Python

Data Structures & Functions

Alish Bista - 11/11/2025

Built-in Structures

List

Collections That Can Change

- Lists are ordered collections you can change anytime
- Think of it like a shopping cart – you can add items, remove them, rearrange them
- Use square brackets [ ] to create them
- Perfect when you need to keep things in order and might need to modify your collection later

```
# Creating a list
sales = [150, 200, 175, 300, 250]

# Adding items
sales.append(280)  # [150, 200, 175,
300, 250, 280]

# Accessing by position
first_sale = sales[0]  # 150

# Changing values
sales[1] = 210  # Updates second item
```

Tuples

The Unchangeable Record

- Tuples are like lists but immutable (fancy word for "can't be changed"): my_tuple = (1, 2, 3)
- Think of it like a birth certificate – once created, the information is locked in
- Use parentheses ( ) to create them
- Great for data that should never change, like coordinates (latitude, longitude) or RGB color values

```
# Creating a tuple
store_location = (40.7128,
-74.0060)  # latitude, longitude

# Accessing items
lat = store_location[0]  #
40.7128

# This will cause an ERROR:
# store_location[0] = 41.0  #
Can't change tuples!
```

Dictionary

Like a Dictionary

- Dictionaries store key-value pairs
- Like a real dictionary: you look up a word (key) to get its definition (value)
- Use curly braces { } with colons to create them
- Perfect when you need to link related information together or look things up quickly

```
# Creating a dictionary
customer = {
    "name": "Sarah Chen",
    "age": 28,
    "purchases": 15,
    "total_spent": 1250.50
}

# Accessing values by key
customer_name = customer["name"]  #
"Sarah Chen"

# Adding new information
customer["email"] = "sarah@email.com"
```

Set

No Duplicates Allowed

- Sets are unordered collections with no duplicates
- Like a VIP guest list: each person appears only once, and order doesn't matter
- Use curly braces { } to create them
- Ideal for removing duplicates or checking membership (is this item in my collection?)

```
# Creating a set
customer_ids = {101, 102, 103,
102, 101}
print(customer_ids)  # {101, 102,
103} - duplicates removed!


# Finding unique values from a
list
all_purchases = [101, 102, 101,
103, 102, 101]
unique_customers =
set(all_purchases)  # {101, 102,
103}
```

Functions

Functions

Reusable Code

- Reusable blocks of code that perform a specific task
- Think of them like a recipe: write it once, use it many times
- Help you avoid repeating the same code
- Take input(s), do something with them, and often give back output(s)

```
# A simple function
def greet(name):
    return f"Hello,
{name}!"

# Using the function
message = greet("Sarah")
# "Hello, Sarah!"
```

Functions

Function's Anatomy

- **def** keyword tells Python you're defining a function
- **Function name** should describe what it does (use lowercase with underscores)
- **Parameters** go in parentheses – these are the inputs your function needs
- **Return** statement sends the result back (optional but common)

```
def calculate_discount(price,
discount_percent):
    # Function body - the work
happens here
    discount_amount = price *
(discount_percent / 100)
    final_price = price -
discount_amount
    return final_price


# Using it
sale_price = calculate_discount(100,
20)  # Returns 80.0
```

Functions

Multiple Returns

- It is possible to get multiple pieces of information back from a function
- In *return* statement, separate values with commas
- The function returns them as a tuple

```
def analyze_sales(sales_list):
    total = sum(sales_list)
    average = total / len(sales_list)
    highest = max(sales_list)
    return total, average, highest

# Getting all three values back
monthly_sales = [1200, 1500, 1350, 1600]
total, avg, peak =
analyze_sales(monthly_sales)

print(f"Total: ${total}, Average: ${avg},
Peak: ${peak}")
# Total: $5650, Average: $1412.5, Peak:
$1600
```

Functions

Default Parameters

- Possible to give parameters default values so they're optional
- If the caller doesn't provide that argument, the default is used
- Put required parameters first, then optional ones with defaults

```
def calculate_commission(sales,
rate=0.10):
    return sales * rate

# Using default rate (10%)
commission1 =
calculate_commission(5000)  #
500.0

# Overriding with custom rate
commission2 =
calculate_commission(5000, 0.15)
# 750.0
```

Practice

Practice

Applying Data Structures and Functions

- Create a list with sample sales data

```
north_sales = [1200, 1500, 900, 1800, 1350]
```

- *Define a function named sales_summary that accepts two parameters: sales_data (required) and region (optional, default = “All”)*
- *Inside the function, calculate these statistics:*
  - *Total: sum(sales_data); Average: sum(sales_data) / len(sales_data); Count: len(sales_data); Minimum: min(sales_data); Maximum: max(sales_data)*
- *Return a dictionary with all the results:*

```
return {
    "region": region,
    "total_sales": # your calculation,
    "average": # your calculation,
    "count": # your calculation,
    "min": # your calculation,
    "max": # your calculation
}
```

Practice

Applying Data Structures and Functions

```
def sales_summary(sales_data, region="All"):
    return {
        "region": region,
        "total_sales": sum(sales_data),
        "average": sum(sales_data) / len(sales_data),
        "count": len(sales_data),
        "min": min(sales_data),
        "max": max(sales_data)
    }

results = sales_summary([1200, 1500, 900, 1800], "North")
print(results)
```

Q&A