Slide 6: Funzioni e Procedure 🔧

Cos'è una funzione?

Una **funzione** è un blocco di codice riutilizzabile che:

- Esegue un compito specifico
- Può ricevere dati in input (parametri)
- Può restituire un risultato (return)
- Rende il codice più organizzato e manutenibile

Analogia: Come una ricetta in cucina - definisci una volta, usi molte volte!

Perché usare le funzioni?

✅ Vantaggi

1. **Riutilizzabilità**: Scrivi una volta, usa ovunque
2. **Organizzazione**: Codice più strutturato e leggibile
3. **Manutenzione**: Modifica in un solo punto
4. **Testing**: Facile testare componenti isolati
5. **Collaborazione**: Team diversi lavorano su funzioni diverse

Sintassi Base

Python

```python
# Definizione di una funzione
def nome_funzione():
    # corpo della funzione
    print("Questa è una funzione!")

# Chiamata della funzione
nome_funzione()
```

JavaScript

```javascript
// Definizione
function nomeFunzione() {
    // corpo della funzione
    console.log("Questa è una funzione!");
}

// Chiamata
nomeFunzione();
```

Funzioni con Parametri

Parametri singoli

```
def saluta(nome):
    print(f"Ciao {nome}!")
    print("Benvenuto nel corso!")

# Chiamate con argomenti diversi
saluta("Mario")    # Ciao Mario!
saluta("Anna")     # Ciao Anna!
saluta("Luigi")    # Ciao Luigi!
```

Parametri multipli

```
def presenta_persona(nome, età, città):
    print(f"Mi chiamo {nome}")
    print(f"Ho {età} anni")
    print(f"Vivo a {città}")

presenta_persona("Marco", 25, "Roma")
# Output:
# Mi chiamo Marco
# Ho 25 anni
# Vivo a Roma
```

Parametri con valori predefiniti

```
def saluta(nome, messaggio="Buongiorno"):
    print(f"{messaggio}, {nome}!")

saluta("Anna")  # Buongiorno, Anna!
saluta("Marco", "Buonasera")  # Buonasera, Marco!
saluta("Luigi", messaggio="Ciao")  # Ciao, Luigi!
```

Funzioni con Return

Return singolo

```python
def somma(a, b):
    risultato = a + b
    return risultato

# Utilizzo
totale = somma(5, 3)
print(totale)  # 8

# Direttamente in un'espressione
print(f"10 + 20 = {somma(10, 20)}")  # 10 + 20 = 30
```

Funzioni matematiche

```
def calcola_area_rettangolo(base, altezza):
    """Calcola l'area di un rettangolo"""
    return base * altezza

def calcola_area_cerchio(raggio):
    """Calcola l'area di un cerchio"""
    pi = 3.14159
    return pi * raggio ** 2

def calcola_ipotenusa(cateto1, cateto2):
    """Calcola l'ipotenusa usando il teorema di Pitagora"""
    return (cateto1**2 + cateto2**2) ** 0.5

# Utilizzo
area1 = calcola_area_rettangolo(5, 10)  # 50
area2 = calcola_area_cerchio(7)  # ~153.94
ipotenusa = calcola_ipotenusa(3, 4)  # 5.0
```

Return multipli

```
def statistiche_lista(numeri):
    """Calcola min, max e media di una lista"""
    if not numeri:
        return None, None, None

    minimo = min(numeri)
    massimo = max(numeri)
    media = sum(numeri) / len(numeri)

    return minimo, massimo, media

# Utilizzo
dati = [10, 25, 18, 30, 22]
min_val, max_val, media_val = statistiche_lista(dati)

print(f"Minimo: {min_val}")    # 10
print(f"Massimo: {max_val}")   # 30
print(f"Media: {media_val}")   # 21.0
```

Scope delle Variabili

Variabili locali vs globali

```
# Variabile globale
totale_studenti = 100

def aggiungi_studente():
    # Variabile locale
    nuovo_studente = "Mario"
    print(f"Nuovo studente: {nuovo_studente}")
    print(f"Totale studenti: {totale_studenti}")

aggiungi_studente()
# print(nuovo_studente)  # ✗ Errore! Non accessibile fuori dalla funzione

# Modificare variabile globale
contatore = 0

def incrementa():
    global contatore
    contatore += 1

incrementa()
incrementa()
print(contatore)  # 2
```

Esempi Pratici Completi

1. Calcolatrice

```python
def calcolatrice(num1, num2, operazione):
    """Esegue operazioni matematiche di base"""
    if operazione == "+":
        return num1 + num2
    elif operazione == "-":
        return num1 - num2
    elif operazione == "*":
        return num1 * num2
    elif operazione == "/":
        if num2 != 0:
            return num1 / num2
        else:
            return "Errore: divisione per zero"
    else:
        return "Operazione non valida"

# Utilizzo
print(calcolatrice(10, 5, "+"))   # 15
print(calcolatrice(10, 5, "-"))   # 5
print(calcolatrice(10, 5, "*"))   # 50
print(calcolatrice(10, 5, "/"))   # 2.0
print(calcolatrice(10, 0, "/"))   # Errore: divisione per zero
```

2. Validatore email

```python
def valida_email(email):
    """Controlla se un'email è valida (versione semplificata)"""
    # Controlli di base
    if "@" not in email:
        return False

    if email.count("@") != 1:
        return False

    if "." not in email:
        return False

    parti = email.split("@")
    if len(parti[0]) == 0 or len(parti[1]) == 0:
        return False

    return True

# Test
print(valida_email("mario@email.com"))    # True
print(valida_email("anna.rossi@uni.it"))  # True
print(valida_email("invalido@"))          # False
print(valida_email("no-chiocciola.com"))  # False
```

3. Convertitore temperatura

```python
def celsius_to_fahrenheit(celsius):
    """Converte Celsius in Fahrenheit"""
    return (celsius * 9/5) + 32

def fahrenheit_to_celsius(fahrenheit):
    """Converte Fahrenheit in Celsius"""
    return (fahrenheit - 32) * 5/9

def celsius_to_kelvin(celsius):
    """Converte Celsius in Kelvin"""
    return celsius + 273.15

def converti_temperatura(valore, da_unità, a_unità):
    """Convertitore universale di temperatura"""
    # Normalizza a Celsius
    if da_unità == "F":
        celsius = fahrenheit_to_celsius(valore)
    elif da_unità == "K":
        celsius = valore - 273.15
    else:  # Già in Celsius
        celsius = valore

    # Converti alla unità desiderata
    if a_unità == "F":
        return celsius_to_fahrenheit(celsius)
    elif a_unità == "K":
        return celsius_to_kelvin(celsius)
    else:  # Celsius
        return celsius

# Utilizzo
print(f"100°C = {converti_temperatura(100, 'C', 'F')}°F")  # 212.0
print(f"32°F = {converti_temperatura(32, 'F', 'C')}°C")    # 0.0
print(f"0°C = {converti_temperatura(0, 'C', 'K')}K")       # 273.15
```

4. Generatore di password

```python
import random
import string

def genera_password(lunghezza=12, include_numeri=True, include_simboli=True):
    """Genera una password casuale"""
    caratteri = string.ascii_letters  # a-z, A-Z

    if include_numeri:
        caratteri += string.digits  # 0-9

    if include_simboli:
        caratteri += "!@#$%^&*"

    password = ''.join(random.choice(caratteri) for _ in range(lunghezza))
    return password

# Utilizzo
print(genera_password())  # Es: aB3$xY9!mK2p
print(genera_password(8, False, False))  # Es: AbCdEfGh
print(genera_password(16, True, True))  # Es: 9x!K2@mP5#nR8$qT
```

Funzioni Lambda (Anonime)

Python

```python
# Funzione normale
def quadrato(x):
    return x ** 2

# Equivalente con lambda
quadrato_lambda = lambda x: x ** 2

print(quadrato(5))          # 25
print(quadrato_lambda(5))  # 25

# Lambda con più parametri
somma = lambda a, b: a + b
print(somma(3, 7))  # 10

# Uso con map, filter
numeri = [1, 2, 3, 4, 5]
quadrati = list(map(lambda x: x**2, numeri))
print(quadrati)  # [1, 4, 9, 16, 25]

pari = list(filter(lambda x: x % 2 == 0, numeri))
print(pari)  # [2, 4]
```

JavaScript

```javascript
// Arrow function (ES6)
const quadrato = (x) => x ** 2;
const somma = (a, b) => a + b;

console.log(quadrato(5));  // 25
console.log(somma(3, 7));  // 10

// Con array methods
const numeri = [1, 2, 3, 4, 5];
const quadrati = numeri.map(x => x ** 2);
const pari = numeri.filter(x => x % 2 === 0);
```

Documentazione delle Funzioni (Docstrings)

```
def calcola_bmi(peso, altezza):
    """
    Calcola l'Indice di Massa Corporea (BMI).

    Parametri:
        peso (float): Peso in chilogrammi
        altezza (float): Altezza in metri

    Returns:
        float: Valore del BMI
        str: Categoria di peso

    Esempio:
        >>> calcola_bmi(70, 1.75)
        (22.86, "Normopeso")
    """
    bmi = peso / (altezza ** 2)

    if bmi < 18.5:
        categoria = "Sottopeso"
    elif bmi < 25:
        categoria = "Normopeso"
    elif bmi < 30:
        categoria = "Sovrappeso"
    else:
        categoria = "Obesità"

    return round(bmi, 2), categoria

# Utilizzo
bmi, categoria = calcola_bmi(70, 1.75)
print(f"BMI: {bmi} - {categoria}")
```

💡 Best Practices

- **Nome descrittivo**: `calcola_totale()` non `calc()`
- **Una responsabilità**: ogni funzione fa una cosa sola
- **Documenta**: usa docstrings per funzioni complesse
- **Evita side effects**: preferisci return a modifiche globali
- **DRY**: Don't Repeat Yourself - riutilizza il codice

Slide 7: Strutture Dati 📊

Cos'è una struttura dati?

Le **strutture dati** sono modi per organizzare, gestire e memorizzare dati in modo efficiente per accedervi e modificarli.

Analogia: Come organizzare i libri in una biblioteca - scaffali, sezioni, cataloghi.

Perché sono importanti?

- ⚡ **Performance**: Operazioni più veloci
- 🎯 **Organizzazione**: Dati strutturati e accessibili
- 💾 **Efficienza**: Uso ottimale della memoria
- 🔍 **Ricerca**: Trovare dati rapidamente

1. Array/Liste (già viste)

Caratteristiche

- Collezione **ordinata** di elementi
- Accesso per **indice** (posizione)
- **Dimensione dinamica** (in Python)

```
# Creazione
numeri = [10, 20, 30, 40, 50]
misto = [1, "ciao", 3.14, True]

# Operazioni comuni
numeri.append(60)       # Aggiunge alla fine
numeri.insert(0, 5)     # Inserisce in posizione
numeri.remove(30)       # Rimuove elemento
numeri.pop()            # Rimuove ultimo
numeri.sort()           # Ordina
numeri.reverse()        # Inverte

# Complessità temporale
# Accesso: O(1) - molto veloce
```

2. Stack (Pila) 📚

Concetto: LIFO (Last In, First Out)

Come una pila di piatti: l'ultimo inserito è il primo ad uscire.

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        """Aggiunge elemento in cima"""
        self.items.append(item)

    def pop(self):
        """Rimuove e restituisce l'elemento in cima"""
        if not self.is_empty():
            return self.items.pop()
        return None

    def peek(self):
        """Guarda l'elemento in cima senza rimuoverlo"""
        if not self.is_empty():
            return self.items[-1]
        return None

    def is_empty(self):
        """Controlla se lo stack è vuoto"""
        return len(self.items) == 0

    def size(self):
        """Restituisce il numero di elementi"""
        return len(self.items)

# Utilizzo
stack = Stack()
stack.push("primo")
stack.push("secondo")
stack.push("terzo")

print(stack.peek())    # "terzo"
print(stack.pop())     # "terzo"
print(stack.pop())     # "secondo"
```

Applicazioni pratiche

- ⬅️ **Undo/Redo** negli editor
- 🌐 **Navigazione browser** (back button)
- 📞 **Call stack** nei linguaggi di programmazione
- ✅ **Validazione parentesi** in espressioni

```
def controlla_parentesi(espressione):
    """Controlla se le parentesi sono bilanciate"""
    stack = []
    coppie = {'(': ')', '[': ']', '{': '}'}

    for carattere in espressione:
        if carattere in coppie:  # Parentesi aperta
            stack.append(carattere)
        elif carattere in coppie.values():  # Parentesi chiusa
            if not stack or coppie[stack.pop()] != carattere:
                return False

    return len(stack) == 0
```

3. Queue (Coda) 🚶🚶🚶

Concetto: FIFO (First In, First Out)

Come una coda al supermercato: il primo arrivato è il primo servito.

```
from collections import deque

class Queue:
    def __init__(self):
        self.items = deque()

    def enqueue(self, item):
        """Aggiunge elemento alla fine della coda"""
        self.items.append(item)

    def dequeue(self):
        """Rimuove e restituisce il primo elemento"""
        if not self.is_empty():
            return self.items.popleft()
        return None

    def front(self):
        """Guarda il primo elemento senza rimuoverlo"""
        if not self.is_empty():
            return self.items[0]
        return None

    def is_empty(self):
        return len(self.items) == 0

    def size(self):
        return len(self.items)

# Utilizzo
coda = Queue()
coda.enqueue("Cliente 1")
coda.enqueue("Cliente 2")
coda.enqueue("Cliente 3")

print(coda.dequeue())  # "Cliente 1"
print(coda.dequeue())  # "Cliente 2"
```

Applicazioni pratiche

- 🖨️ **Code di stampa**
- 📱 **Gestione messaggi** in chat
- 🎮 **Sistemi di matchmaking** nei giochi
- 💻 **Task scheduling** nei sistemi operativi

```
# Simulazione coda supermercato
class CassaSupermercato:
    def __init__(self, nome):
        self.nome = nome
        self.coda = Queue()

    def aggiungi_cliente(self, cliente):
        self.coda.enqueue(cliente)
        print(f"✓ {cliente} si è accodato alla {self.nome}")

    def servi_cliente(self):
        if not self.coda.is_empty():
            cliente = self.coda.dequeue()
            print(f"💳 Servendo {cliente} alla {self.nome}")
            return cliente
        else:
            print(f"Nessun cliente in attesa alla {self.nome}")
            return None

    def clienti_in_attesa(self):
        return self.coda.size()
```

4. Dizionari/Hash Maps (già visti)

Caratteristiche

- Coppie **chiave-valore**
- Accesso **molto veloce** per chiave
- Chiavi devono essere **uniche**

```
# Rubrica telefonica
rubrica = {
    "Mario Rossi": "333-1234567",
    "Anna Bianchi": "340-9876543",
    "Luigi Verdi": "347-5551234"
}

# Operazioni O(1) - molto veloci!
telefono = rubrica["Mario Rossi"]  # Accesso
rubrica["Sara Neri"] = "328-1112233"  # Inserimento
del rubrica["Luigi Verdi"]  # Rimozione
```

Applicazioni pratiche

- 🖨️ **Database** in memoria
- 🔐 **Cache** di risultati
- 🌐 **Configurazioni** applicazioni
- 📊 **Conteggio occorrenze**

```
def conta_parole(testo):
    """Conta le occorrenze di ogni parola in un testo"""
    parole = testo.lower().split()
    conteggio = {}

    for parola in parole:
        # Rimuovi punteggiatura
        parola_pulita = ''.join(c for c in parola if c.isalnum())

        if parola_pulita:
            if parola_pulita in conteggio:
                conteggio[parola_pulita] += 1
            else:
                conteggio[parola_pulita] = 1
```

5. Set (Insieme) 

Caratteristiche

- Elementi **unici** (no duplicati)
- **Non ordinato**
- Operazioni **insiemistiche** efficienti

```
# Creazione
numeri_unici = {1, 2, 3, 4, 5}
numeri_con_duplicati = {1, 2, 2, 3, 3, 3}
print(numeri_con_duplicati)  # {1, 2, 3} - duplicati rimossi

# Operazioni
numeri_unici.add(6)        # Aggiunge elemento
numeri_unici.remove(1)     # Rimuove elemento
numeri_unici.discard(10)   # Rimuove (no errore se non esiste)

# Operazioni insiemistiche
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

unione = A | B             # {1, 2, 3, 4, 5, 6, 7, 8}
```

Applicazioni pratiche

```
# 1. Rimozione duplicati
numeri_con_duplicati = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
numeri_unici = list(set(numeri_con_duplicati))
print(numeri_unici)  # [1, 2, 3, 4]

# 2. Analisi utenti social
utenti_facebook = {"Mario", "Anna", "Luigi", "Sara"}
utenti_instagram = {"Anna", "Luigi", "Paolo", "Chiara"}

# Utenti su entrambe le piattaforme
su_entrambe = utenti_facebook & utenti_instagram
print(f"Su entrambe: {su_entrambe}")  # {'Anna', 'Luigi'}

# Utenti totali
tutti_utenti = utenti_facebook | utenti_instagram
print(f"Totali: {len(tutti_utenti)}")  # 6

# Solo su Facebook
solo_facebook = utenti_facebook - utenti_instagram
print(f"Solo Facebook: {solo_facebook}")  # {'Mario', 'Sara'}
```

6. Linked List (Lista Concatenata) 🔗

Concetto

Una sequenza di **nodi** collegati, dove ogni nodo contiene:

- Un **valore** (dato)
- Un **riferimento** al nodo successivo

```
class Nodo:
    def __init__(self, valore):
        self.valore = valore
        self.successivo = None

class ListaConcatenata:
    def __init__(self):
        self.testa = None

    def aggiungi_in_testa(self, valore):
        """Aggiunge un nodo all'inizio"""
        nuovo_nodo = Nodo(valore)
        nuovo_nodo.successivo = self.testa
        self.testa = nuovo_nodo

    def aggiungi_in_coda(self, valore):
        """Aggiunge un nodo alla fine"""
        nuovo_nodo = Nodo(valore)

        if self.testa is None:
            self.testa = nuovo_nodo
            return

        corrente = self.testa
        while corrente.successivo:
            corrente = corrente.successivo

        corrente.successivo = nuovo_nodo

    def stampa_lista(self):
        """Stampa tutti gli elementi"""
        corrente = self.testa
        elementi = []

        while corrente:
            elementi.append(str(corrente.valore))
            corrente = corrente.successivo

        print(" -> ".join(elementi))

    def cerca(self, valore):
        """Cerca un valore nella lista"""
        corrente = self.testa
        posizione = 0

        while corrente:
            if corrente.valore == valore:
```

Confronto Strutture Dati

Struttura	Accesso	Ricerca	Inserimento	Rimozione	Uso Principale
Array/Lista	O(1)	O(n)	O(n)	O(n)	Accesso indicizzato
Stack	O(n)	O(n)	O(1)	O(1)	LIFO, undo/redo
Queue	O(n)	O(n)	O(1)	O(1)	FIFO, task scheduling
Dizionario	O(1)	O(1)	O(1)	O(1)	Lookup veloce
Set	-	O(1)	O(1)	O(1)	Unicità, operazioni insieme
Linked List	O(n)	O(n)	O(1)	O(1)	Inserimenti frequenti

Scegliere la struttura giusta

```
# ✅ USA LISTA quando:
# - Hai bisogno di accesso per indice
# - L'ordine è importante
menu_ristorante = ["Antipasto", "Primo", "Secondo", "Dolce"]

# ✅ USA DIZIONARIO quando:
# - Hai coppie chiave-valore
# - Serve lookup veloce
prezzi_menu = {"Antipasto": 8, "Primo": 12, "Secondo": 15}

# ✅ USA SET quando:
# - Elementi devono essere unici
# - Serve controllo appartenenza veloce
ingredienti_allergeni = {"glutine", "lattosio", "uova"}

# ✅ USA STACK quando:
# - Serve comportamento LIFO
# - Undo/redo, navigazione
cronologia_browser = Stack()

# ✅ USA QUEUE quando:
# - Serve comportamento FIFO
# - Gestione ordini, task
coda_stampa = Queue()
```

💡 Best Practices

- **Scegli in base alle operazioni** più frequenti
- **Considera la complessità** temporale
- **Testa con dati realistici** prima di decidere
- **Non ottimizzare prematuramente**: parti dal semplice