jumlah in keranjang.items():
       total += produk[kode]["harga"] * jumlah
   return total
```

MENGAPA dibuat Fungsi terpisah?

Masalah: Perhitungan total dibutuhkan di beberapa tempat: 1. Saat tampilkan ringkasan belanja 2. Saat cek apakah uang cukup 3. Saat cetak struk

Tanpa fungsi (BURUK - kode berulang):

```
# Di tempat 1
total = 0
for kode, jumlah in keranjang.items():
   total += produk[kode]["harga"] * jumlah

# Di tempat 2 (copy-paste lagi)
total = 0
for kode, jumlah in keranjang.items():
   total += produk[kode]["harga"] * jumlah
```

Dengan fungsi (BAIK):

```
total = hitung_total(keranjang)  # Panggil kapan saja
```

MENGAPA pakai Parameter `keranjang`?

Masalah: Fungsi perlu tahu keranjang mana yang mau dihitung.

Penjelasan Parameter:

```
def hitung_total(keranjang):   # keranjang = data yang dikirim dari luar
   # Fungsi bisa akses data keranjang di sini

# Cara pakai:
keranjang_saya = {1: 2, 3: 1}
hasil = hitung_total(keranjang_saya)  # Kirim data ke fungsi
```

MENGAPA pakai `total = 0` di awal?

Logika: Kita mau MENJUMLAHKAN semua harga. Untuk menjumlah, perlu tempat penampung yang dimulai dari 0.

Analogi: Seperti kalkulator, sebelum mulai hitung harus di-reset ke 0 dulu.

```
total = 0                    # Mulai dari 0
total += 30000               # 0 + 30000 = 30000
total += 13000               # 30000 + 13000 = 43000
# Hasil akhir: 43000
```

MENGAPA pakai `+=`?

Penjelasan: `+=` adalah singkatan untuk "tambahkan ke nilai sebelumnya"

```
total = total + 10000    # Cara panjang
total += 10000           # Cara singkat (sama artinya)
```

MENGAPA pakai `return`?

Masalah: Fungsi sudah hitung total, tapi hasilnya perlu dikirim keluar agar bisa dipakai.

Tanpa return (TIDAK BERGUNA):

```
def hitung_total(keranjang):
   total = 0
   for kode, jumlah in keranjang.items():
       total += produk[kode]["harga"] * jumlah
   # Tidak ada return, hasil total hilang!

hasil = hitung_total(keranjang)  # hasil = None (kosong)
```

Dengan return (BENAR):

```
def hitung_total(keranjang):
   total = 0
   for kode, jumlah in keranjang.items():
       total += produk[kode]["harga"] * jumlah
   return total  # Kirim hasil keluar

hasil = hitung_total(keranjang)  # hasil = 43000
```

Contoh Perhitungan Step by Step:

```
keranjang = {1: 2, 3: 1}  # 2 Nasi Goreng, 1 Bakso

Awal: total = 0

Loop 1: kode=1, jumlah=2
 harga = produk[1]["harga"] = 15000
 subtotal = 15000 * 2 = 30000
 total = 0 + 30000 = 30000

Loop 2: kode=3, jumlah=1
 harga = produk[3]["harga"] = 13000
 subtotal = 13000 * 1 = 13000
```

```
 total = 30000 + 13000 = 43000

return 43000
```

---

4. Fungsi cetak_struk()

```
def cetak_struk(keranjang, bayar):
   total = hitung_total(keranjang)
   kembalian = bayar - total
   ...
```

MENGAPA ada 2 Parameter (keranjang, bayar)?

Logika: Untuk cetak struk, kita butuh 2 informasi: 1. `keranjang` = apa saja yang dibeli (untuk hitung total dan cetak daftar) 2. `bayar` = berapa uang yang dibayar (untuk hitung kembalian)

```
def cetak_struk(keranjang, bayar):
   total = hitung_total(keranjang)  # Pakai keranjang untuk hitung total
   kembalian = bayar - total         # Pakai bayar untuk hitung kembalian
```

MENGAPA memanggil fungsi `hitung_total()` lagi?

Prinsip: Jangan ulangi kode yang sama! (Don't Repeat Yourself / DRY)

Cara buruk:

```
def cetak_struk(keranjang, bayar):
   # Hitung total lagi dari awal (copy-paste kode)
   total = 0
   for kode, jumlah in keranjang.items():
       total += produk[kode]["harga"] * jumlah
```

Cara baik:

```
def cetak_struk(keranjang, bayar):
   total = hitung_total(keranjang)  # Manfaatkan fungsi yang sudah ada!
```

MENGAPA hitung `kembalian = bayar - total`?

Logika kehidupan nyata: - Pelanggan bayar Rp 50.000 - Total belanja Rp 43.000 - Kembalian = 50.000 - 43.000 = Rp 7.000

```
bayar = 50000
total = 43000
kembalian = bayar - total  # 50000 - 43000 = 7000
```

Contoh Output Struk:

```
========================================
         STRUK PEMBELIAN
========================================
Nasi Goreng     x2     Rp  30.000
Es Teh          x3     Rp  15.000
----------------------------------------
Total:          Rp     45.000
Bayar:          Rp     50.000
Kembalian:      Rp      5.000
========================================
        Terima Kasih!
========================================
```

5. Fungsi main() - Alur Utama Program

Bagian 1: Inisialisasi Keranjang Kosong

```
def main():
   keranjang = {}
```

MENGAPA `keranjang = {}`?

Logika: Saat program mulai, pelanggan belum beli apa-apa. Keranjang harus kosong dulu.

Analogi: Seperti ke supermarket, ambil troli kosong dulu baru mulai belanja.

```
keranjang = {}           # Awal: kosong
keranjang[1] = 2         # Tambah 2 Nasi Goreng
keranjang[3] = 1         # Tambah 1 Bakso
# Sekarang: {1: 2, 3: 1}
```

Bagian 2: Input Kode Produk

```
pesanan = input("Kode produk: ").strip()
```

MENGAPA pakai `.strip()`?

Masalah: User mungkin tidak sengaja ketik spasi di awal/akhir.

```
pesanan = input()           # User ketik: "  1, 2, 3  "
print(pesanan)              # "  1, 2, 3  " (ada spasi)
print(pesanan.strip())      # "1, 2, 3" (spasi dihapus)
```

Bagian 3: Memproses Kode dengan `.split(",")`

```
items = pesanan.split(",")
```

MENGAPA pakai `.split(",")`?

Masalah: User input "1, 3, 5" dalam satu baris. Kita perlu pisahkan jadi bagian-bagian.

```
pesanan = "1, 3, 5"
items = pesanan.split(",")   # Pecah berdasarkan koma
print(items)                 # ["1", " 3", " 5"]
```

Penjelasan visual:

```
"1, 3, 5"
  ↓ split(",")
["1", " 3", " 5"]
 ↓     ↓     ↓
item  item  item
```

Bagian 4: Konversi String ke Integer dengan Try-Except

```
for item in items:
   try:
       kode = int(item.strip())
   except ValueError:
       print("Input tidak valid")
```

MENGAPA perlu `int()`?

Masalah: Input dari user SELALU berupa string (teks), bukan angka.

```
pesanan = input("Masukkan angka: ")  # User ketik: 5
print(type(pesanan))                 # <class 'str'> (STRING, bukan angka!)
print(pesanan + 1)                   # ERROR! Tidak bisa "5" + 1

kode = int(pesanan)                  # Konversi ke integer
print(type(kode))                    # <class 'int'> (ANGKA!)
print(kode + 1)                      # 6 (bisa dihitung)
```

MENGAPA pakai Try-Except?

Masalah: Bagaimana kalau user ketik huruf bukan angka?

```
# TANPA try-except (PROGRAM CRASH):
kode = int("abc")  # ERROR! ValueError: invalid literal

# DENGAN try-except (PROGRAM AMAN):
try:
   kode = int("abc")
except ValueError:
   print("Bukan angka!")  # Program tetap jalan
```

Analogi: Seperti jaring pengaman. Kalau ada kesalahan, jaring menangkap supaya tidak jatuh.

Bagian 5: Validasi Input dengan While True

```
for kode in kode_list:
   while True:
       try:
           jumlah = int(input(f"Jumlah {produk[kode]['nama']}: "))
           if jumlah <= 0:
               print("Jumlah harus lebih dari 0!")
               continue
           break
       except ValueError:
           print("Masukkan angka yang valid!")
```

MENGAPA pakai `while True`?

Logika: Kita mau PAKSA user memasukkan input yang benar. Kalau salah, minta lagi sampai benar.

```
Jumlah Nasi Goreng: abc     ← Salah (bukan angka)
Masukkan angka yang valid!
Jumlah Nasi Goreng: -5      ← Salah (negatif)
Jumlah harus lebih dari 0!
Jumlah Nasi Goreng: 2       ← Benar! Keluar dari loop
```

MENGAPA pakai `continue` dan `break`?

- `continue` = "Salah! Ulangi dari awal while loop"
- `break` = "Benar! Keluar dari while loop, lanjut ke kode berikutnya"

```
while True:                    # Loop terus-menerus
   jumlah = int(input())
   if jumlah <= 0:
       continue               # Kembali ke while True (minta input lagi)
   break                      # Keluar dari loop (input sudah benar)
```

Bagian 6: Menyimpan ke Keranjang

```
if kode in keranjang:
   keranjang[kode] += jumlah
else:
   keranjang[kode] = jumlah
```

MENGAPA cek `if kode in keranjang`?

Masalah: Bagaimana kalau user pesan produk yang sama 2 kali?

Contoh kasus:

```
Input pertama: 1 (Nasi Goreng), jumlah 2
Input kedua: 1 (Nasi Goreng lagi), jumlah 3
```

Tanpa pengecekan (SALAH):

```
keranjang[1] = 2    # {1: 2}
keranjang[1] = 3    # {1: 3} ← Nilai 2 HILANG, diganti 3!
```

Dengan pengecekan (BENAR):

```
# Input pertama
keranjang[1] = 2    # {1: 2}

# Input kedua
if 1 in keranjang:           # True, kode 1 sudah ada
   keranjang[1] += 3        # 2 + 3 = 5
# Hasil: {1: 5} ← Total 5 Nasi Goreng
```

Bagian 7: Entry Point Program

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

MENGAPA ada baris ini?

Penjelasan sederhana: Baris ini memastikan `main()` hanya jalan kalau file dieksekusi langsung.

```
# Kalau jalankan: python main.py
# __name__ akan bernilai "__main__"
# Jadi main() akan dipanggil

# Kalau di-import dari file lain: import main
# __name__ akan bernilai "main" (nama file)
# Jadi main() TIDAK dipanggil otomatis
```

Untuk pemula: Anggap saja ini "tombol start" program. Selalu tulis di akhir file.

---

6. Ringkasan Konsep Python yang Dipakai

A. Dictionary - Menyimpan Data Berpasangan

```
# KAPAN PAKAI: Saat butuh data yang bisa diakses dengan "kunci"
produk = {1: "Nasi Goreng", 2: "Mie Ayam"}
print(produk[1])   # "Nasi Goreng"
```

B. List - Menyimpan Kumpulan Data Berurutan

```
# KAPAN PAKAI: Saat butuh kumpulan data yang berurutan
kode_list = [1, 3, 5]
for kode in kode_list:
   print(kode)
```

C. For Loop - Mengulang untuk Setiap Item

```
# KAPAN PAKAI: Saat tahu berapa kali mau mengulang
for i in range(5):    # Ulang 5 kali (0,1,2,3,4)
   print(i)
```

D. While Loop - Mengulang Sampai Kondisi Terpenuhi

```
# KAPAN PAKAI: Saat tidak tahu berapa kali mengulang
while True:           # Ulang terus sampai break
   jawab = input()
   if jawab == "y":
       break
```

E. Try-Except - Menangani Error

```
# KAPAN PAKAI: Saat ada kemungkinan error dari input user
try:
   angka = int(input())   # Mungkin error kalau bukan angka
except ValueError:
   print("Bukan angka!")  # Tangkap error
```

F. Function - Kode yang Bisa Dipanggil Ulang

```
# KAPAN PAKAI: Saat ada kode yang dipakai berkali-kali
def sapa(nama):
   print(f"Halo {nama}!")

sapa("Budi")   # "Halo Budi!"
sapa("Ani")    # "Halo Ani!"
```

G. Return - Mengembalikan Nilai dari Fungsi

```
# KAPAN PAKAI: Saat fungsi perlu kirim hasil keluar
def tambah(a, b):
   return a + b

hasil = tambah(5, 3)   # hasil = 8
```

H. F-String - Format Teks dengan Variabel

```
# KAPAN PAKAI: Saat mau gabung teks dengan variabel
nama = "Budi"
umur = 20
print(f"Nama: {nama}, Umur: {umur}")
```

Pola Pikir Programmer

1. Pecah Masalah Jadi Bagian Kecil

```
Masalah: Buat program kasir

Pecah jadi:
■■■ Tampilkan menu
■■■ Input pesanan
■■■ Hitung total
■■■ Terima pembayaran
■■■ Cetak struk
```

2. Hindari Pengulangan Kode (DRY)

```
BURUK: Copy-paste kode yang sama di banyak tempat
BAIK: Buat fungsi, panggil berkali-kali
```

3. Validasi Input User

```
BURUK: Asumsikan user selalu input dengan benar
BAIK: Selalu cek dan tangani input yang salah
```

4. Beri Nama yang Jelas

```
BURUK: x = 15000, y = 2, z = x * y
BAIK: harga = 15000, jumlah = 2, subtotal = harga * jumlah
```

Latihan untuk Pemula

Level 1 - Mudah: 1. Tambahkan 3 menu baru ke dalam data produk 2. Ubah format harga menjadi "Rp. 15.000,-"

Level 2 - Sedang: 3. Buat diskon 10% jika total belanja di atas Rp 50.000 4. Tambahkan tanggal dan waktu di struk

Level 3 - Sulit: 5. Buat fitur untuk menghapus item dari keranjang 6. Simpan struk ke file teks (.txt) 7. Buat laporan penjualan harian

---

Tips Debugging (Mencari Error)

1. **Baca pesan error** - Python memberitahu baris mana yang error
2. **Pakai print()** - Cetak nilai variabel untuk lihat isinya
3. **Cek tipe data** - Pakai `print(type(variabel))`
4. **Coba di bagian kecil** - Test fungsi satu per satu

```
# Contoh debugging
print(f"DEBUG: keranjang = {keranjang}")
print(f"DEBUG: type = {type(kode)}")
```

Selamat belajar!