SDEV 1001

Programming Fundamentals

Modules and Functions - 3

A LEADING POLYTECHNIC COMMITTED TO YOUR SUCCESS

Expectations - What I expect from you

- No Late Assignments

- No Cheating

- Be a good classmate

- Don’t waste your time

- Show up to class

Agenda

On the right is what we will cover today.

- Flexible Function Arguments in Python
- Positional vs. Keyword Arguments
- Using *args for Variable-Length Arguments
- Using **kwargs for Variable-Length Keyword Arguments
- Combining *args and **kwargs
- Argument Unpacking
- Summary

Flexible Function Arguments in Python

What are flexible function arguments?

Flexible function arguments allow you to define functions that can accept a variable number of arguments. This is useful when you don't know beforehand how many arguments will be passed to the function.

For example you could create a function that could take:

- No arguments
- 2 arguments
- 4 arguments
- Any number of arguments

Why use flexible arguments?

- Handle varying input sizes.
- Make functions more reusable and generic.

Positional vs. Keyword Arguments

As we saw in the last classes there are two main types of arguments in Python:

- **Positional arguments**: Order matters, the first argument goes to the first parameter, the second to the second, and so on.
- **Keyword arguments**: Specify by name, order doesn't matter.

Example:

```python
def describe_pet(animal, name):
    print(f"{name} is a {animal}.")

describe_pet("dog", "Buddy")            # Positional
describe_pet(animal="cat", name="Milo") # Keyword
```

Here in the above example, the first call uses positional arguments, while the second uses keyword arguments.

Using *args for Variable-Length Arguments

To accept a variable number of positional arguments, you can use `*args`. This allows you to pass any number of arguments to a function.

Example:

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

print_numbers(1, 2, 3)
print_numbers(10, 20)
print_numbers(5, 15, 25, 35)
```

In the above example here `print_numbers` can take any number of arguments, making it flexible for different use cases.

- in the first call, it prints 1, 2, and 3.
- in the second call, it prints 10 and 20.
- in the third call, it prints 5, 15, 25, and 35.

Using **kwargs for Variable-Length Keyword Arguments

- `**kwargs` allows a function to accept any number of keyword arguments.

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

print_settings(theme="dark", font="Arial", size=12)
```

Here `print_settings` can take any number of keyword arguments, making it flexible for different settings.

- In the above example, it prints:
  - theme: dark
  - font: Arial
  - size: 12

WE ARE ESSENTIAL TO ALBERTA | NAIT

Combining *args and **kwargs

You can use both in the same function for maximum flexibility.

```python
def order_summary(*items, **options):
    print("Items ordered:")
    for item in items:
        print("-", item)
    print("Options:")
    for key, value in options.items():
        print(f"{key}: {value}")

order_summary("burger", "fries", drink="cola", size="large")
```

Here's the output of the above example:

```
Items ordered:
- burger
- fries
Options:
drink: cola
size: large
```

WE ARE ESSENTIAL TO ALBERTA | NAIT

Argument Unpacking

You can unpack lists and dictionaries into function arguments using `*` and `**`.

- Use `*` to unpack a list/tuple into positional arguments.
- Use `**` to unpack a dictionary into keyword arguments.

```python
def show_info(name, age):
    print(f"{name} is {age} years old.")

person = ("Alice", 30)
show_info(*person)

settings = {"name": "Bob", "age": 25}
show_info(**settings)
```

In the above example: - `show_info(*person)` unpacks the tuple into positional arguments. - `show_info(**settings)` unpacks the dictionary into keyword arguments.

This technique allows you to pass arguments dynamically, making your functions even more flexible and reusable.

Summary

- Use `*args` for variable numbers of positional arguments.
- Use `**kwargs` for variable numbers of keyword arguments.
- Argument unpacking makes your functions even more flexible and reusable.

NAIT

Example

Let’s go run a few examples together