بخش 3: توابع (Functions) در پایتون

بخش 3: توابع (Functions)

1. تعریف توابع

در زبان پایتون، توابع بلوک‌هایی از کد هستند که می‌توانند یک یا چند عملیات را انجام دهند و می‌توانند ورودی‌هایی دریافت کرده و خروجی‌هایی تولید کنند. توابع به شما این امکان را می‌دهند که کدهای خود را سازماندهی کنید، از تکرار جلوگیری کنید و ساختار کد را قابل فهم‌تر کنید.

1.1. نحوه تعریف و استفاده از توابع

برای تعریف یک تابع در پایتون، از کلمه کلیدی `def` استفاده می‌شود، به دنبال آن نام تابع و سپس پارامترهای تابع (در صورت وجود) داخل پرانتز آورده می‌شود. کد داخل تابع باید با استفاده از فاصله گذاری (indentation) مشخص شود. در نهایت، می‌توان تابع را با استفاده از نام تابع فراخوانی کرد.

ساختار تعریف یک تابع:

```
def function_name(parameters):
    کدهایی که داخل تابع اجرا می‌شوند #
    return value  # مقدار برگشتی از تابع
```

- `function_name` : نام تابع است که شما آن را برای فراخوانی تابع استفاده خواهید کرد.
- `parameters` : پارامترهایی که تابع می‌تواند دریافت کند. این پارامترها ورودی‌هایی هستند که تابع برای انجام عملیات به آن‌ها نیاز دارد.
- `return value` : مقداری که تابع به عنوان نتیجه عملیات خود بازمی‌گرداند.

1.2. معرفی ساختار پایه تابع و فراخوانی آن

به عنوان مثال، یک تابع ساده که دو عدد را جمع می‌کند، به صورت زیر است:

```
def add_numbers(a, b):
    return a + b
```

در این مثال:

- تابع `add_numbers` دو پارامتر به نام‌های `a` و `b` دریافت می‌کند.
- داخل تابع، جمع این دو عدد انجام شده و نتیجه آن با استفاده از دستور `return` به تابع بازگشت می‌کند.
- تابع بعد از بازگشت از دستور `return` پایان می‌یابد و مقدار برگشتی را به فراخوانی تابع ارسال می‌کند.

برای فراخوانی این تابع و استفاده از آن، می‌توان از کد زیر استفاده کرد:

```
result = add_numbers(3, 5)
print(result)
```

در اینجا:

- `add_numbers(3, 5)` تابع `add_numbers` را فراخوانی کرده و به آن مقادیر `3` و `5` را به عنوان ورودی می‌دهد.

- نتیجه‌ی بازگشتی تابع (که 8 است) در متغیر `result` ذخیره می‌شود و سپس با دستور `print` چاپ می‌شود.

خروجی برنامه:

```
8
```

1.3. توابع بدون مقدار برگشتی

توابع در پایتون می‌توانند مقدار برگشتی نداشته باشند. در چنین حالتی، تابع می‌تواند عملیاتی را انجام دهد، اما چیزی به عنوان نتیجه برنگرداند. به طور پیش‌فرض، در صورتی که تابع هیچ مقدار برگشتی نداشته باشد، پایتون مقدار `None` را باز می‌گرداند.

مثال از تابع بدون مقدار برگشتی:

```
def print_greeting(name):
    print(f"Hello, {name}!")
```

این تابع هیچ مقدار بازگشتی ندارد، بلکه صرفاً یک پیام را چاپ می‌کند. می‌توان این تابع را به شکل زیر فراخوانی کرد:

```
print_greeting("Alice")
```

خروجی برنامه:

```
Hello, Alice!
```

1.4. نکات مهم هنگام تعریف توابع

- **نام تابع** باید یک نام معتبر (که باید با یک حرف یا آندرلاین شروع شود) باشد و نباید نام‌های کلیدی پایتون (مثل `True` ,`def` ,`return` و ...) را استفاده کرد.
- **پارامترها** می‌توانند به عنوان ورودی برای انجام محاسبات یا عملیات‌های دیگر در داخل تابع استفاده شوند.
- اگر در داخل تابع از دستور `return` استفاده شود، تابع بلافاصله پایان می‌یابد و نتیجه‌ای باز می‌گرداند. در غیر این صورت، تابع به صورت پیش‌فرض `None` را باز می‌گرداند.

1.5. فراخوانی توابع

پس از تعریف یک تابع، برای استفاده از آن، فقط کافی است نام تابع را همراه با پارامترهای مورد نظر (در صورت نیاز) فراخوانی کنیم. توابع می‌توانند در هر جایی از برنامه (پس از تعریف شدن) فراخوانی شوند.

```
def multiply(a, b):
    return a * b

result = multiply(4, 6)
print(result)
```

در اینجا:

- تابع `multiply` تعریف شده است که دو عدد را به هم ضرب می‌کند.
- سپس این تابع فراخوانی می‌شود و نتیجه‌ی آن (24) در متغیر `result` ذخیره می‌شود و چاپ می‌شود.

خروجی برنامه:

```
24
```

در این بخش، شما یاد گرفتید که چگونه توابع را در پایتون تعریف کنید، چطور آن‌ها را فراخوانی کنید و مقادیر بازگشتی را از آن‌ها دریافت کنید. این مبانی به شما کمک خواهد کرد تا برنامه‌های پیچیده‌تری بسازید و کدهای خود را سازماندهی کنید.

==========

پارامترها و مقادیر بازگشتی (Return)

در این بخش به نحوه تعریف پارامترهای تابع و نحوه استفاده از آن‌ها برای دریافت داده‌ها و همچنین نحوه بازگشت مقادیر از تابع با استفاده از دستور `return` پرداخته خواهد شد. همچنین در مورد بازگشت بیش از یک مقدار از تابع نیز توضیح داده می‌شود.

1. تعریف پارامترهای تابع و نحوه استفاده از آن‌ها

پارامترها، ورودی‌هایی هستند که به تابع داده می‌شوند تا در هنگام اجرای تابع از آن‌ها استفاده شود. هر تابع می‌تواند یک یا چند پارامتر داشته باشد که هنگام فراخوانی تابع، مقدارهای مشخصی به آن‌ها ارسال می‌شود.

ساختار پارامترهای تابع:

```
def function_name(parameter1, parameter2):
    # عملیات روی پارامترها
    return result
```

در اینجا:

- `parameter1` و `parameter2` پارامترهای تابع هستند.
- هنگام فراخوانی تابع، مقادیر خاصی به این پارامترها ارسال می‌شود.

مثال:

```
def greet(name, age):
    print(f"Hello {name}, you are {age} years old.")

greet("Alice", 30)
```

در اینجا:

- تابع `greet` دو پارامتر به نام‌های `name` و `age` دارد.
- هنگامی که تابع فراخوانی می‌شود، مقادیر `"Alice"` و `30` به ترتیب به پارامترهای `name` و `age` ارسال می‌شود.

خروجی برنامه:

```
Hello Alice, you are 30 years old.
```

2. نحوه بازگشت مقادیر از تابع با استفاده از دستور `return`

دستور `return` برای بازگشت مقدار از یک تابع استفاده می‌شود. هنگامی که یک تابع به دستور `return` می‌رسد، مقدار بازگشتی به فراخوانی تابع باز می‌گردد و اجرای تابع پایان می‌یابد. شما می‌توانید با استفاده از `return` هر نوع داده‌ای از جمله اعداد، رشته‌ها، لیست‌ها و غیره را بازگردانید.

ساختار دستور `return`:

```
def function_name():
    return value
```

مثال:

```
def add(a, b):
    return a + b

result = add(3, 5)
print(result)
```

در اینجا:

- تابع `add` دو پارامتر `a` و `b` را دریافت می‌کند و جمع آن‌ها را بازمی‌گرداند.
- دستور `return a + b` نتیجه جمع را بازمی‌گرداند.
- مقدار بازگشتی به متغیر `result` اختصاص داده می‌شود و سپس چاپ می‌شود.

خروجی برنامه:

```
8
```

3. کاربرد `return` برای بازگشت بیش از یک مقدار (از طریق Tuple)

پایتون این امکان را فراهم می‌کند که یک تابع بیش از یک مقدار را از طریق یک ساختار داده ترکیبی مثل `tuple` بازگرداند. شما می‌توانید چندین مقدار را در یک `tuple` قرار داده و آن را به عنوان خروجی از تابع بازگردانید.

ساختار بازگشت چندین مقدار:

```
def function_name():
    return value1, value2, value3
```

در اینجا:

- چندین مقدار از تابع بازگشت داده می‌شود، و پایتون به طور خودکار آن‌ها را در یک `tuple` قرار می‌دهد.

مثال:

```
def calculate(a, b):
    sum_result = a + b
    difference = a - b
    product = a * b
    return sum_result, difference, product

result = calculate(10, 5)
print(result)
```

در اینجا:

- تابع `calculate` سه مقدار را محاسبه کرده و آن‌ها را به صورت یک `tuple` باز می‌گرداند.
- نتیجه به متغیر `result` اختصاص داده می‌شود و سپس چاپ می‌شود.

خروجی برنامه:

```
(15, 5, 50)
```

در صورتی که بخواهید از مقادیر بازگشتی به صورت جداگانه استفاده کنید، می‌توانید از بازگشت چندگانه (Multiple Assignment) استفاده کنید:

```
sum_result, difference, product = calculate(10, 5)
print(sum_result, difference, product)
```

خروجی برنامه:

```
15 5 50
```

نکات:

- در پایتون، هنگامی که چند مقدار را با استفاده از `return` باز می‌گردانید، این مقادیر به طور خودکار در یک `tuple` قرار می‌گیرند.
- شما می‌توانید مقادیر بازگشتی را به صورت مجزا نیز به متغیرهای مختلف اختصاص دهید.

در این بخش، شما یاد گرفتید که چگونه پارامترها را به توابع ارسال کرده و از آن‌ها استفاده کنید و همچنین چطور می‌توانید از دستور `return` برای بازگشت مقادیر از تابع استفاده کنید. همچنین با نحوه بازگشت چندین مقدار با استفاده از `tuple` آشنا شدید که می‌تواند به شما کمک کند تا داده‌های مختلف را به راحتی از توابع بازگردانید.

==========

توابع بدون مقدار برگشتی

توابع بدون مقدار برگشتی، توابعی هستند که تنها یک یا چند عملیات خاص را انجام می‌دهند و نیازی به بازگشت داده ندارند. این توابع می‌توانند عملیات‌هایی مانند چاپ اطلاعات، تغییر وضعیت متغیرهای خارجی، یا اجرای سایر وظایف جانبی را انجام دهند.

1. تعریف توابع بدون مقدار برگشتی

یک تابع بدون مقدار برگشتی، معمولاً با استفاده از دستور `return` به هیچ مقداری بازنمی‌گردد. در این حالت، تابع فقط عملیات خاصی را انجام می‌دهد و سپس به پایان می‌رسد. اگر تابع هیچ مقداری را برنگرداند، به صورت پیش‌فرض مقدار `None` بازمی‌گرداند، ولی این مقدار معمولاً در توابع بدون مقدار برگشتی مورد استفاده قرار نمی‌گیرد.

ساختار تابع بدون مقدار برگشتی:

```
def function_name():
    # انجام عملیات
    print("This is a function with no return value.")
```

در اینجا:

- تابع `function_name` هیچ مقدار برگشتی ندارد.
- تنها عملیات داخل آن انجام می‌شود.

مثال:

```
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

در اینجا:

- تابع `greet` تنها عملیاتی مانند چاپ پیامی به کنسول انجام می‌دهد و هیچ مقدار بازگشتی ندارد.

خروجی برنامه:

```
Hello, Alice!
```

2. استفاده از `print` در توابع بدون مقدار برگشتی

تابع بدون مقدار برگشتی می‌تواند برای نمایش اطلاعات به کاربر از دستور `print` استفاده کند. این کار به خصوص برای توابعی که تنها به خروجی اطلاعات نیاز دارند و نیازی به بازگشت مقدار ندارند، مفید است.

مثال:

```
def display_sum(a, b):
    result = a + b
    print(f"The sum of {a} and {b} is {result}")

display_sum(10, 5)
```

در اینجا:

- تابع `display_sum` دو عدد را دریافت کرده و حاصل جمع آن‌ها را چاپ می‌کند.
- این تابع هیچ مقدار بازگشتی ندارد، بلکه فقط نتیجه عملیات را به صورت خروجی در کنسول نمایش می‌دهد.

خروجی برنامه:

```
The sum of 10 and 5 is 15
```

3. اعمال تغییرات در متغیرهای خارجی در توابع بدون مقدار برگشتی

در پایتون، می‌توان در توابع بدون مقدار برگشتی، تغییراتی را در متغیرهای خارجی (متغیرهایی که خارج از تابع تعریف شده‌اند) اعمال کرد. این عملیات معمولاً برای تغییر وضعیت داده‌ها یا انجام محاسبات در خارج از تابع کاربرد دارد.

مثال:

```
def update_value(a):
    a += 5  # تغییر مقدار متغیر
    print(f"Updated value: {a}")

x = 10
update_value(x)
print(f"Value of x after function call: {x}")
```

در اینجا:

- تابع `update_value` مقدار ورودی `a` را تغییر می‌دهد و پس از تغییر آن را چاپ می‌کند.
- با وجود این‌که متغیر `x` به تابع ارسال می‌شود، این تغییرات تنها در داخل تابع اعمال می‌شود، زیرا متغیر `x` به عنوان یک مقدار (By Value) به تابع ارسال شده است.

خروجی برنامه:

```
Updated value: 15
Value of x after function call: 10
```

در اینجا:

- مقدار `x` در داخل تابع تغییر می‌کند، ولی چون در پایتون متغیرها به صورت مقادیر ارسال می‌شوند (نه به صورت ارجاع)، تغییرات فقط در داخل تابع مؤثر هستند.

برای تغییر متغیرهای خارجی از نوع "ارجاعی" (مثل لیست‌ها یا دیکشنری‌ها) در توابع بدون مقدار برگشتی، می‌توان مستقیماً آن‌ها را تغییر داد:

مثال با لیست‌ها (متغیر ارجاعی):

```
def append_to_list(lst, item):
    lst.append(item)

my_list = [1, 2, 3]
append_to_list(my_list, 4)
print(my_list)
```

در اینجا:

- تابع `append_to_list` به لیست `lst` یک آیتم جدید اضافه می‌کند.
- چون لیست‌ها به صورت ارجاعی (By Reference) به توابع ارسال می‌شوند، تغییرات در داخل تابع مستقیماً در متغیر خارجی `my_list` اعمال می‌شود.

خروجی برنامه:

```
[1, 2, 3, 4]
```

نکات:

- توابع بدون مقدار برگشتی معمولاً برای انجام وظایف جانبی مانند چاپ اطلاعات یا تغییر وضعیت داده‌ها استفاده می‌شوند.
- دستور `return` در توابع بدون مقدار برگشتی معمولاً وجود ندارد، اما اگر هم وجود داشته باشد، به طور پیش‌فرض مقدار `None` بازمی‌گرداند.
- در پایتون، متغیرها به صورت مقادیر به توابع ارسال می‌شوند (برای متغیرهای عددی، رشته‌ها و تاپل‌ها)، اما برای انواع داده ارجاعی مانند لیست‌ها و دیکشنری‌ها، تغییرات به صورت مستقیم در خارج از تابع اعمال می‌شود.

در این بخش، شما یاد گرفتید که توابع بدون مقدار برگشتی چگونه عمل می‌کنند و در چه مواقعی از آن‌ها استفاده می‌شود. همچنین به شما نشان داده شد که چگونه می‌توانید از دستور `print` برای نمایش خروجی‌ها و از متغیرهای خارجی برای ذخیره تغییرات در توابع بدون بازگشت استفاده کنید.

==========

2. توابع با تعداد متغیر ورودی متغیر (Arbitrary Arguments)

در پایتون، زمانی که می‌خواهید یک تابع تعریف کنید که تعداد آرگومان‌های ورودی آن ثابت نیست و می‌تواند تعداد متغیری از آرگومان‌ها را دریافت کند، از ویژگی‌هایی مانند `args*` و `kwargs**` استفاده می‌کنیم.

1. استفاده از `args*\`

پارامتر `args*` به شما این امکان را می‌دهد که تعداد نامحدودی از آرگومان‌ها را به یک تابع ارسال کنید. وقتی از `args*` در تعریف تابع استفاده می‌کنید، پایتون تمامی آرگومان‌های ارسالی را در قالب یک تاپل (tuple) قرار می‌دهد.

- **چطور کار می‌کند؟**
  وقتی تابعی با `args*` تعریف می‌شود، تمام آرگومان‌هایی که به آن تابع ارسال می‌شوند، به صورت یک تاپل در `args` قرار می‌گیرند.
- **نحوه استفاده از `args*\`:**
  برای استفاده از `args*`، کافی است در تعریف تابع، قبل از نام پارامتر از `*` استفاده کنید. سپس می‌توانید این پارامتر را همانند یک تاپل (tuple) در داخل تابع استفاده کنید.

مثال:

```
def add_numbers(*args):
    total = 0
    for num in args:
        total += num
    return total

# فراخوانی تابع با تعداد متغیر آرگومان‌ها
result = add_numbers(10, 20, 30)
print(result)
```

در اینجا:

- تابع `add_numbers` به تعداد نامحدودی از اعداد به عنوان آرگومان ورودی می‌پذیرد.
- در داخل تابع، با استفاده از یک حلقه، تمامی مقادیر موجود در `args` جمع می‌شوند.
- مقدار `total` که حاصل جمع است، به عنوان نتیجه برگشت داده می‌شود.

خروجی برنامه:

```
60
```

در این مثال، تعداد آرگومان‌ها متغیر است و می‌توانیم هر تعداد عدد را به تابع ارسال کنیم.

2. ارسال لیستی از مقادیر به تابع با استفاده از `args*\`

اگر یک لیست یا تاپل داشته باشید و بخواهید آن را به عنوان آرگومان به تابع ارسال کنید، می‌توانید از `args*` برای "گسترش" لیست یا تاپل و ارسال مقادیر آن به تابع استفاده کنید.

مثال:

```
def print_numbers(*args):
    for number in args:
        print(number)

numbers_list = [1, 2, 3, 4, 5]
print_numbers(*numbers_list)
```

در اینجا:

- از `numbers_list*` برای ارسال هر عدد موجود در لیست به عنوان آرگومان جداگانه به تابع `print_numbers` استفاده شده است.
- به این صورت که هر عدد داخل لیست به عنوان یک پارامتر مستقل به تابع ارسال می‌شود.

خروجی برنامه:

```
1
2
3
4
5
```

این نشان می‌دهد که چطور می‌توانید یک لیست را با استفاده از `args*` به تابع ارسال کنید و مقادیر داخل آن را به صورت جداگانه دریافت کنید.

3. جمع کردن اعداد مختلف ارسال شده به تابع با استفاده از `args*\`

یکی از کاربردهای متداول `args*`، جمع کردن تعداد متغیر آرگومان‌ها است. به راحتی می‌توانید با استفاده از یک حلقه یا توابع built-in مانند `()sum` تمامی اعداد ارسال شده را جمع کنید.

مثال:

```python
def sum_numbers(*args):
    return sum(args)

# فراخوانی تابع با آرگومان‌های مختلف
result = sum_numbers(1, 2, 3, 4, 5)
print(result)
```

در اینجا:

- از تابع `sum()` برای جمع کردن تمامی مقادیر موجود در `args` استفاده می‌شود.
- تابع `sum_numbers` با دریافت هر تعداد عدد، آن‌ها را جمع می‌کند.

خروجی برنامه:

```
15
```

4. مزایای استفاده از `*args`:

- **انعطاف‌پذیری:** با استفاده از `*args` می‌توانید توابعی بنویسید که تعداد متغیری از آرگومان‌ها را پذیرش کنند.
- **ساده‌سازی کد:** به جای اینکه تعداد زیادی پارامتر برای تابع تعریف کنید، می‌توانید فقط از یک پارامتر با نام `*args` استفاده کنید و تعداد نامحدودی از آرگومان‌ها را به آن ارسال کنید.
- **پشتیبانی از لیست‌ها و تاپل‌ها:** با استفاده از `*args` می‌توانید به راحتی لیست‌ها یا تاپل‌ها را به عنوان آرگومان به تابع ارسال کنید.

نکات:

- `*args` همیشه باید در آخرین پارامترهای تابع قرار بگیرد. اگر بخواهید هم از پارامترهای نامشخص و هم از پارامترهای مشخص استفاده کنید، باید پارامترهای مشخص ابتدا آمده و سپس از `*args` برای پارامترهای متغیر استفاده کنید.
- به جز `*args`، از `**kwargs` نیز می‌توان برای پذیرش آرگومان‌های کلیدی (keyword arguments) استفاده کرد که مشابه `*args` اما به صورت دیکشنری عمل می‌کند.

در این بخش شما یاد گرفتید که چگونه از `*args` برای پذیرش تعداد نامحدودی از آرگومان‌ها استفاده کنید. همچنین نمونه‌هایی از کاربرد آن در جمع کردن اعداد و ارسال لیست به تابع را مشاهده کردید. این تکنیک‌ها به شما کمک می‌کنند تا توابعی بنویسید که انعطاف‌پذیرتر و مقیاس‌پذیرتر باشند.

==========

استفاده از `**kwargs`

در پایتون، `**kwargs` به شما این امکان را می‌دهد که تعداد نامحدودی از آرگومان‌ها را به صورت کلید-مقدار (keyword arguments) به یک تابع ارسال کنید. به عبارت دیگر، با استفاده از `**kwargs` می‌توانید آرگومان‌های ورودی را به صورت جفت‌های کلید-مقدار دریافت کنید. در این حالت، آرگومان‌های ارسالی به صورت یک دیکشنری به تابع منتقل می‌شوند.

1. معرفی `kwargs**\` برای ارسال آرگومان‌های کلید-مقدار به تابع

پارامتر `kwargs**` به تابع این امکان را می‌دهد که یک دیکشنری از کلیدها و مقادیر دریافت کند. برخلاف `args*` که آرگومان‌ها را به صورت یک تاپل می‌گیرد، `kwargs**` کلیدها را به عنوان نام‌گذاری برای مقادیر و در قالب یک دیکشنری دریافت می‌کند.

چطور کار می‌کند؟
وقتی از `kwargs**` استفاده می‌کنید، تمامی آرگومان‌های کلید-مقدار ارسال شده به تابع به صورت یک دیکشنری به نام `kwargs` ذخیره می‌شوند که کلیدها نام آرگومان‌ها و مقادیر مربوط به آنها هستند.

2. توضیح نحوه استفاده از `kwargs**\` برای دریافت تعداد نامحدود از آرگومان‌های کلید-مقدار

برای استفاده از `kwargs**`، باید در تعریف تابع از دو ستاره (`**`) قبل از نام پارامتر استفاده کنید. این پارامتر به صورت یک دیکشنری عمل کرده و هر جفت کلید-مقدار که به تابع ارسال می‌شود در آن ذخیره خواهد شد.

مثال:

```
def print_person_details(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

# فراخوانی تابع با آرگومان•های کلید-مقدار مختلف
print_person_details(name="Ali", age=30, occupation="Engineer")
```

در اینجا:

- تابع `print_person_details` با استفاده از `kwargs**` تمام آرگومان‌های کلید-مقدار را دریافت می‌کند.
- داخل تابع، از حلقه `for` و متد `.items()` برای دسترسی به کلیدها و مقادیر دیکشنری استفاده شده است.
- می‌توانیم برای هر فرد، اطلاعاتی مانند نام، سن، شغل و هر جزئیات دیگری را به صورت کلید-مقدار ارسال کنیم.

خروجی برنامه:

```
name: Ali
age: 30
occupation: Engineer
```

3. مثال از ایجاد یک تابع که جزئیات مربوط به یک شخص را دریافت کرده و چاپ کند

در این مثال، تابعی نوشته‌ایم که جزئیات مربوط به یک شخص شامل نام، سن و شغل را دریافت کرده و آن‌ها را چاپ می‌کند. این اطلاعات به صورت کلید-مقدار ارسال می‌شوند.

مثال:

```python
def describe_person(**kwargs):
    name = kwargs.get('name', 'Unknown')
    age = kwargs.get('age', 'Not provided')
    occupation = kwargs.get('occupation', 'Not specified')

    print(f"Name: {name}")
    print(f"Age: {age}")
    print(f"Occupation: {occupation}")

# فراخوانی تابع با آرگومان•های کلید-مقدار
describe_person(name="Sara", age=25, occupation="Teacher")
```

در اینجا:

- از متد `()get.` برای استخراج مقادیر از دیکشنری `kwargs` استفاده شده است.
- اگر کلیدی موجود نباشد، مقدار پیش‌فرض به عنوان مقدار برگشتی در نظر گرفته می‌شود (مثلاً `'Unknown'` برای `name`).

خروجی برنامه:

```
Name: Sara
Age: 25
Occupation: Teacher
```

4. مزایای استفاده از `kwargs**\`:

- **انعطاف‌پذیری بالا:** می‌توانید تعداد نامحدودی از آرگومان‌ها را به تابع ارسال کنید بدون اینکه نیاز به مشخص کردن دقیق تعداد پارامترها در تعریف تابع داشته باشید.
- **قابلیت استفاده از نام‌های معنادار:** به جای ارسال مقادیر بدون نام، با استفاده از `kwargs**` می‌توانید از نام‌های معنادار (کلیدها) برای مقادیر استفاده کنید که کد را خوانا و قابل فهم‌تر می‌کند.
- **پشتیبانی از مقادیر پیش‌فرض:** می‌توانید مقادیر پیش‌فرض برای هر کلید تعیین کنید، که در صورتی که کلیدی ارسال نشد، از آن استفاده شود.

نکات:

- `kwargs**` باید در آخرین پارامترهای تابع قرار بگیرد. اگر هم از `args*` و هم از `kwargs**` استفاده می‌کنید، باید `args*` اول و `kwargs**` در انتها قرار گیرد.
- می‌توانید همزمان از پارامترهای معمولی، `args*` و `kwargs**` در یک تابع استفاده کنید.

با استفاده از `kwargs**` می‌توانید توابعی بنویسید که به راحتی تعداد نامحدودی از آرگومان‌های کلید-مقدار را دریافت کنند و با استفاده از آن‌ها، اطلاعات متنوعی را پردازش یا نمایش دهید. این تکنیک به شما انعطاف‌پذیری بالایی در نوشتن کد می‌دهد.

==========

3. توابع بازگشتی (Recursive Functions)

توابع بازگشتی، توابعی هستند که درون خود به طور مستقیم یا غیرمستقیم فراخوانی می‌شوند. استفاده از بازگشت در برنامه‌نویسی می‌تواند به حل مسائل پیچیده‌ای که به صورت تدریجی تقسیم می‌شوند کمک کند. توابع بازگشتی می‌توانند برای حل مسائلی که به همان نوع مسئله کوچکتر تقسیم می‌شوند، بسیار مفید باشند.

1. مفهوم بازگشتی

بازگشت به این معنی است که یک تابع خودش را در حین اجرای خودش فراخوانی کند. معمولاً از این تکنیک برای حل مسائلی استفاده می‌شود که می‌توانند به مشکلات مشابه اما ساده‌تر تقسیم شوند.

به عنوان مثال، اگر بخواهیم یک مسئله پیچیده مانند محاسبه فاکتوریل یک عدد را حل کنیم، می‌توانیم این کار را با استفاده از بازگشت انجام دهیم. در واقع، فاکتوریل یک عدد `n` برابر است با `n` ضربدر فاکتوریل `n-1`. این فرآیند ادامه پیدا می‌کند تا به عدد 1 برسیم.

2. مثال‌هایی از توابع بازگشتی

- **محاسبه فاکتوریل (Factorial)**

فاکتوریل یک عدد `n` با فرمول زیر محاسبه می‌شود:

```
n! = n × (n-1)!
```

شرط پایانی یا پایه (base case) زمانی است که `n` برابر با 1 می‌شود و در این حالت فاکتوریل برابر با 1 است.

کد پایتون برای محاسبه فاکتوریل با استفاده از بازگشت:

```python
def factorial(n):
    # شرط پایانی برای جلوگیری از حلقه بی‌پایان
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)

# فراخوانی تابع برای محاسبه فاکتوریل 5
print(factorial(5))  # خروجی: 120
```

در اینجا:

- تابع `factorial` خودش را برای مقادیر کوچکتر از `n` فراخوانی می‌کند.
- وقتی که `n == 1` می‌شود، تابع بازگشت را متوقف می‌کند و مقدار 1 را باز می‌گرداند.
- این فرآیند در نهایت به این شکل ختم می‌شود:

```
factorial(5) = 5 * factorial(4)
factorial(4) = 4 * factorial(3)
factorial(3) = 3 * factorial(2)
factorial(2) = 2 * factorial(1)
factorial(1) = 1
```

خروجی برنامه:

```
120
```

- **دنباله فیبوناچی (Fibonacci Sequence)**

دنباله فیبوناچی یک دنباله عددی است که در آن هر عدد برابر با جمع دو عدد قبلی است. اولین دو عدد در دنباله برابر با 0 و 1 هستند، به این ترتیب:

```
F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2) for n > 1
```

کد پایتون برای محاسبه عدد فیبوناچی با استفاده از بازگشت:

```python
def fibonacci(n):
    # شرط پایانی برای جلوگیری از حلقه بی‌پایان
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# فراخوانی تابع برای محاسبه عدد فیبوناچی برای n=6
print(fibonacci(6))  # خروجی: 8
```

در اینجا:

- تابع `fibonacci` دو بار خودش را فراخوانی می‌کند تا دنباله فیبوناچی را محاسبه کند.
- وقتی که `n == 0` یا `n == 1` می‌شود، تابع بازگشت را متوقف می‌کند و مقادیر 0 یا 1 را باز می‌گرداند.

خروجی برنامه:

```
8
```

3. تعریف شرایط پایانی برای جلوگیری از حلقه‌های بی‌پایان

شرط پایانی یا **Base Case** برای هر تابع بازگشتی ضروری است. اگر این شرط به درستی تعریف نشود، تابع به صورت بی‌پایان خودش را فراخوانی می‌کند و منجر به **Stack Overflow** (اشباع حافظه پشته) خواهد شد.

در مثال‌های فوق، شرایط پایانی به این صورت تعریف شده است:

- در تابع `factorial`, وقتی `n == 1` شد، تابع متوقف می‌شود.
- در تابع `fibonacci`, وقتی `n == 0` یا `n == 1` شد، تابع متوقف می‌شود.

این شرایط پایانی باعث می‌شوند که تابع بازگشتی به درستی متوقف شده و نتایج صحیح بازگشت داده شوند.

4. مزایای استفاده از توابع بازگشتی

- **سادگی و خوانایی:** بسیاری از مسائل پیچیده به صورت طبیعی با استفاده از بازگشت حل می‌شوند. در این حالت، کد ساده‌تر و خواناتر می‌شود.
- **تقسیم‌بندی مسئله:** با استفاده از بازگشت، می‌توان یک مسئله پیچیده را به مسائل کوچکتر و مشابه تقسیم کرد و به این ترتیب حل آن راحت‌تر می‌شود.
- **کاربردهای الگوریتمی:** بسیاری از الگوریتم‌ها مانند جستجوی دودویی، الگوریتم‌های تقسیم و حل (Divide and Conquer)، جستجو در گراف‌ها و درخت‌ها، مرتب‌سازی‌ها و غیره از توابع بازگشتی بهره می‌برند.

5. معایب و چالش‌ها

- **حافظه و کارایی:** توابع بازگشتی ممکن است منابع زیادی از حافظه مصرف کنند، به ویژه زمانی که تعداد بازگشت‌ها زیاد باشد. این موضوع می‌تواند باعث ایجاد مشکلات عملکردی شود.
- **خطر Stack Overflow:** در صورتی که شرط پایانی نادرست باشد، یا تعداد فراخوانی‌ها زیاد باشد، ممکن است به Stack Overflow منجر شود.

توابع بازگشتی ابزاری قدرتمند برای حل مسائل مشابه به‌طور خودکار هستند و می‌توانند برنامه‌های ما را ساده‌تر و خواناتر کنند. با این حال، باید از شرایط پایانی مناسب برای جلوگیری از مشکلات حافظه و عملکرد استفاده کنیم.

==========

3. حل مسائل با استفاده از توابع بازگشتی

توابع بازگشتی به‌طور خاص برای حل مسائل پیچیده‌ای که می‌توانند به زیرمسائل مشابه تقسیم شوند، بسیار مفید هستند. یکی از کاربردهای رایج توابع بازگشتی، حل مسائل ساختاری مانند جستجو در درخت‌ها یا گراف‌ها است. این ساختارها اغلب به‌صورت طبیعی با استفاده از بازگشت حل می‌شوند.

1. استفاده از توابع بازگشتی برای جستجو در درخت‌ها و گراف‌ها

درخت‌ها و گراف‌ها ساختارهای داده‌ای پیچیده‌ای هستند که به‌طور طبیعی با استفاده از توابع بازگشتی به بهترین شکل قابل پیمایش و جستجو هستند. دو الگوریتم مهم در این زمینه، جستجوی عمق اول (DFS) و جستجوی عرض اول (BFS) هستند. در اینجا، ما به جستجوی عمق اول در درخت‌ها با استفاده از بازگشت می‌پردازیم.

مثال: جستجو در درخت با استفاده از بازگشت (DFS)

درخت‌ها به ساختارهایی گفته می‌شوند که هر گره (Node) می‌تواند گره‌های فرزند داشته باشد. در جستجوی عمق اول، ابتدا گره‌های پایین‌تر و عمیق‌تر درخت جستجو می‌شوند.

کد پایتون برای جستجوی عمق اول در درخت:

```
class Node:
    def __init__(self, value):
        self.value = value
        self.children = []

    def add_child(self, child_node):
        self.children.append(child_node)

def dfs(node):
    print(node.value)  # نمایش مقدار گره
    for child in node.children:
        dfs(child)  # فراخوانی بازگشتی برای فرزند گره

# ساخت یک درخت
root = Node(1)
child1 = Node(2)
child2 = Node(3)
child3 = Node(4)

root.add_child(child1)
```

```
root.add_child(child2)
child1.add_child(child3)

# جستجو در عمق درخت
dfs(root)  # خروجی: 1 2 1 3 4
```

در این مثال:

- یک کلاس `Node` تعریف کرده‌ایم که گره‌ها و فرزندانشان را نگهداری می‌کند.
- تابع `dfs` به‌صورت بازگشتی از گره ریشه شروع کرده و تمام گره‌های فرزند را به ترتیب عمق جستجو می‌کند.

2. کاربرد بازگشتی در مسائل مرتب‌سازی و جستجو

توابع بازگشتی در بسیاری از الگوریتم‌های مرتب‌سازی و جستجو نیز استفاده می‌شوند. یکی از معروف‌ترین الگوریتم‌های مرتب‌سازی بازگشتی **مرتب‌سازی سریع (Quick Sort)** است.

مثال: مرتب‌سازی سریع (Quick Sort)

الگوریتم مرتب‌سازی سریع یک الگوریتم بازگشتی است که در آن یک عنصر به‌عنوان "محور" انتخاب می‌شود و داده‌ها حول این محور تقسیم می‌شوند. سپس به‌صورت بازگشتی بخش‌های تقسیم شده مرتب می‌شوند.

کد پایتون برای مرتب‌سازی سریع (Quick Sort):

```
def quick_sort(arr):
    if len(arr) <= 1:  # شرط پایانی
        return arr
    pivot = arr[len(arr) // 2]  # انتخاب عنصر محور
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)  # بازگشت به چپ و راست

# مثال
arr = [3, 6, 8, 10, 1, 2, 1]
sorted_arr = quick_sort(arr)
print(sorted_arr)  # خروجی: [1, 1, 2, 3, 6, 8, 10]
```

در این کد:

- ابتدا با استفاده از `pivot` (عنصر محور) داده‌ها تقسیم می‌شوند.
- سپس با استفاده از بازگشت، هر دو بخش `left` و `right` مرتب می‌شوند.

مثال: جستجو دوتایی (Binary Search)

جستجو دوتایی یکی از الگوریتم‌های بازگشتی است که برای پیدا کردن یک عنصر خاص در یک لیست مرتب‌شده به‌کار می‌رود. این الگوریتم با تقسیم کردن فهرست به دو نیمه و مقایسه عنصر میانه با مقدار موردنظر شروع می‌شود.

کد پایتون برای جستجوی دوتایی:

```
def binary_search(arr, target, low, high):
    if low <= high:  # شرط پایانی
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
```

```
        elif arr[mid] < target:
            return binary_search(arr, target, mid + 1, high)
        else:
            return binary_search(arr, target, low, mid - 1)
    else:
        return -1  # عنصر پیدا نشد

# مثال
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
target = 7
result = binary_search(arr, target, 0, len(arr) - 1)
print(result)  # خروجی: 6
```

در این کد:

- جستجوی دوتایی به‌صورت بازگشتی با مقایسه میانه لیست و تقسیم آن به دو نیمه انجام می‌شود.
- وقتی که مقدار `target` پیدا شد، اندیس آن باز می‌گردد. در غیر این صورت، الگوریتم بازگشتی به جستجو در نیمه دیگر ادامه می‌دهد.

3. مزایای و چالش‌ها

- **مزایا:** استفاده از توابع بازگشتی برای حل مسائل ساختاری و مرتب‌سازی ساده‌تر و خواناتر است. بسیاری از الگوریتم‌ها مانند DFS و Quick Sort به‌طور طبیعی با بازگشت بهینه‌تر می‌شوند.
- **چالش‌ها:** الگوریتم‌های بازگشتی ممکن است منابع زیادی از حافظه مصرف کنند، به‌ویژه زمانی که عمق فراخوانی‌ها زیاد باشد. به‌علاوه، باید مراقب حلقه‌های بی‌پایان باشیم و از شرایط پایانی صحیح استفاده کنیم.

نتیجه‌گیری:

توابع بازگشتی ابزار قدرتمندی برای حل مسائل پیچیده‌ای هستند که به ساختارهایی مانند درخت‌ها، گراف‌ها و داده‌های مرتب‌سازی مرتبط هستند. این تکنیک‌ها به‌ویژه در الگوریتم‌های جستجو و مرتب‌سازی بسیار مفید هستند و به‌راحتی می‌توانند مسائل پیچیده را به مسائل ساده‌تر و مشابه تقسیم کنند.

==========

3. بهینه‌سازی توابع بازگشتی

توابع بازگشتی می‌توانند در صورتی که به درستی پیاده‌سازی نشوند، مشکلاتی مانند مصرف بالای حافظه و پیچیدگی زمانی بالا ایجاد کنند. خوشبختانه تکنیک‌های مختلفی برای بهینه‌سازی این توابع وجود دارد که می‌توانند عملکرد برنامه را به‌طور چشمگیری بهبود دهند. در اینجا دو تکنیک مهم بهینه‌سازی توابع بازگشتی معرفی می‌شوند: Memoization و Tail Recursion.

1. Memoization

Memoization یک تکنیک بهینه‌سازی است که در آن نتایج محاسبات قبلی ذخیره می‌شوند تا از محاسبات تکراری جلوگیری شود. این تکنیک به‌ویژه در مسائل بازگشتی مفید است که در آن‌ها محاسبات مشابه بارها و بارها انجام می‌شود.

روش کار Memoization:

در Memoization، از یک ساختار داده مانند دیکشنری (یا جدول هش) برای ذخیره نتایج میانه استفاده می‌کنیم. زمانی که یک نتیجه قبلاً محاسبه شده باشد، به‌جای محاسبه مجدد آن، از نتیجه ذخیره شده استفاده می‌کنیم.

مثال: محاسبه دنباله فیبوناچی با استفاده از Memoization

دنباله فیبوناچی یک مثال کلاسیک از مشکلاتی است که با استفاده از Memoization می‌توان آن را به‌طور چشمگیری بهینه کرد. در نسخه بازگشتی ساده از فیبوناچی، برای محاسبه یک مقدار، ممکن است همان محاسبات دوباره تکرار شوند که منجر به پیچیدگی زمانی نمایی می‌شود. با استفاده از Memoization، این تکرارها حذف می‌شوند.

کد پایتون با استفاده از Memoization برای دنباله فیبوناچی:

```python
def fibonacci(n, memo={}):
    if n in memo:  # چک می‌کنیم که آیا نتیجه قبلاً محاسبه شده است یا نه
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo)  # ذخیره کردن نتیجه
    return memo[n]

# مثال
print(fibonacci(10))  # خروجی: 55
```

در این کد:

- از دیکشنری `memo` برای ذخیره نتایج قبلی استفاده می‌کنیم.
- قبل از انجام هر محاسبه، چک می‌کنیم که آیا نتیجه قبلاً محاسبه شده و ذخیره شده است یا خیر.
- این کار باعث می‌شود که پیچیدگی زمانی به $O(n)$ کاهش یابد.

مزایای Memoization:

- به‌طور قابل توجهی پیچیدگی زمانی را کاهش می‌دهد.
- نتایج محاسبات قبلی را ذخیره کرده و به‌جای انجام محاسبات دوباره از آن‌ها استفاده می‌کند.

2. Tail Recursion

Tail Recursion یک نوع خاص از بازگشت است که در آن آخرین عملی که قبل از بازگشت تابع انجام می‌شود، فراخوانی مجدد همان تابع است. این نوع بازگشت به‌طور خاص قابل بهینه‌سازی توسط کامپایلر یا مفسر است.

چرا Tail Recursion مهم است؟

در توابع بازگشتی معمولی، پس از فراخوانی تابع، باید منتظر نتیجه بازگشتی بمانیم تا پس از بازگشت، نتیجه نهایی محاسبه شود. این فرآیند نیاز به ذخیره‌سازی وضعیت‌های میان‌دوره‌ای در پشته (stack) دارد که می‌تواند باعث افزایش مصرف حافظه شود.

اما در **Tail Recursion**، چون هیچ عملی پس از بازگشت تابع انجام نمی‌شود، کامپایلر می‌تواند از بهینه‌سازی **Tail Call Optimization (TCO)** استفاده کند تا از مصرف اضافی حافظه جلوگیری کند و از پشته بهینه‌تری استفاده کند.

مثال: محاسبه فاکتوریل با استفاده از Tail Recursion

کد پایتون برای فاکتوریل با استفاده از Tail Recursion:

```
def factorial_tail(n, accumulator=1):
    if n == 0:
        return accumulator
    return factorial_tail(n - 1, accumulator * n)

# مثال
print(factorial_tail(5))  # خروجی: 120
```

در این کد:

- تابع `factorial_tail` به‌صورت بازگشتی به‌گونه‌ای پیاده‌سازی شده که از یک پارامتر اضافی (`accumulator`) برای نگه‌داری نتیجه میانه استفاده می‌کند.
- هیچ عملی پس از فراخوانی مجدد تابع انجام نمی‌شود، بنابراین این تابع به‌صورت Tail Recursive است.

تفاوت Tail Recursion با سایر نوع‌های بازگشتی:

- در بازگشت‌های عادی (Non-tail recursion)، هر بار که یک تابع بازگشتی فراخوانی می‌شود، باید نتیجه بازگشتی را منتظر بماند تا به‌عنوان یک مرحله نهایی به‌کار رود. این وضعیت باعث مصرف زیاد حافظه و پشته می‌شود.
- در **Tail Recursion**، چون هیچ عملی پس از فراخوانی تابع انجام نمی‌شود، کامپایلر می‌تواند از **Tail Call Optimization** برای بهینه‌سازی پشته استفاده کند، بنابراین حافظه مصرفی بسیار کمتر خواهد بود.

محدودیت‌ها:

- پایتون از **Tail Call Optimization** به‌طور پیش‌فرض پشتیبانی نمی‌کند، بنابراین حتی با استفاده از Tail Recursion نیز ممکن است در صورت عمق زیاد بازگشت، با خطای پشته مواجه شوید.
- با این حال، در زبان‌هایی مانند Scheme یا Haskell که از TCO پشتیبانی می‌کنند، Tail Recursion می‌تواند به‌طور چشمگیری به بهینه‌سازی حافظه کمک کند.

نتیجه‌گیری:

- **Memoization** به شما کمک می‌کند که با ذخیره نتایج محاسبات قبلی، پیچیدگی زمانی توابع بازگشتی را کاهش دهید.
- **Tail Recursion** با حذف نیاز به ذخیره‌سازی وضعیت‌های میانه در پشته، به کاهش مصرف حافظه کمک می‌کند و در زبان‌هایی که از TCO پشتیبانی می‌کنند، می‌تواند به‌طور قابل توجهی بهینه‌سازی شود.
- این تکنیک‌ها باعث می‌شوند که توابع بازگشتی نه تنها از نظر زمان اجرا بلکه از نظر استفاده از منابع سیستم نیز بهینه‌تر شوند.