# Python Cheat Sheet

A comprehensive quick reference guide for beginner programmers learning Python. This cheat sheet covers essential syntax, data types, control flow, functions, and best practices to support your programming journey. Keep this handy as you work on your projects and experiments.

## Variables & Data Types

Variables are containers that store values so they can be reused throughout your program. Think of them as labeled boxes where you keep information. In Python, you don't need to declare the type explicitly—Python figures it out automatically based on the value you assign.

### int (Integer)

Whole numbers without decimal points

```
money = 30
day = 1
score = -5
```

### float (Decimal)

Numbers with decimal points

```
price = 2.5
rate = 0.1
pi = 3.14
```

### str (String)

Text enclosed in quotes

```
name = "Alice"
item = 'coffee'
msg = "Hello!"
```

### bool (Boolean)

True or False values

```
is_open = False
active = True
done = False
```

Variables can be reassigned at any time, and their type can even change. For example, you could start with `x = 5` and later do `x = "hello"`. However, it's generally better practice to keep variable types consistent to avoid confusion.

# Expressions & Basic Math

Python provides a comprehensive set of arithmetic operators for performing mathematical calculations. These operators work with both integers and floating-point numbers, allowing you to build complex expressions for calculations in your programs.

1 **Addition (+)**

Combines two numbers

```
total = 10 + 5  # Result: 15
```

2 **Subtraction (-)**

Finds the difference

```
change = 20 - 7  # Result: 13
```

3 **Multiplication (*)**

Multiplies values together

```
total = price * quantity
```

4 **Division (/)**

Divides and returns a float

```
avg = total / count  # 10/3 = 3.333
```

5 **Floor Division (//)**

Divides and rounds down

```
pages = items // 10  # 25//10 = 2
```

6 **Remainder (%)**

Returns division remainder

```
odd = num % 2  # 7%2 = 1
```

7 **Exponent (**)**

Raises to a power

```
squared = base ** 2  # 5**2 = 25
```

## Real-World Examples

### Shopping Cart

```
total = price * quantity
```

Calculate the total cost by multiplying the unit price by the number of items purchased.

### Game Upgrade

```
boost = base * (1 + 0.1 * level)
```

Calculate stat boosts that increase by 10% per level, common in RPG games.

### Exponential Growth

```
cost = base_cost * (growth_factor ** (level - 1))
```

Calculate costs that grow exponentially, like building upgrades in strategy games.

**Pro Tip:** Use parentheses to control the order of operations. Python follows standard mathematical order (PEMDAS), but parentheses make your intentions clear: `(a + b) * c` vs a + (b * c)

# Input & Output

Interacting with users is a fundamental part of programming. Python provides simple functions for displaying information to users and collecting input from them. Understanding how these functions work is essential for creating interactive programs.

### Printing Output

The `print()` function displays information to the console. You can print variables, strings, or multiple items separated by commas.

```
print("Money:", money)
print("Welcome to the shop!")
print("Total:", total, "Gold")
```

When printing multiple items with commas, Python automatically adds spaces between them.

### Getting User Input

The `input()` function displays a prompt and waits for the user to type something and press Enter.

```
name = input("What's your name? ")
item = input("Choose item: ")
print("Hello,", name)
```

The text inside `input()` is shown to the user as a prompt. Whatever they type becomes the value of the variable.

## Converting Input Types

Here's a critical point that trips up many beginners: the `input()` function **always** returns a string, even if the user types a number. If you need to use the input as a number for calculations, you must convert it first.

**1**

### String Input

```
qty = input("How many? ")
# qty is "5" (text)
```

**2**

### Convert to Integer

```
qty = int(input("How many? "))
# qty is 5 (number)
```

**3**

### Use in Math

```
total = price * qty
# Now this works!
```

### Common Conversions

- `int(x)` — Convert to integer
- `float(x)` — Convert to decimal
- `str(x)` — Convert to string

### Example

```
age = int(input("Age? "))
price = float(input("Price? "))
code = str(123)  # "123"
```

> **Remember:** If you try to do math with a string number like `"5" * 2`, you'll get `"55"` instead of `10`! Always convert numeric input before calculations.

# If Statements

Conditional statements enable programs to make decisions and execute different code based on conditions, making your code dynamic.

## Basic Structure

01

**if condition:**

Executes if condition is True.

02

**elif other_condition:**

Checks another condition if previous were False. Multiple `elif` blocks allowed.

03

**else:**

Executes if all preceding conditions are False.

## Example Code

```
if money >= 10:
    print("You can afford it!")
elif money >= 5:
    print("Almost enough...")
else:
    print("Not enough money.")
```

Only one block executes. Python checks conditions sequentially, stopping at the first True one.

## Indentation Matters

Python uses indentation (4 spaces or 1 tab) to define code blocks.

```
# Correct
if x > 5:
    print("Big")
    print("Really big")

# Wrong - IndentationError
if x > 5:
print("Big")
```

## Comparison Operators

These operators compare two values and return True or False:

### == (Equal)

```
if name == "Alice":
```

Checks if values are equal.

### != (Not Equal)

```
if status != "closed":
```

Checks if values are different.

### > < (Greater/Less)

```
if score > 100:
if age < 18:
```

Compares numeric values.

### >= <= (Or Equal)

```
if level >= 10:
if temp <= 0:
```

Includes boundary value.

## Logical Operators

Combine multiple conditions to create more complex logic:

### and

Both conditions True.

```
if money >= 10 and level >= 5:
    print("Can buy upgrade!")
```

### or

At least one condition True.

```
if day == "Saturday" or day ==
"Sunday":
    print("Weekend!")
```

### not

Inverts boolean value.

```
if not is_closed:
 print("Shop is open!")
```

# Loops

Loops repeat code efficiently. Python offers `for` loops for iterating through sequences and `while` loops for repeating based on conditions.

## For Loop

Use `for` loops to iterate through sequences (lists, strings, `range()`) or when the number of repetitions is known.

```
items = ["coffee", "tea", "cake"]
for item in items:
    print(item)

# Loop with numbers
for i in range(5):
    print(i)  # 0, 1, 2, 3, 4
```

The loop variable takes each sequence value.

## While Loop

Use `while` loops to repeat code as long as a condition is true, especially when the number of iterations is unknown.

```
running = True
while running:
    choice = input("Continue? (y/n) ")
    if choice == "n":
        running = False

# Countdown
count = 5
while count > 0:
    print(count)
    count -= 1
```

The loop runs while its condition is True. Ensure the condition can eventually become False to prevent infinite loops.

## The range() Function

The `range()` function generates sequences of numbers, perfect for `for` loops:

### range(n)

Generates 0 to n-1

```
for i in range(3):  # 0, 1, 2
```

### range(start, stop)

Generates start to stop-1

```
for i in range(2, 5):  # 2, 3, 4
```

### range(start, stop, step)

Generates with custom increment

```
for i in range(0, 10, 2):  # 0, 2, 4, 6, 8
```

## Loop Control Keywords

### break

Exits loop immediately

```
for item in items:
    if item == "exit":
        break
    print(item)
```

### continue

Skips current iteration, moves to next

```
for num in range(10):
    if num % 2 == 0:
        continue
    print(num)  # Odd numbers only
```

**Warning:** Be careful with `while` loops! If the condition never becomes False, you'll create an infinite loop. Always ensure there's a way for the loop to end. Use **Ctrl+C** to stop a runaway loop.

# Lists

Lists are ordered, mutable collections of items, used to group and manipulate related data in Python.

## Creating and Using Lists

### Basic List Operations

```
items = ["coffee", "tea", "cake"]
empty = []
mixed = [1, "two", 3.0, True]
```

Lists can contain any type of data, including mixed types.

### Accessing Elements

```
first = items[0]     # "coffee"
last = items[-1]     # "cake"
second = items[1]    # "tea"
```

Python uses zero-based indexing—the first item is at index 0. Negative indices count from the end.

## Common List Methods

1

### append(item)

Adds an item to the end.

```
items.append("sandwich")
# ["coffee", "tea", "cake", "sandwich"]
```

2

### insert(index, item)

Inserts an item at a specific position.

```
items.insert(1, "juice")
# ["coffee", "juice", "tea", "cake"]
```

3

### remove(item)

Removes the first occurrence of an item.

```
items.remove("tea")
# ["coffee", "cake", "sandwich"]
```

4

### pop(index)

Removes and returns item at index (default: last).

```
last = items.pop()
# Returns "sandwich", list is now shorter
```

5

### len(list)

Gets the number of items in the list.

```
count = len(items)
# Returns 3
```

## List Slicing

Extract portions of a list using slice notation:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
first_three = numbers[0:3]    # [0, 1, 2]
middle = numbers[3:7]         # [3, 4, 5, 6]
last_two = numbers[-2:]       # [8, 9]
every_other = numbers[::2]    # [0, 2, 4, 6, 8]
```

## Looping Through Lists

### By Item (Recommended)

```
for item in items:
    print(item)
```

Use for values only.

### By Index

```
for i in range(len(items)):
    print(i, items[i])
```

Use for position and value.

**Pro Tip:** Lists are perfect for storing inventory items, player scores, menu options, or any collection where order matters and you might add or remove elements.

# Dictionaries

Dictionaries store data as key-value pairs, allowing you to look up values using descriptive keys instead of numeric indices. They're ideal for structured data like user profiles or configuration settings.

## Creating and Accessing Dictionaries

### Create

```
inventory = {
    "coffee": 10,
    "tea": 5,
    "cake": 3
}
```

Keys and values are separated by colons.

### Access Values

```
coffee_qty = inventory["coffee"] # 10
sugar = inventory.get("sugar", 0) # 0
```

Use brackets or get() for safe access.

### Modify Values

```
inventory["coffee"] = 15
inventory["muffin"] = 8
```

Assignment creates or updates keys.

### Remove Items

```
del inventory["cake"]
removed = inventory.pop("tea")
```

del removes; pop() removes and returns.

## Common Dictionary Methods

### Keys & Values

```
keys = inventory.keys()
# dict_keys(['coffee', 'tea'])

values = inventory.values()
# dict_values([10, 5])

items = inventory.items()
# Key-value pairs
```

### Check Membership

```
if "coffee" in inventory:
    print("We have coffee!")

if "pizza" not in inventory:
    print("Out of pizza!")
```

### Length

```
num_items = len(inventory)
# Returns number of keys

is_empty = len(inventory) == 0
```

# Dictionaries (Part 2)

## Looping Through Dictionaries

### Keys

```
for item in inventory:
    print(item)
```

### Values

```
for qty in
inventory.values():
    print(qty)
```

### Both (Best)

```
for item, qty in
inventory.items():
    print(f"{item}: {qty}")
```

## Nested Dictionaries

Dictionaries can contain other dictionaries:

```
shop_data = {
    "inventory": {"coffee": 10},
    "prices": {"coffee": 3.50}
}
coffee_price = shop_data["prices"]["coffee"]  # Access nested values
```

Access nested values as shown in the example.

**Remember:** Dictionary keys must be immutable (strings, numbers, tuples), but values can be anything.

# Functions

Functions are reusable blocks of code that perform specific tasks, helping organize programs, avoid repetition, and improve maintainability.

## Function Basics

### Defining Functions

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

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

def get_status():
    return "Active"
```

Define with `def`, name, parameters, and colon. Indent body.

### Calling Functions

```
greet("Alice")
# Prints: Hello, Alice!

result = add(3, 4)
# result is 7

status = get_status()
# status is "Active"
```

Call by name with arguments. Capture returns with assignment.

## Parameters vs Arguments

### Parameters

Variables in function definition; placeholders for needed data.

```
def calculate_price(quantity, unit_price):
```

`quantity` and `unit_price` are **parameters**.

### Arguments

Actual values passed when calling the function.

```
total = calculate_price(5, 3.50)
```

`5` and `3.50` are **arguments**.

## Return Values

The `return` statement sends a value back and exits. Functions without it return `None`.

01

### Single Return

```
def square(x):
    return x ** 2
```

Calculates and returns a single value.

02

### Multiple Returns

```
def min_max(numbers):
    return min(numbers),
max(numbers)

low, high = min_max([3, 7, 2, 9])
```

Returns multiple values (tuple) for unpacking.

03

### Early Return

```
def divide(a, b):
    if b == 0:
        return None
    return a / b
```

Exits function early based on conditions.

# Functions (Part 2)

## Function Design Guidelines

**One Purpose** - Each function should do one thing well

**Use Parameters** - Pass values via parameters, avoid hardcoding

**Prefer Return** - Return values, don't just print them, for reusability

**Descriptive Names** - Use verbs describing action: calculate_total(), get_user_input()

## Default Parameters

Give parameters default values to make them optional:

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

```
greet("Alice")  # Uses default: "Hello, Alice!"
greet("Bob", "Hey")  # Custom greeting: "Hey, Bob!"
```

# Basics of Classes

Classes are blueprints for creating objects that bundle related data and behavior. They are the foundation of object-oriented programming (OOP), helping structure complex programs by organizing data and functions that operate on that data.

## Understanding Classes

### What Are Classes?

A class is a blueprint (like a cookie cutter) defining properties (data) and methods (functions) for objects. Each object created from the class is an independent instance with its own data.

```
class Shop:
    def __init__(self):
        self.money = 0
        self.inventory = {}

    def show_money(self):
        print(f"Money: ${self.money}")

    def add_item(self, item, qty):
        self.inventory[item] = qty
```

### Creating and Using Objects

Create multiple objects (instances) from a class. Each instance holds its own separate data.

```
shop1 = Shop()
shop1.money = 100
shop1.add_item("coffee", 10)

shop2 = Shop()
shop2.money = 50
shop2.add_item("tea", 5)

shop1.show_money()  # Money: $100
shop2.show_money()  # Money: $50
```

## Key Concepts

### __init__ Method

The **constructor** (__init__) runs automatically when an object is created, setting initial attribute values.

```
def __init__(self, starting_money):
    self.money = starting_money
    self.level = 1

shop = Shop(100)  # Passes 100 to __init__
```

### self Parameter

Every method must have self as its first parameter. It refers to the specific object instance, allowing methods to access its data.

```
def add_money(self, amount):
    self.money += amount  # Modifies THIS object's money
```

### Instance Variables

Variables belonging to a specific object, defined with `self.variable_name`. Each instance has its own copy.

```
self.money = 0
self.name = "Main Shop"
self.inventory = []
```

### Methods

Functions that belong to a class. They operate on the object's data and are called via `object.method()`.

```
def calculate_total(self):
    return sum(self.inventory.values())

total = shop.calculate_total()
```

**Why Use Classes?** Classes organize related data and functions, making complex programs easier to understand and maintain.

## Complete Example: Player Class

```
class Player:
    def __init__(self, name):
        self.name = name
        self.health = 100
        self.level = 1
        self.inventory = []

    def take_damage(self, amount):
        self.health -= amount
        if self.health < 0:
            self.health = 0

    def heal(self, amount):
        self.health += amount
        if self.health > 100:
            self.health = 100

    def level_up(self):
        self.level += 1
        self.health = 100
        print(f"{self.name} reached level {self.level}!")

    def add_item(self, item):
        self.inventory.append(item)

# Usage
player = Player("Hero")
player.take_damage(30)
player.add_item("sword")
player.level_up()
```

# JSON Save & Load

JSON is a lightweight text format for storing and exchanging data. Python's `json` module simplifies saving and loading program states, ideal for game saves, user preferences, and application settings.

## Why Use JSON?

### Human-Readable

Plain text, easy to read and debug. Can be edited manually if needed.

### Universal Format

Works across different programming languages (Python, JavaScript, etc.) and platforms.

### Perfect for Python

Maps naturally to Python types: dicts to objects, lists to arrays, basic types directly.

## Saving Data (Writing JSON)

### Basic Save Operation

```
import json

game_data = {
    "player": "Alice",
    "level": 5
}

with open("savegame.json", "w") as f:
    json.dump(game_data, f)
```

`json.dump()` writes data to a file.

### Pretty Printing

```
with open("savegame.json", "w") as f:
    json.dump(game_data, f, indent=2)

# Output:
# {
#   "player": "Alice",
#   "level": 5
# }
```

Add `indent=2` for readable formatting.

## Loading Data (Reading JSON)

### Open File

```
with open("savegame.json")
as f:
```

Open in read mode (default).

### Parse JSON

```
    data = json.load(f)
```

`json.load()` converts JSON to Python objects.

### Use the Data

```
player_name =
data["player"]
current_level = data["level"]
```

Access like a Python dict or list.

## Complete Example: Game Save System

```
import json

def save_game(player_data, filename="savegame.json"):
    with open(filename, "w") as f:
        json.dump(player_data, f, indent=2)
    print("Game saved!")

def load_game(filename="savegame.json"):
    try:
        with open(filename) as f:
            return json.load(f)
    except FileNotFoundError:
        print("No save file found. Starting new game.")
        return None

player = {
    "name": "Hero",
    "health": 85,
    "level": 7,
    "gold": 450
}

save_game(player)
loaded_player = load_game()
if loaded_player:
    print(f"Welcome back, {loaded_player['name']}!")
    print(f"Level: {loaded_player['level']}")
```

## What Can JSON Store?

| Python Type | JSON Type | Example |
|---|---|---|
| dict | object | `{"key": "value"}` |
| list, tuple | array | `[1, 2, 3]` |
| str | string | `"hello"` |
| int, float | number | `42`, `3.14` |
| True, False | true, false | `true` |
| None | null | `null` |

**Important:** JSON cannot directly store custom objects, functions, or classes. Convert these to dictionaries before saving, and reconstruct them after loading.

# Error Handling

Errors (also called exceptions) are inevitable in programming—users enter invalid data, files go missing, networks fail, and unexpected situations arise. Rather than letting your program crash, you can use error handling to catch problems gracefully and respond appropriately. This makes your programs more robust and user-friendly.

## The Try-Except Pattern

### Basic Structure

```
try:
    # Code that might cause an error
    risky_operation()
except ErrorType:
    # What to do if that error occurs
    handle_error()
```

Python tries to execute the code in the `try` block. If an error occurs, it immediately jumps to the matching `except` block instead of crashing.

### Common Example

```
try:
    qty = int(input("How many? "))
    print(f"You want {qty} items")
except ValueError:
    print("Please enter a number.")
    qty = 1  # Use default value
```

If the user types "five" instead of "5", the `int()` conversion fails. The except block catches the error and handles it gracefully.

## Common Exception Types

### ValueError

Invalid value for a function, like converting non-numeric text to int.

```
try:
    age = int("twenty")
except ValueError:
    print("Not a valid number!")
```

### KeyError

Accessing a dictionary key that doesn't exist.

```
try:
    price = prices["pizza"]
except KeyError:
    print("Item not found!")
    price = 0
```

### FileNotFoundError

Trying to open a file that doesn't exist.

```
try:
    with open("data.txt") as f:
        content = f.read()
except FileNotFoundError:
    print("File not found!")
```

### ZeroDivisionError

Dividing by zero in mathematical operations.

```
try:
    result = total / count
except ZeroDivisionError:
    print("Can't divide by zero!")
    result = 0
```

# Error Handling (Part 2)

## Advanced Error Handling

### 1

#### Multiple Except Blocks

Handle different errors differently based on what went wrong.

```
try:
    # Some risky code
    x = 10 / int(input("Enter a number: "))
    my_list = [1, 2]
    print(my_list[x])
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
except ValueError:
    print("Error: Invalid input, please enter a number.")
except IndexError:
    print("Error: Index out of range.")
```

### 2

#### Catch Multiple Errors

Handle multiple error types the same way using a tuple.

```
try:
    num = int("abc") # ValueError
    # num = 10 / 0 # ZeroDivisionError
except (ValueError, ZeroDivisionError) as e:
    print(f"An input or calculation error occurred: {e}")
```

### 3

#### Catch All Errors

`Exception` catches most errors, stored in `e`.

```
try:
    result = some_function_that_might_fail()
except Exception as e:
    print(f"An unexpected error occurred: {e}")
    # Log the error for debugging
    # Optionally, re-raise the exception if it cannot be
handled here: raise
```

### 4

#### Else and Finally

`else` runs if no errors, `finally` always runs.

```
try:
    file = open("data.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found!")
else:
    print("File read successfully.")
    # Process content
finally:
    if 'file' in locals() and not file.closed:
        file.close()
    print("Cleanup complete.")
```

## Best Practices

→ **Be Specific**

Catch specific errors rather than all errors when possible. This helps you write more targeted error handling logic and prevents you from accidentally catching errors you don't expect.

→ **Don't Hide Errors Silently**

Always do something in an `except` block. Print an error message, log the error, notify the user, or take corrective action. An empty `except` block can make debugging very difficult.

→ **Use for User Input**

Wrap conversions or operations involving user input in `try-except` blocks. Users are unpredictable, and their input is a common source of runtime errors.

→ **Combine with Loops**

Use `try-except` inside loops for validation. This allows a program to gracefully handle invalid entries without crashing, prompting the user to try again until valid input is received.

## Practical Example: Robust User Input

This function keeps asking until valid input is provided.

```
def get_positive_integer(prompt):
    while True:
        try:
            value = int(input(prompt))
            if value <= 0:
                print("Please enter a positive number.")
            else:
                return value
        except ValueError:
            print("Invalid input. Please enter a whole number.")

# Usage
age = get_positive_integer("Enter your age: ")
print(f"Your age is: {age}")
```

**Remember:** Error handling isn't about preventing errors—it's about controlling what happens when they inevitably occur.

# Good Program Design

1

### Keep Related Code Together

Group related functions, variables, and classes in the same place. If you're building a shop system, put all inventory functions near each other, all money-related functions together, and so on.

```
# Good: Related functions grouped
def add_inventory(item, qty):
    inventory[item] = qty

def remove_inventory(item):
    del inventory[item]

def show_inventory():
    for item, qty in inventory.items():
        print(f"{item}: {qty}")
```

This organization makes code easier to find and understand. Consider using classes to bundle related data and functions together.

2

### Avoid Long Functions

If a function is longer than one screen (about 30-50 lines), it probably does too much. Break it into smaller, focused functions that each do one thing well.

```
# Bad: One giant function doing everything
def run_shop():
    # 100 lines of code here...

# Good: Broken into logical pieces
def display_menu():
    # Show options

def get_user_choice():
    # Get and validate input

def process_purchase():
    # Handle buying

def run_shop():
    display_menu()
    choice = get_user_choice()
    if choice == "buy":
        process_purchase()
```

Smaller functions are easier to test, debug, and reuse in different contexts.

3

### Use Clear, Descriptive Names

Variables and functions should explain their purpose without needing comments. Someone reading your code should understand what each element does from its name alone.

```
# Bad: Unclear names
def calc(x, y):
    return x * y + 10
a = calc(5, 3)

# Good: Self-explanatory names
def calculate_total_price(quantity, unit_price):
    base_cost = quantity * unit_price
    shipping_fee = 10
    return base_cost + shipping_fee

total_cost = calculate_total_price(5, 3)
```

Use `snake_case` for variables and functions, `PascalCase` for classes. Avoid single-letter names except in short loops.

# Good Program Design (Part 2)

1

### Keep Related Code Together

Group related functions, variables, and classes in the same place. If you're building a shop system, put all inventory functions near each other, all money-related functions together, and so on.

```
# Good: Related functions grouped
def add_inventory(item, qty):
    inventory[item] = qty

def remove_inventory(item):
    del inventory[item]

def show_inventory():
    for item, qty in inventory.items():
        print(f"{item}: {qty}")
```

This organization makes code easier to find and understand. Consider using classes to bundle related data and functions together.

2

### Avoid Long Functions

If a function is longer than one screen (about 30-50 lines), it probably does too much. Break it into smaller, focused functions that each do one thing well.

```
# Bad: One giant function doing everything
def run_shop():
    # 100 lines of code here...

# Good: Broken into logical pieces
def display_menu():
    # Show options

def get_user_choice():
    # Get and validate input

def process_purchase():
    # Handle buying

def run_shop():
    display_menu()
    choice = get_user_choice()
    if choice == "buy":
        process_purchase()
```

Smaller functions are easier to test, debug, and reuse in different contexts.

3

### Use Clear, Descriptive Names

Variables and functions should explain their purpose without needing comments. Someone reading your code should understand what each element does from its name alone.

```
# Bad: Unclear names
def calc(x, y):
    return x * y + 10
a = calc(5, 3)

# Good: Self-explanatory names
def calculate_total_price(quantity, unit_price):
    base_cost = quantity * unit_price
    shipping_fee = 10
    return base_cost + shipping_fee

total_cost = calculate_total_price(5, 3)
```

Use `snake_case` for variables and functions, `PascalCase` for classes. Avoid single-letter names except in short loops.

# Good Program Design (Part 3)

1

### Don't Repeat Yourself (DRY)

If copying code, create a function instead. This makes your code more concise, easier to read, and less prone to errors.

Bad example - Repeated player calculations:

```
player1_score = 100
player1_bonus = 10
player1_adjusted = player1_score + player1_bonus

player2_score = 150
player2_bonus = 5
player2_adjusted = player2_score + player2_bonus

player3_score = 80
player3_bonus = 12
player3_adjusted = player3_score + player3_bonus
```

Good example - Reusable function:

```
def calculate_adjusted_value(score, bonus):
    return score + bonus

player1_adjusted = calculate_adjusted_value(100, 10)
player2_adjusted = calculate_adjusted_value(150, 5)
player3_adjusted = calculate_adjusted_value(80, 12)
```

2

### Test Small Parts Separately

Write and test small functions immediately before building larger programs. This helps catch bugs early and ensures each component works as expected.

```
def calculate_discount(price, percentage):
    return price * (1 - percentage / 100)

# Test cases
assert calculate_discount(100, 10) == 90
assert calculate_discount(50, 50) == 25
assert calculate_discount(200, 0) == 200
print("All discount tests passed!")
```

# Good Program Design (Part 4)

This example demonstrates how code organization can significantly improve readability, maintainability, and reusability.

### Poor Organization

Everything mixed, uses globals, poor naming.

```
inventory = {"apple": 5, "banana": 10}
price_list = {"apple": 1.0, "banana": 0.5}

def buy_item(item, qty):
    global inventory
    if item in inventory and inventory[item] >= qty:
        inventory[item] -= qty
        total = qty * price_list[item]
        print(f"Bought {qty} {item}(s) for ${total:.2f}")
    else:
        print(f"Not enough {item} in stock.")

buy_item("apple", 2)
```

### Better Organization

Uses a Shop class, clear methods, no globals, descriptive names.

```
class Shop:
    def __init__(self):
        self.inventory = {"apple": 5, "banana": 10}
        self.price_list = {"apple": 1.0, "banana": 0.5}

    def process_purchase(self, item_name, quantity):
        if item_name in self.inventory and self.inventory[item_name] >= quantity:
            self.inventory[item_name] -= quantity
            total_cost = quantity * self.price_list[item_name]
            print(f"Purchased {quantity} {item_name}(s) for ${total_cost:.2f}")
        else:
            print(f"Insufficient {item_name} in stock.")

my_shop = Shop()
my_shop.process_purchase("apple", 2)
```

> "Think Like a Writer: Just as good writers revise their prose, good programmers refactor their code. Your first version doesn't have to be perfect—write it to work, then improve it to be clear."

# Debugging Tips

Debugging is the art of finding and fixing errors in your code. Every programmer, from beginners to experts, spends significant time debugging. The key difference is that experienced programmers have systematic strategies for tracking down problems. These techniques will help you debug more efficiently and learn from your mistakes.

### Print Intermediate Values

When your code produces wrong results, use print statements to see what's happening at each step. This is the simplest and most effective debugging technique.

```
def calculate_total(prices):
    total = 0
    for price in prices:
        print(f"Adding {price}, total is now {total}")  # Debug print
        total += price
    print(f"Final total: {total}")  # Debug print
    return total
```

Print variable values before and after critical operations. Once you find the bug, remove the debug prints or comment them out.

### Check Indentation

Python is extremely sensitive to indentation—tabs and spaces matter! If code seems correct but doesn't work, check that all lines in a block are indented consistently.

```
# Wrong - inconsistent indentation
if x > 5:
    print("Big")
  print("Also big")  # Too few spaces

# Right - consistent indentation
if x > 5:
    print("Big")
    print("Also big")
```

Configure your editor to show whitespace characters or use 4 spaces consistently. Mixing tabs and spaces causes hard-to-spot errors.

### Verify Data Types

Many bugs occur when variables have unexpected types. Use `type()` to check what type a variable actually is.

```
qty = input("How many? ")
print(f"qty is: {qty}, type: {type(qty)}")  #

if qty > 5:  # Error! Can't compare string to number
    print("Many")

# Fix: Convert to int first
qty = int(input("How many? "))
print(f"qty is: {qty}, type: {type(qty)}")  #
```

Remember: `input()` always returns strings, `/` always returns floats, list indices must be integers.

# Debugging Tips (Part 1.5)

Continue with debugging tips 4-5:

### Verify Spelling

Python is case-sensitive and won't warn you about typos in variable names—it just creates a new variable. Double-check that variable names are spelled exactly the same everywhere.

```
player_score = 100
player_scoer = 50  # Typo! Creates new variable

print(player_score)  # 100 - not what you wanted!
```

Common mistakes:

- `myList` vs `mylist`
- `userName` vs `username`
- `getData` vs `getdata`

Use consistent naming conventions and enable your editor's autocomplete to avoid typos.

### Test One Function at a Time

When building complex programs, test each function in isolation before connecting everything. This helps you identify exactly where problems occur.

```
# Test a function alone
def calculate_discount(price, percent):
    return price * (1 - percent / 100)

# Test with known values
print(calculate_discount(100, 20))  # Should be 80
print(calculate_discount(50, 50))   # Should be 25
print(calculate_discount(200, 0))   # Should be 200

# Only after tests pass, integrate into larger program
```

Write simple test cases with known correct answers. This is called "unit testing."

# Debugging Tips (Part 2)

Understanding common error messages and employing advanced debugging techniques are crucial for efficient troubleshooting. This section details frequent Python errors and strategies to effectively locate and resolve them.

## Common Error Messages & Solutions

| Error Message | Solution |
|---|---|
| SyntaxError | broke grammar rules → check for missing colons, quotes, parentheses |
| NameError | variable doesn't exist → check spelling or initialize first |
| TypeError | wrong type for operation → check types, convert if needed |
| IndexError | list index out of range → check length, indices start at 0 |
| KeyError | dictionary key doesn't exist → use 'in' to check or .get() |
| IndentationError | inconsistent spaces/tabs → use 4 spaces consistently |

## Advanced Debugging Techniques

### Use Python Debugger (pdb)

Use `set_trace()` to pause program execution and inspect variables, step through code, and examine the call stack. This provides a powerful way to understand program flow at critical points.

### Binary Search Debugging

When a bug is difficult to pinpoint, add print statements roughly halfway through your code. If the bug occurs before the print, move the print statement earlier; if after, move it later. Repeat to quickly narrow down the problematic section.

### Rubber Duck Debugging

Explain your code line by line to an inanimate object (like a rubber duck) or a colleague. The act of articulating your logic often helps you spot flaws, assumptions, or misunderstandings in your own code.

> "Bugs are learning opportunities. Every error teaches you something about how Python works."

**Pro Tip:** Before asking for help, create a minimal example that reproduces the bug. This forces you to isolate the problem and provides a clear, concise issue for others to understand.