



Introduction to Python for Games

Year 9 Engineering: Lets Make Games!

Learning Intention:

Learn to tell stories with code!

Success Criteria:

- display text to players
- get input from players
- remember information (variables)



```
mirror_mod = modifier_obj
# Set mirror object to mirror
mirror_mod.mirror_object = selected_obj

operation = "MIRROR_X":
mirror_mod.use_x = True
mirror_mod.use_y = False
mirror_mod.use_z = False

operation == "MIRROR_Y":
mirror_mod.use_x = False
mirror_mod.use_y = True
mirror_mod.use_z = False

operation == "MIRROR_Z":
mirror_mod.use_x = False
mirror_mod.use_y = False
mirror_mod.use_z = True

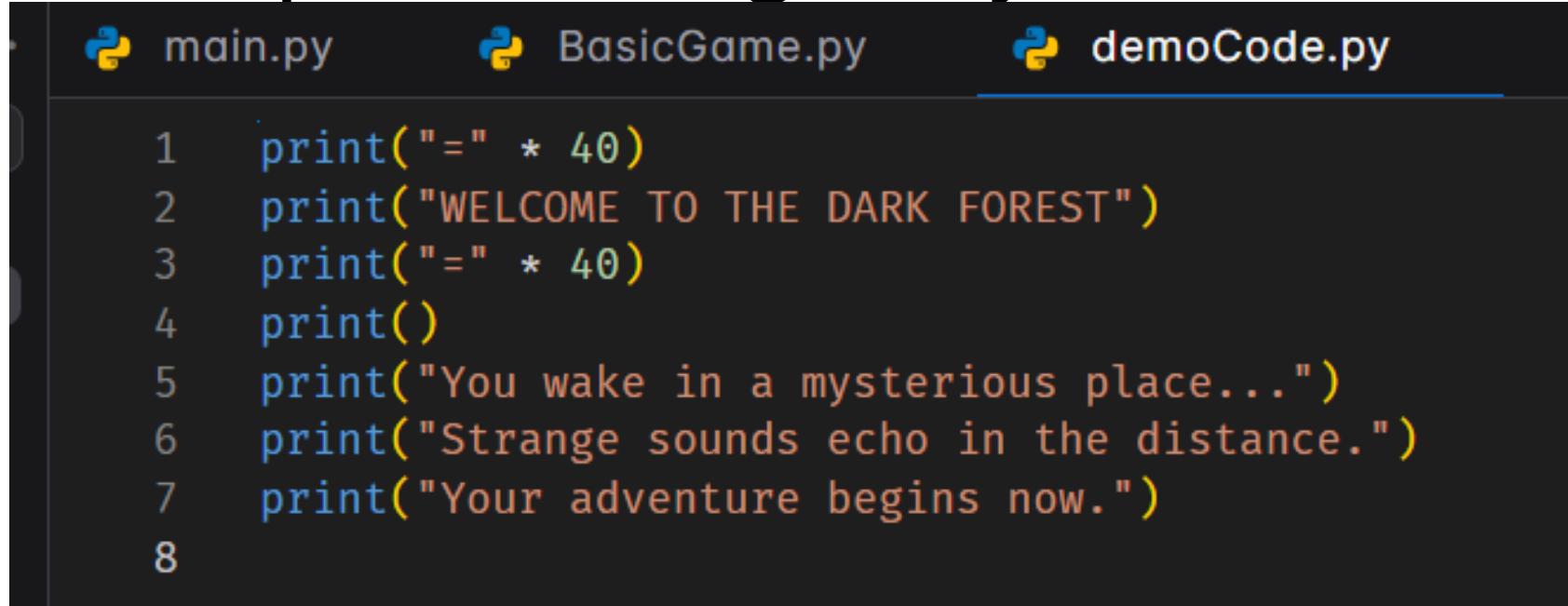
# Selection at the end - add
if _ob.select= 1
    mirr_ob.select=1
context.scene.objects.active = eval("Selected" + str(modifier))
    mirror_ob.select = 0
    bpy.context.selected_objects.append(data.objects[one.name].select)
    print("please select exactly one object")
else:
    print("please select exactly one object")

- OPERATOR CLASSES ----

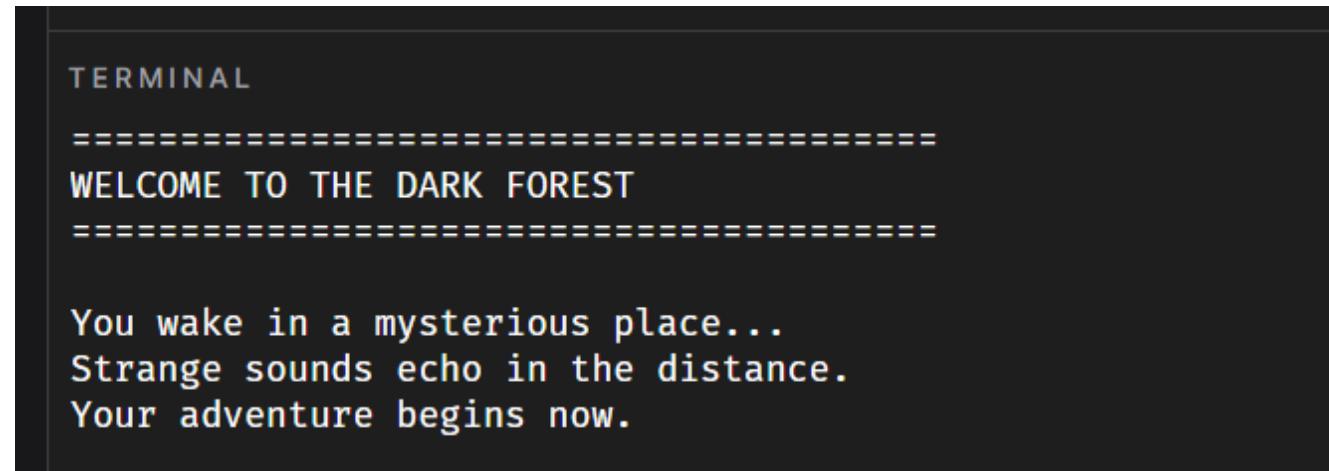
class MIRROR_OT_Mirror(bpy.types.Operator):
    bl_idname = "object.mirror"
    bl_label = "X mirror to the selected object.mirror_mirrror_x"
    bl_options = {'REGISTER', 'UNDO'}
    bl_description = "Mirror X"

    def execute(self, context):
        if context.object:
            if context.active_object is not None:
                if context.active_object.type == "MESH":
```

Example: Printing story elements on screen



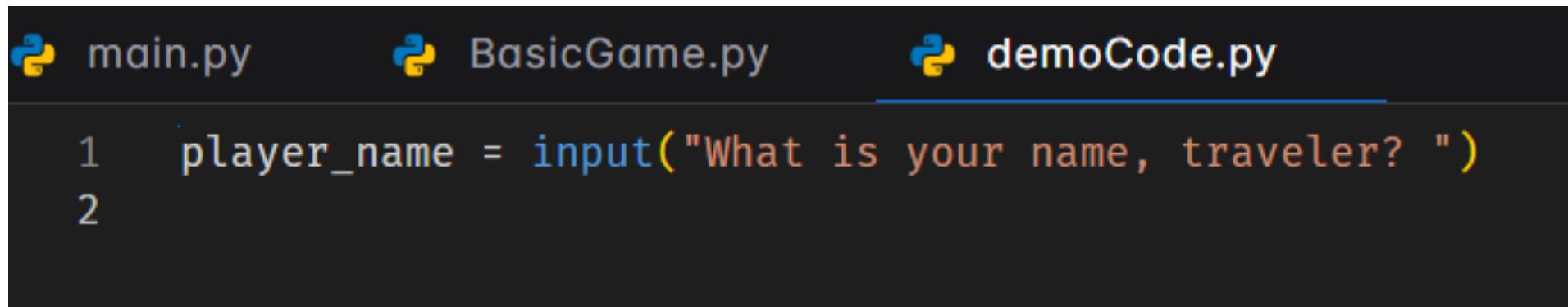
```
main.py          BasicGame.py          demoCode.py
1   print("=" * 40)
2   print("WELCOME TO THE DARK FOREST")
3   print("=" * 40)
4   print()
5   print("You wake in a mysterious place...")
6   print("Strange sounds echo in the distance.")
7   print("Your adventure begins now.")
8
```



```
TERMINAL
=====
WELCOME TO THE DARK FOREST
=====

You wake in a mysterious place...
Strange sounds echo in the distance.
Your adventure begins now.
```

Example: Getting user input



```
main.py      BasicGame.py      demoCode.py
1 player_name = input("What is your name, traveler? ")
2
```

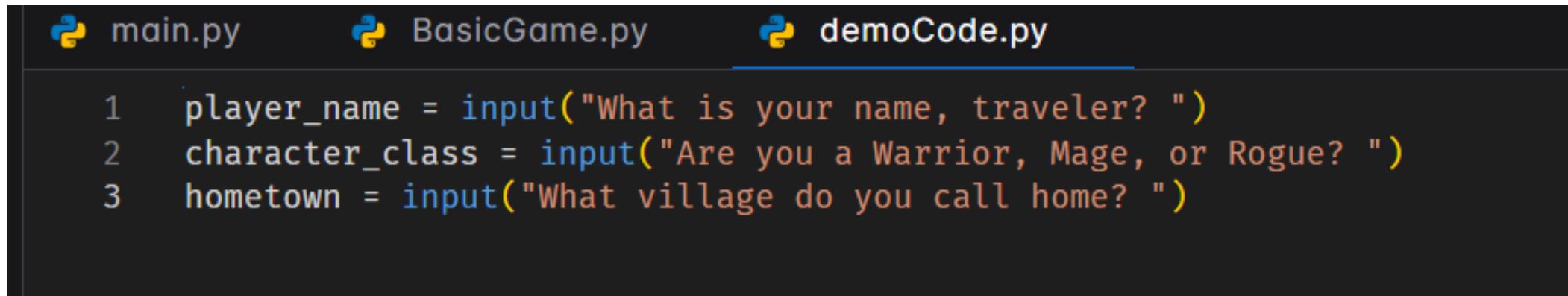
1. Python displays the question
2. Waits for the player to type
3. Stores their answer in the **variable** ‘name’

Variables

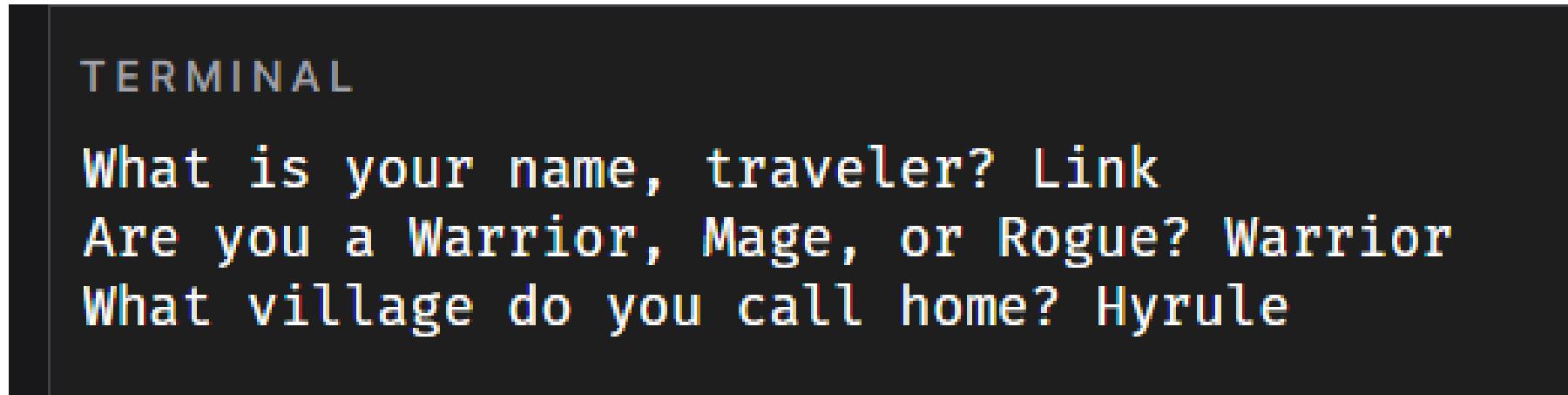
Think of variables as labeled boxes:

- The box has a name (player_name)
- You can put information inside it
- You can look at what's inside anytime

Example: Getting more user input

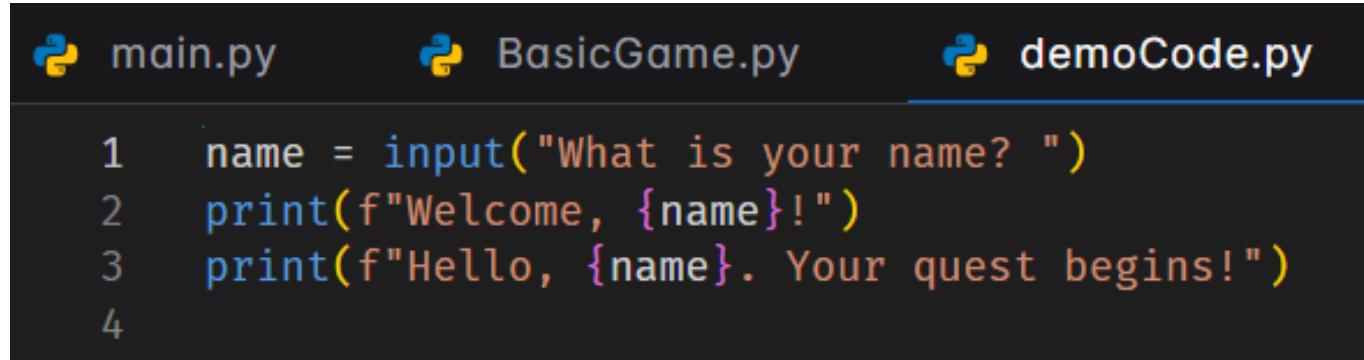


```
main.py          BasicGame.py          demoCode.py
1 player_name = input("What is your name, traveler? ")
2 character_class = input("Are you a Warrior, Mage, or Rogue? ")
3 hometown = input("What village do you call home? ")
```

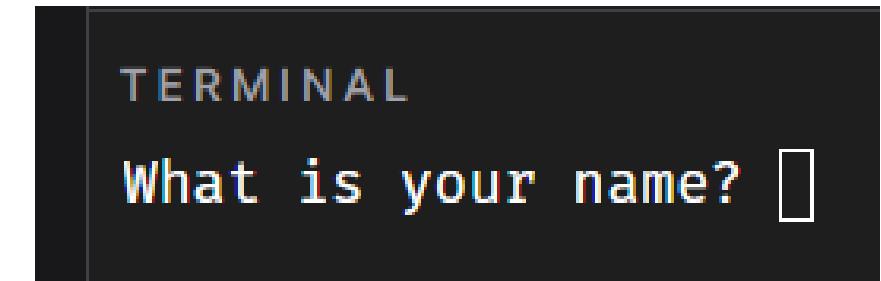


```
TERMINAL
What is your name, traveler? Link
Are you a Warrior, Mage, or Rogue? Warrior
What village do you call home? Hyrule
```

Example: Displaying user input



```
main.py          BasicGame.py          demoCode.py
1 name = input("What is your name? ")
2 print(f"Welcome, {name}!")
3 print(f"Hello, {name}. Your quest begins!")
4
```



TERMINAL
What is your name? □

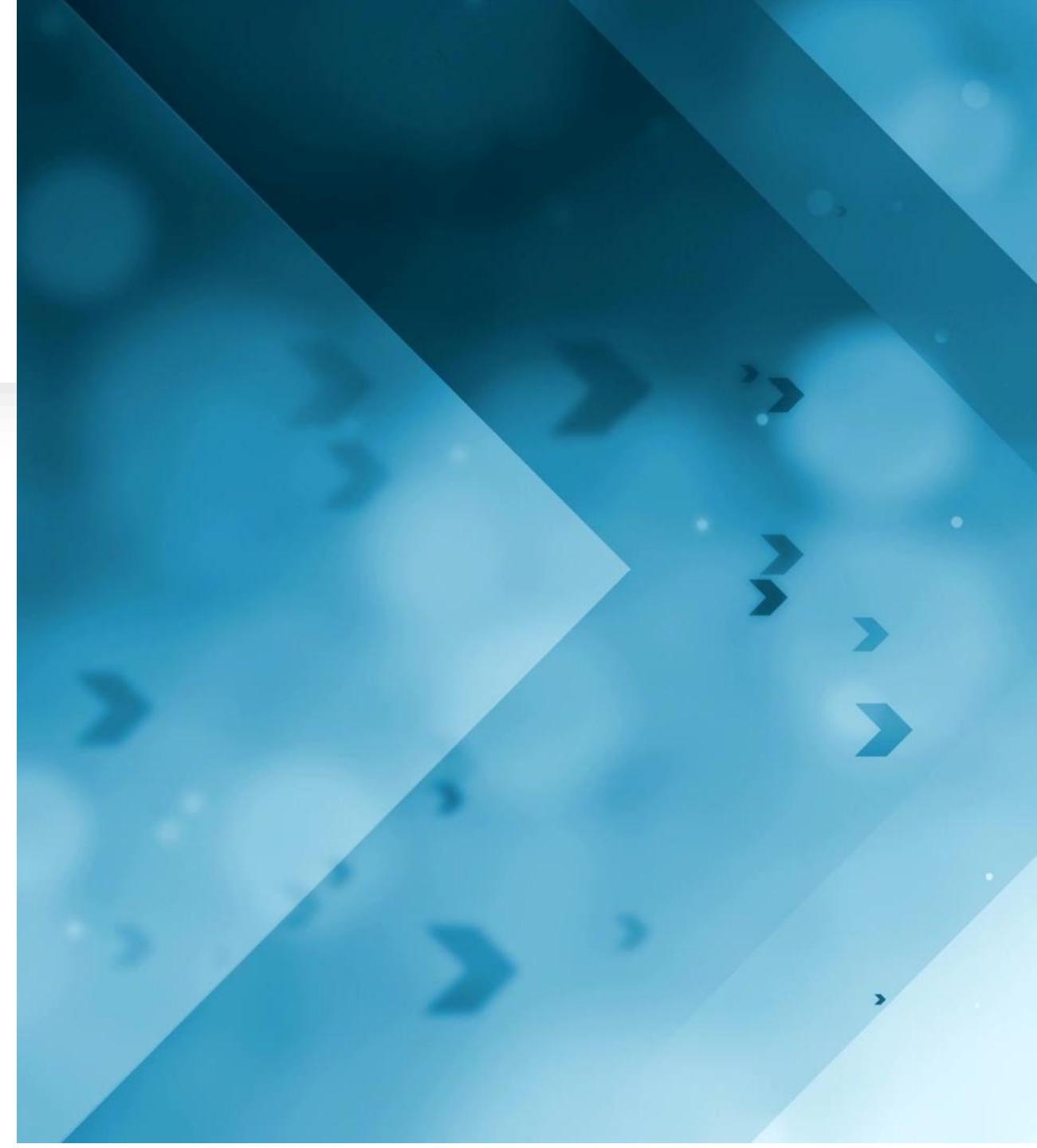
The **f** before the quotes:

- Means "format this string"
- Let's you put variables inside {curly braces}
- The variable's value appears in the text!

Challenge 1: Create Your Character

Your code should:

- Display a title for your adventure
- Print an opening scene (3+ lines)
- Ask for: name, character class, hometown
- Display a personalized welcome message



A close-up photograph of a person's hand pointing their index finger towards a computer monitor. The monitor displays a block of Python code. The code is part of a Blender operator class, specifically for a 'MIRROR' type operator. It includes logic for setting up mirror modifiers, selecting objects, and handling user input for selecting exactly one object. The background is dark, making the blue-tinted code stand out.

```
mirror_mod = modifier_obj
        mirror_mod.mirror_object = selected_object
        mirror_mod.mirror_object_to_mirror = True
    operation == "MIRROR_X":
        mirror_mod.use_x = True
        mirror_mod.use_y = False
        mirror_mod.use_z = False
    operation == "MIRROR_Y":
        mirror_mod.use_x = False
        mirror_mod.use_y = True
        mirror_mod.use_z = False
    operation == "MIRROR_Z":
        mirror_mod.use_x = False
        mirror_mod.use_y = False
        mirror_mod.use_z = True

    selection at the end -add
    _ob.select= 1
    mirr_ob.select=1
    context.scene.objects.active = eval(
        ("Selected" + str(modifier)))
    mirror_ob.select = 0
    bpy.context.selected_objects.append(data.objects[one.name].select)
    print("please select exactly one object")
    return {"FINISHED"}  
- OPERATOR CLASSES ---  
  
    types.Operator):
        X mirror to the selected object.mirror_mirror_x"
        "mirror X"
    context):
        context.active_object is not None
```

Learning Intention:

Making Choices Using if/elif/else to Branch Your Story

Success Criteria:

- I can use if statements
- I can use elif for multiple options
- I can use else for defaults
- Different choices lead to different outcome

Comparing Values

- `==` means "is equal to" (NOT `=`)
- `!=` means "is not equal to"
- `>` greater than
- `<` less than

If Statements - The Basics

Key point: Code inside the if MUST be indented!

 main.py

 BasicGame.py

```
1  if condition:  
2      # Code here runs if condition is True  
3      print("This happens!")
```

If Statements - The Basics

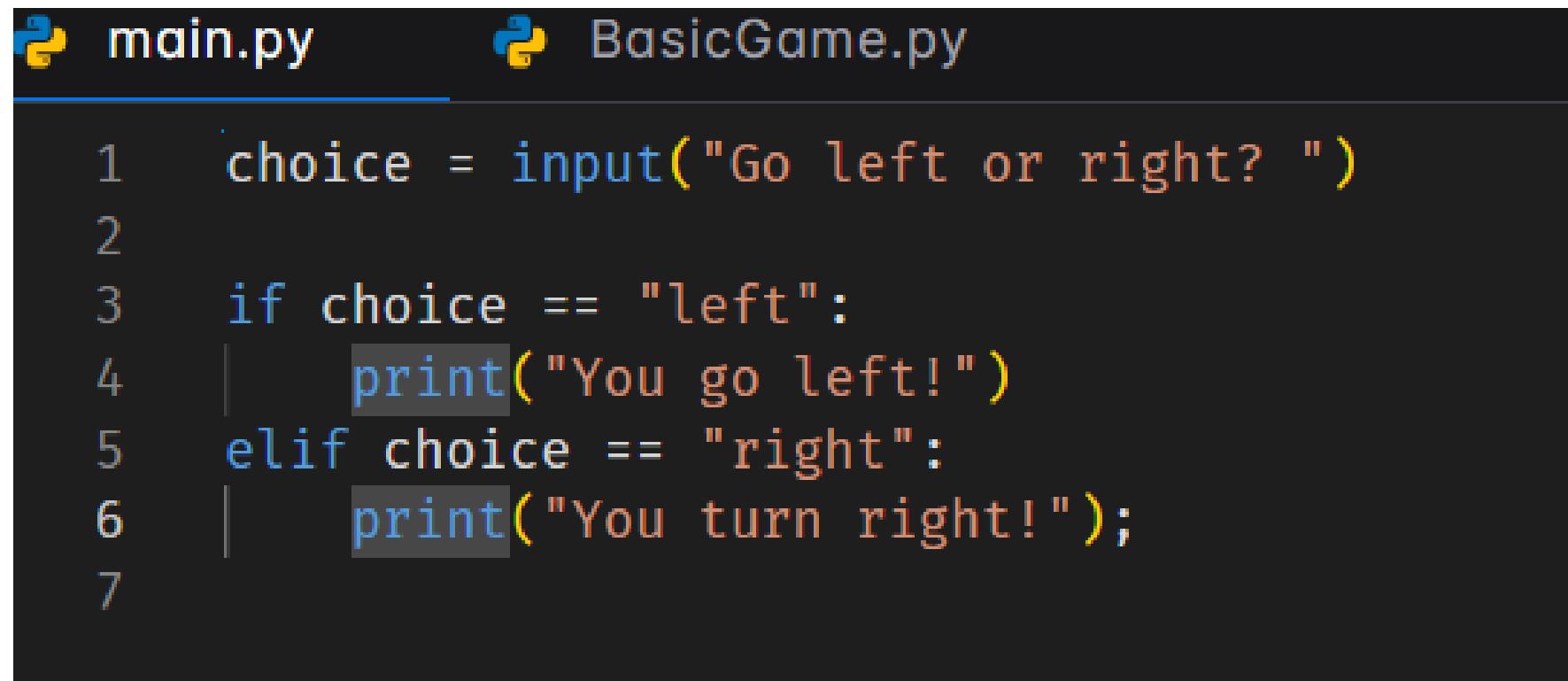
Let's say you want to allow players to pick a direction or option...

```
🐍 main.py          🐍 BasicGame.py

1 choice = input("Go left or right? ")
2
3 if choice == "left":
4     print("You go left!")
```

If Statements - The Basics

What if we want the player to be able to go left and right?

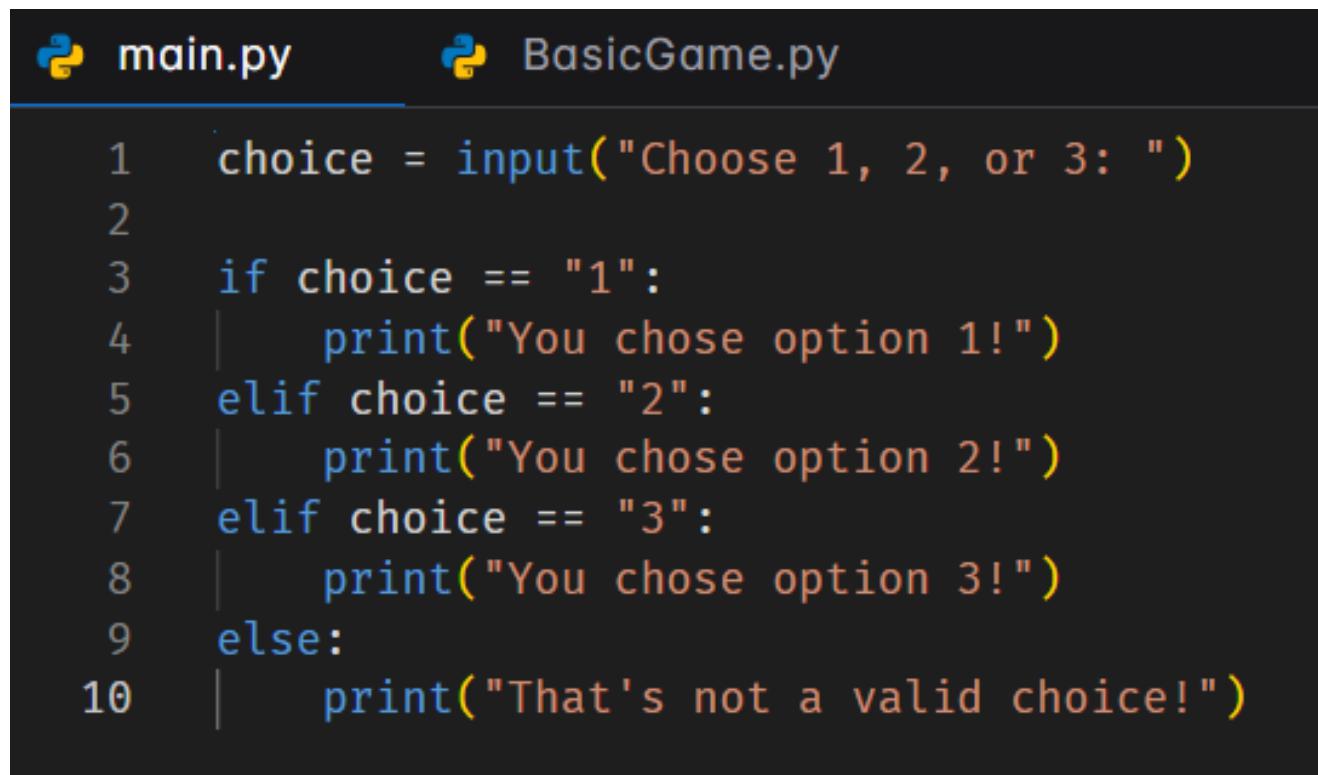


```
main.py          BasicGame.py

1 choice = input("Go left or right? ")
2
3 if choice == "left":
4     print("You go left!")
5 elif choice == "right":
6     print("You turn right!");
7
```

If Statements - The Basics

What if we want multiple choices plus a way to make sure the player knows they made a mistake?



```
main.py          BasicGame.py
1 choice = input("Choose 1, 2, or 3: ")
2
3 if choice == "1":
4     print("You chose option 1!")
5 elif choice == "2":
6     print("You chose option 2!")
7 elif choice == "3":
8     print("You chose option 3!")
9 else:
10    print("That's not a valid choice!")
```

else catches everything not caught by if/elif

- Useful for invalid inputs
- Always comes last
- No condition needed

UI/U

Make
playe
respo
code

```
7
8     if choice == "1":
9         print("You chose option 1!")
10    elif choice == "2":
11        print("You chose option 2!")
12    elif choice == "3":
13        print("You chose option 3!")
14    else:
15        print("That's not a valid choice!")
```

...

TERMINAL

You come to a fork in the road



Challenge 2: The first choice

Create a scenario with:

- A clear situation
- 2-3 numbered choices
- Different outcomes for each choice
- 1-2 sentences per outcome

```
mirror_mod = modifier_obj
# Set mirror object to mirror
mirror_mod.mirror_object = selected_obj

operation = "MIRROR_X":
    mirror_mod.use_x = True
    mirror_mod.use_y = False
    mirror_mod.use_z = False
operation == "MIRROR_Y":
    mirror_mod.use_x = False
    mirror_mod.use_y = True
    mirror_mod.use_z = False
operation == "MIRROR_Z":
    mirror_mod.use_x = False
    mirror_mod.use_y = False
    mirror_mod.use_z = True

# selection at the end - add
# _ob.select= 1
# mirr _ob.select=1
context.scene.objects.active = eval("Selected" + str(modifier))
mirror_ob.select = 0
bpy.context.selected_objects.append(data.objects[one.name].select)

print("please select exactly one object")
# - OPERATOR CLASSES ----

types.Operator:
    # X mirror to the selected object.mirror_mirror_x"
    "mirror X"
    context):
        # context.active_object is not None
```

Learning Intention:

Keeping Score - Tracking Stats Using Variables That Change

Success Criteria:

- I can create numeric variables
- I can change variable values
- I can display current stats
- Different choices change stats differently

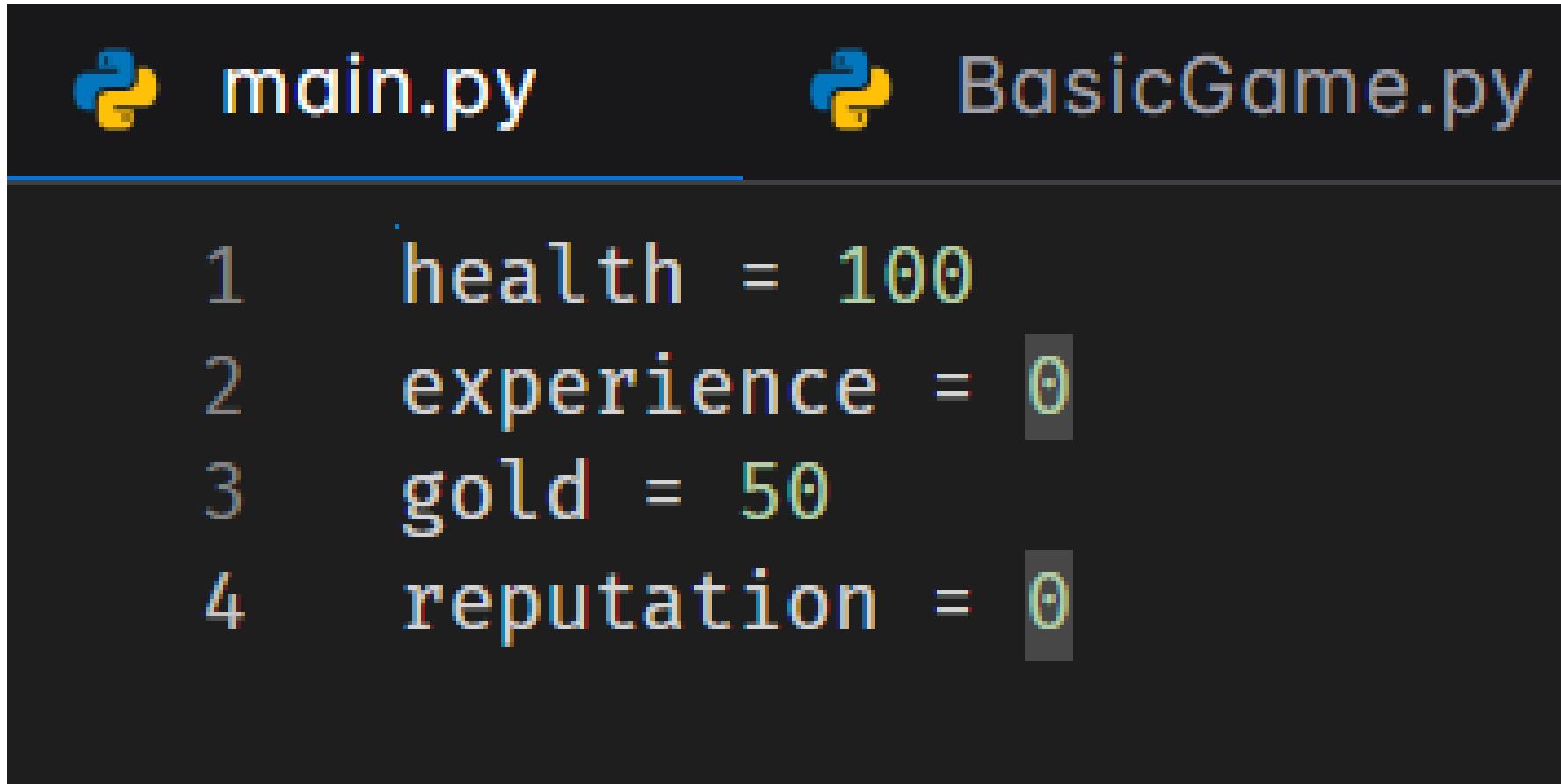
Why Track Stats?

Games need to remember:

- How much health you have
- How much gold you've collected
- Your experience points
- Your reputation
- Inventory items

Variables can do all of this!

Creating Numeric Variables



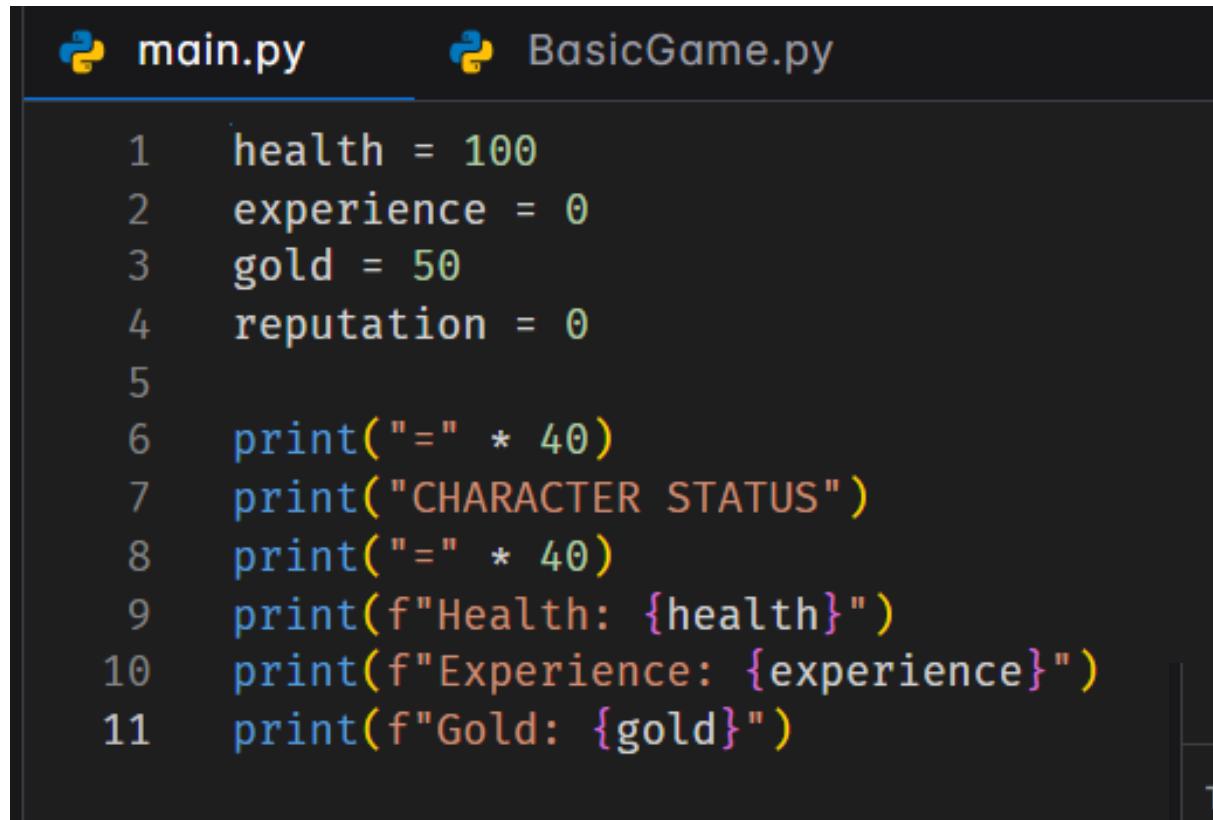
The image shows a dark-themed code editor interface with two tabs at the top: "main.py" and "BasicGame.py". Both tabs feature the Python logo icon. The "main.py" tab is active, indicated by a blue underline. Below the tabs, the code for "main.py" is displayed:

```
1 health = 100
2 experience = 0
3 gold = 50
4 reputation = 0
```

The code consists of four lines, each starting with a number (1, 2, 3, or 4) followed by a variable name and an equals sign, then a numeric value. The variable names are "health", "experience", "gold", and "reputation". The values are 100, 0, 50, and 0 respectively. The "experience" and "reputation" lines have small gray rectangular boxes with the number "0" inside them positioned to the right of the assignment operator.

Notice: No quotes! These are numbers, not text. Only Strings (text) needs quotes.

Displaying Stats



```
main.py
```

```
BasicGame.py
```

```
1 health = 100
2 experience = 0
3 gold = 50
4 reputation = 0
5
6 print("=" * 40)
7 print("CHARACTER STATUS")
8 print("=" * 40)
9 print(f"Health: {health}")
10 print(f"Experience: {experience}")
11 print(f"Gold: {gold}")
```

TERMINAL

```
=====
CHARACTER STATUS
=====
Health: 100
Experience: 0
Gold: 50
```

Changing Values

What happens if our player drinks a health potion or takes damage?
How can we reflect that in our stats?

```
main.py      BasicGame.py

1  health = 100
2  experience = 0
3  gold = 50
4  reputation = 0
5
6  print("=" * 40)
7  print("CHARACTER STATUS")
8  print("=" * 40)
9  print(f"Health: {health}")
10 print(f"Experience: {experience}")
11 print(f"Gold: {gold}")
12 print(" " * 40)
13 print("AH! You take an arrow to the knee, lose 20HP");
14 health = health - 20 # Take damage
15 print(f"Health is now: {health}")
```

```
TERMINAL
=====
CHARACTER STATUS
=====
Health: 100
Experience: 0
Gold: 50

AH! You take an arrow to the knee, lose 20HP
Health is now: 80
```

Changing variables with shorthand notation

Instead of `health = health - 20`

You can use shorthand...

`health -= 20 # Subtract 20`

`health += 50 # Add 50`

`gold *= 2 # Multiply by 2`

Both ways work! Use what makes sense to you.

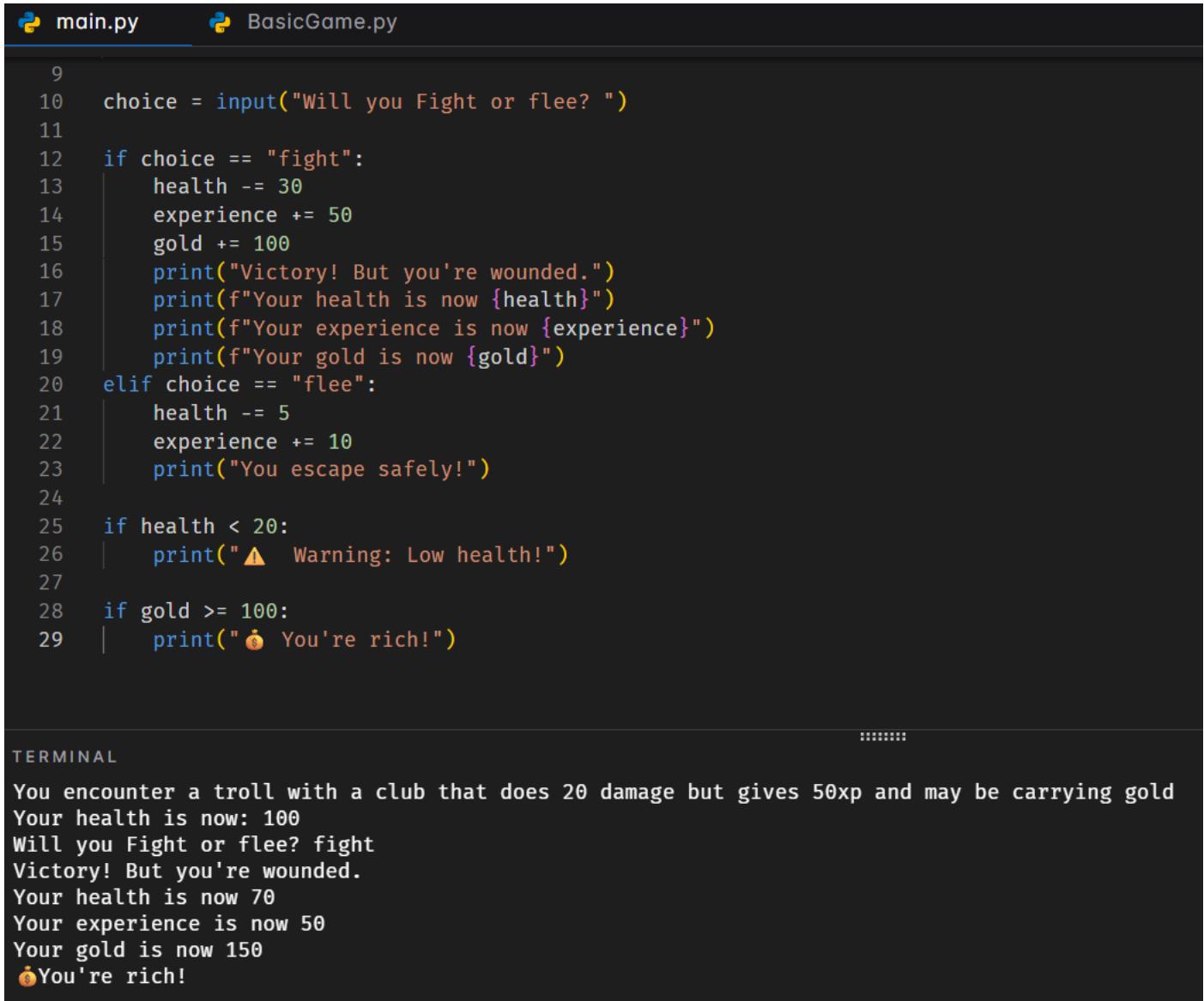
Stat Changes Based on Choices

```
main.py  BasicGame.py

1  health = 100
2  experience = 0
3  gold = 50
4  reputation = 0
5
6  print("You encounter a troll with a club that does 20 damage but gives 50xp and may be carrying gold");
7  print(f"Your health is now: {health}")
8
9
10 choice = input("Will you Fight or flee? ")
11
12 if choice == "fight":
13     health -= 30
14     experience += 50
15     gold += 100
16     print("Victory! But you're wounded.")
17     print(f"Your health is now {health}")
18     print(f"Your experience is now {experience}")
19     print(f"Your gold is now {gold}")
20 elif choice == "flee":
21     health -= 5
22     experience += 10
23     print("You escape safely!")

TERMINAL
You encounter a troll with a club that does 20 damage but gives 50xp and may be carrying gold
Your health is now: 100
Will you Fight or flee? fight
Victory! But you're wounded.
Your health is now 70
Your experience is now 50
Your gold is now 150
```

Triggering messages to the player based on stats

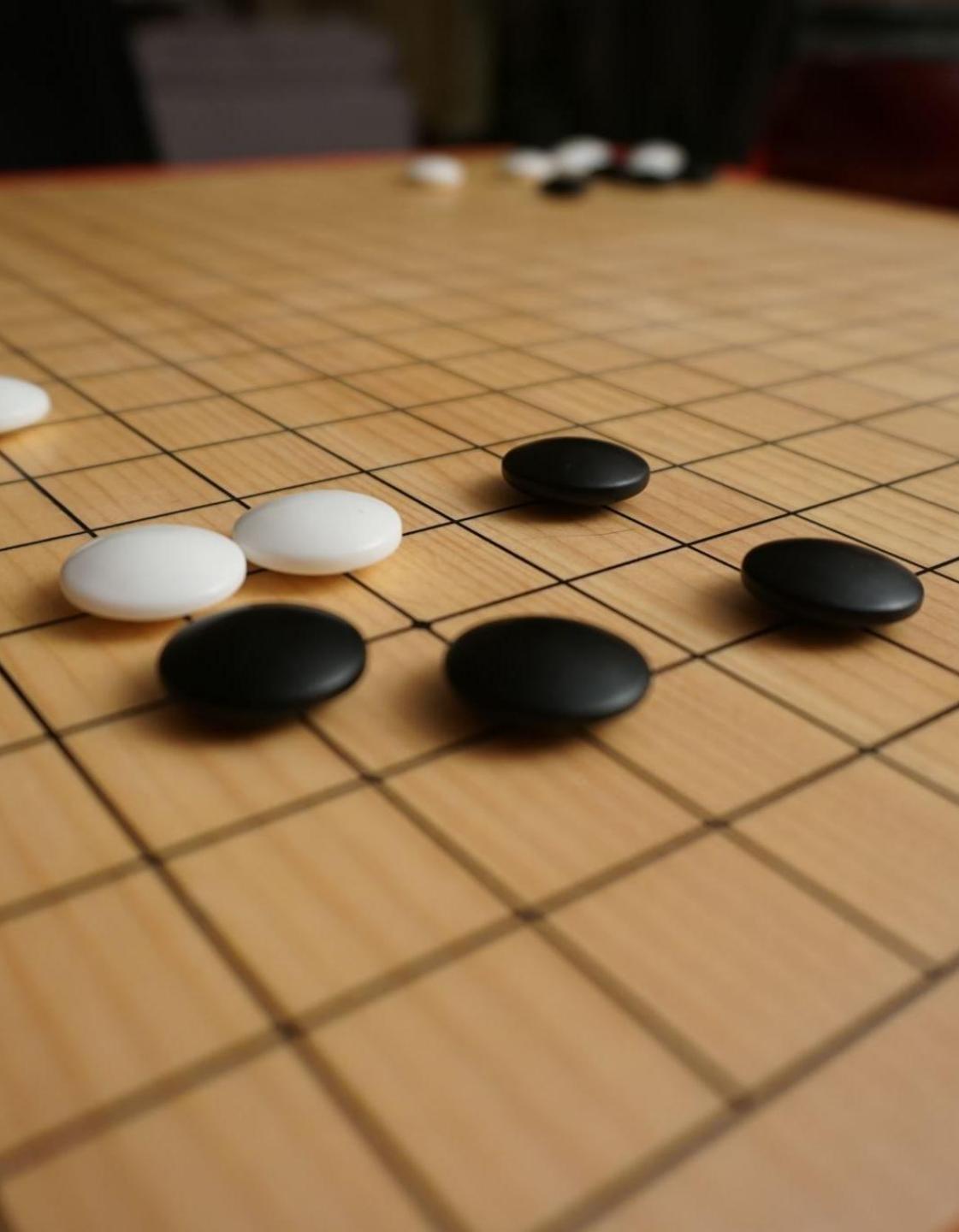


```
main.py          BasicGame.py
```

```
9
10    choice = input("Will you Fight or flee? ")
11
12    if choice == "fight":
13        health -= 30
14        experience += 50
15        gold += 100
16        print("Victory! But you're wounded.")
17        print(f"Your health is now {health}")
18        print(f"Your experience is now {experience}")
19        print(f"Your gold is now {gold}")
20    elif choice == "flee":
21        health -= 5
22        experience += 10
23        print("You escape safely!")
24
25    if health < 20:
26        print("⚠ Warning: Low health!")
27
28    if gold >= 100:
29        print("💰 You're rich!")
```

TERMINAL

```
You encounter a troll with a club that does 20 damage but gives 50xp and may be carrying gold
Your health is now: 100
Will you Fight or flee? fight
Victory! But you're wounded.
Your health is now 70
Your experience is now 50
Your gold is now 150
💰You're rich!
```



Challenge 3: The Encounter

Create code that:

- Sets up starting stats (3+)
- Presents an encounter
- Player makes a choice
- Stats change based on choice
- Shows before and after stats for both choices

```
mirror_mod = modifier_obj
    mirror object to mirror
    mirror_mod.mirror_object = ob
    operation == "MIRROR_X":
        mirror_mod.use_x = True
        mirror_mod.use_y = False
        mirror_mod.use_z = False
    operation == "MIRROR_Y":
        mirror_mod.use_x = False
        mirror_mod.use_y = True
        mirror_mod.use_z = False
    operation == "MIRROR_Z":
        mirror_mod.use_x = False
        mirror_mod.use_y = False
        mirror_mod.use_z = True

    selection at the end -add
    _ob.select= 1
    mirr_ob.select=1
    context.scene.objects.active = eval(
        ("Selected" + str(modifier)))
    mirror_ob.select = 0
    bpy.context.selected_objects.append(data.objects[one.name])
    data.objects[one.name].select = 1
    print("please select exactly one object")
    print("operator added to the operator classes")

- OPERATOR CLASSES ----

types.Operator:
    X mirror to the selected object.mirror_mirror_x"
    "mirror X"
    context):
    context.active_object is not None
```

Learning Intention:

TAKING CHANCES- Random Events Adding Chance and Risk

Success Criteria:

- I can import random
- I can use random.randint()
- I can create percentage chances
- Random outcomes affect the game

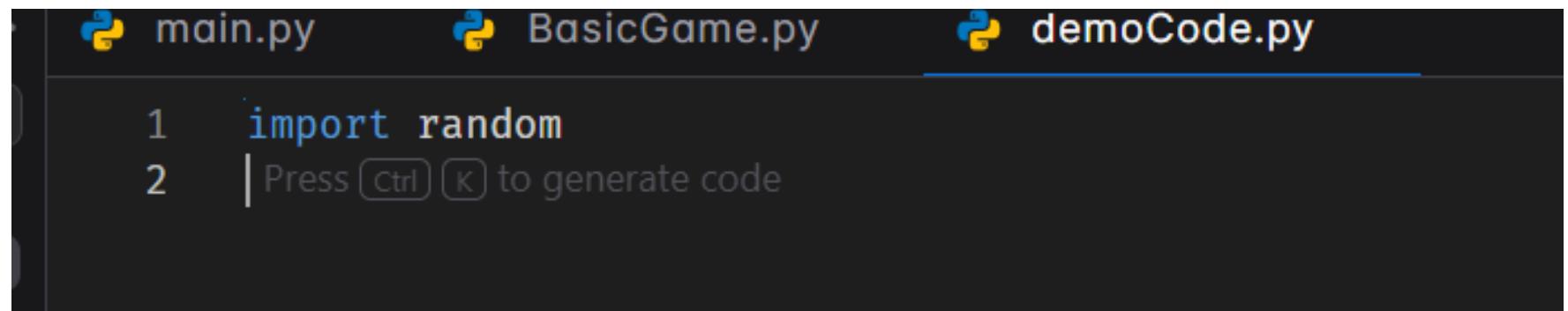
Why Randomness?

- Games are more exciting when:
- Outcomes aren't guaranteed
- There's risk and reward
- You can't predict everything
- Luck plays a role

Random = unpredictable!

The Random Module

- **Put this at the very TOP of your file!**
- This gives you access to random functions



```
main.py          BasicGame.py          demoCode.py
1 import random
2 | Press [ctrl] K to generate code
```

Random Numbers

- `random.randint(1, 6)` gives a random number from 1 to 6
- Like rolling a 6-sided die!
- Could be 1, 2, 3, 4, 5, or 6

- How would you simulate a D20?

```
1 import random
2
3 roll = random.randint(1, 6)
4 print(f"You rolled: {roll}")
```

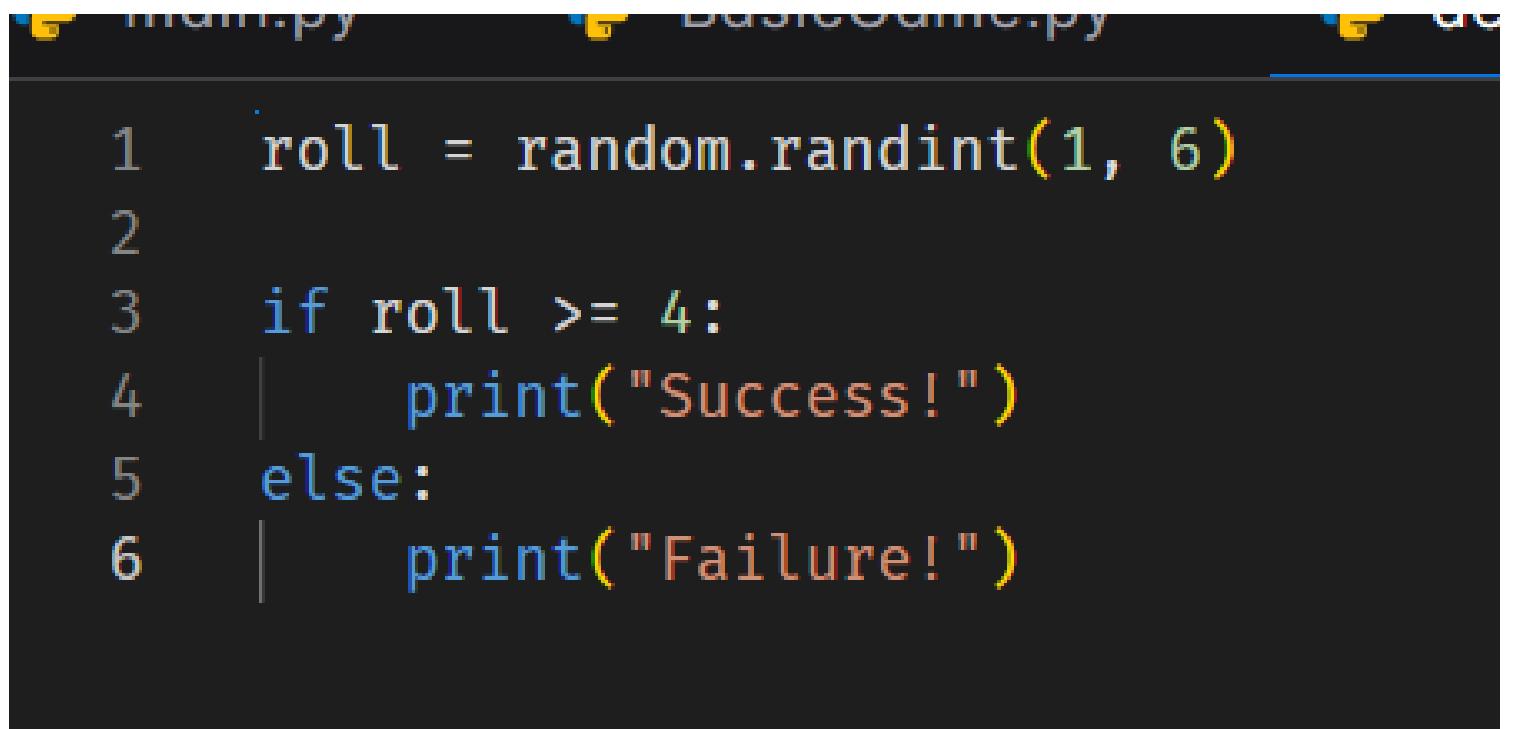
TERMINAL

You rolled: 2

** Process exited - Return Code: 0 **

Using Random for Success/Failure

- 50% chance: rolls 4, 5, or 6 succeed
- Notice ≥ 4



```
main.py
```

```
BasicGame.py
```

```
1 roll = random.randint(1, 6)
2
3 if roll >= 4:
4     print("Success!")
5 else:
6     print("Failure!")
```

Percentage Chances

Pattern:

- Generate number 1-100
- Check if \leq your percentage

```
1     chance = random.randint(1, 100)
2
3     if chance <= 40:
4         print("Success! (40% chance)")
5     else:
6         print("Failure! (60% chance)")
```

Common Patterns

```
1 import random
2 # 50/50 chance
3 if random.randint(1, 2) == 1:
4     print("Heads!")
5 else:
6     print("Tails!")
7
8 # 25% chance
9 if random.randint(1, 100) <= 25:
10    print("Rare event!")
```

Random in Combat

- What is the code doing?

```
1 import random
2 print("You attack the dragon!")
3 chance = random.randint(1, 100)

4

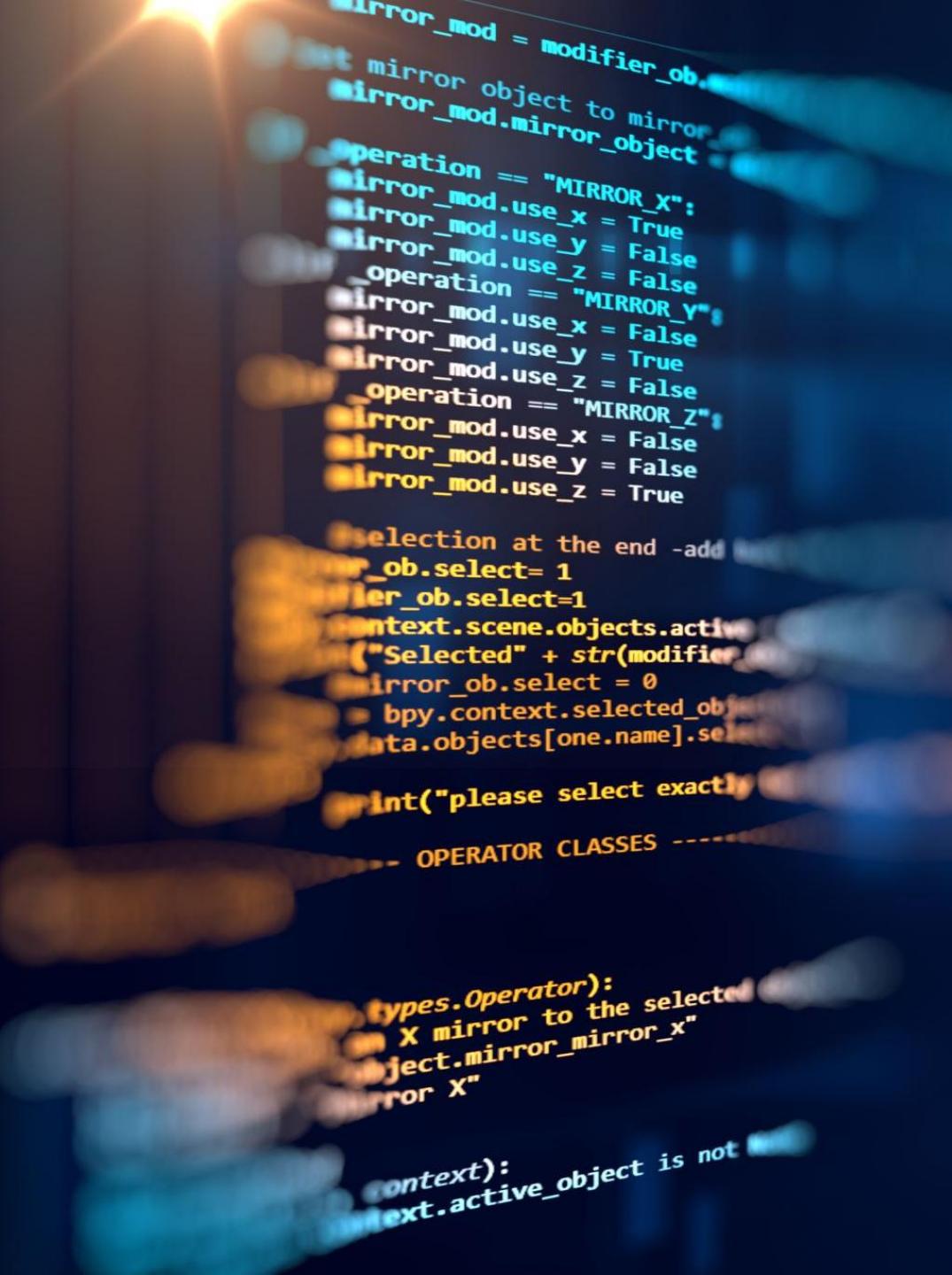
5 if chance <= 40:
6     print("Hit! 50 damage!")
7     boss_health -= 50
8     xp += 100
9 else:
10    print("Miss! The dragon counters!")
11    health -= 30
```



Challenge 4: The Risky Decision

Create a risky situation with:

- Clear description of the risk
- A % chance of success (you choose!)
- `random.randint()` to determine outcome
- Different results for success/failure
- Stats change based on outcome

A photograph of a person's hand pointing towards a computer screen. The screen displays a block of Python code. The code is related to Blender's operator classes, specifically for creating mirror modifiers. It includes logic for selecting objects, checking if they are mirrors, and setting up mirror modifiers with specific axis settings. The code uses various Python libraries like bpy and context, and includes comments explaining its purpose.

```
mirror_mod = modifier_obj
# Set mirror object to mirror
mirror_mod.mirror_object = selected_obj
# operation == "MIRROR_X":
mirror_mod.use_x = True
mirror_mod.use_y = False
mirror_mod.use_z = False
# operation == "MIRROR_Y":
mirror_mod.use_x = False
mirror_mod.use_y = True
mirror_mod.use_z = False
# operation == "MIRROR_Z":
mirror_mod.use_x = False
mirror_mod.use_y = False
mirror_mod.use_z = True

# selection at the end - add
# _ob.select= 1
# mirr_ob.select=1
context.scene.objects.active = eval("Selected" + str(modifier))
mirr_ob.select = 0
bpy.context.selected_objects.append(data.objects[one.name])
# print("please select exactly one object")
# - OPERATOR CLASSES ----

types.Operator:
    # X mirror to the selected object.mirror_mirror_x"
    "mirror X"
    # context):
    # context.active_object is not None
```

Learning Intention:

LEVELING UP - Complex Interactions Combining Everything We've Learned

Success Criteria:

- I can check stats before allowing actions
- I can use and/or for complex conditions
- I can create stat-based chances
- I can combine all concepts together

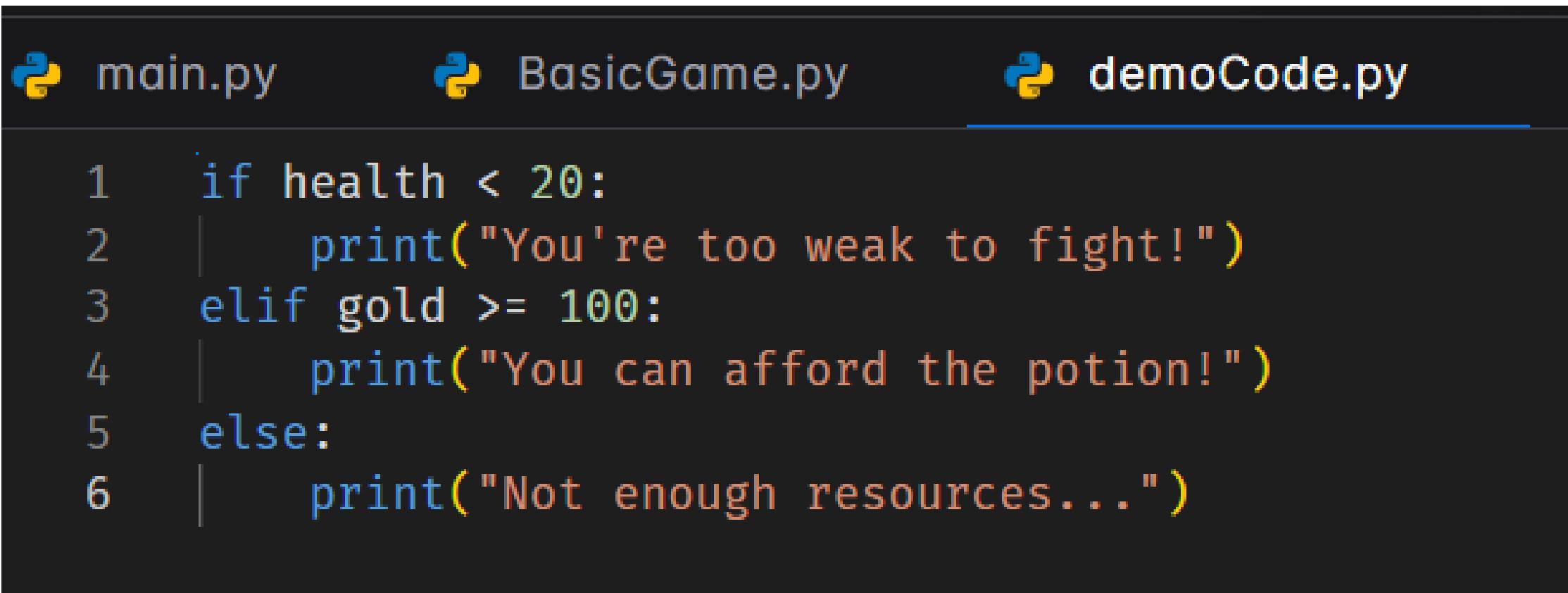
What We Know

- print() and input()
- Variables (text and numbers)
- if/elif/else
- Changing stats
- Random events

Today: Put it all together!

Checking Before Allowing

- Check stats BEFORE letting players do things!

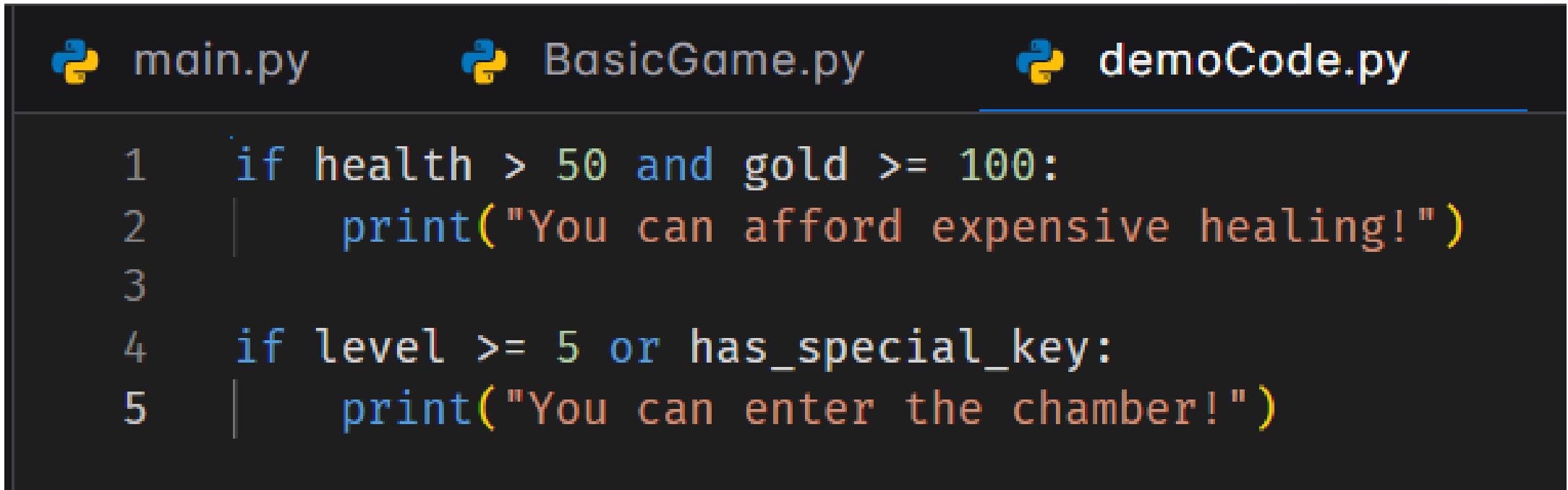


The image shows a code editor interface with three tabs at the top: "main.py", "BasicGame.py", and "demoCode.py". The "demoCode.py" tab is currently selected, indicated by a blue underline. Below the tabs, there is a code snippet:

```
1 if health < 20:  
2     print("You're too weak to fight!")  
3 elif gold >= 100:  
4     print("You can afford the potion!")  
5 else:  
6     print("Not enough resources...")
```

Multiple Conditions

- **and** -> both must be true
- **or** -> at least one must be true

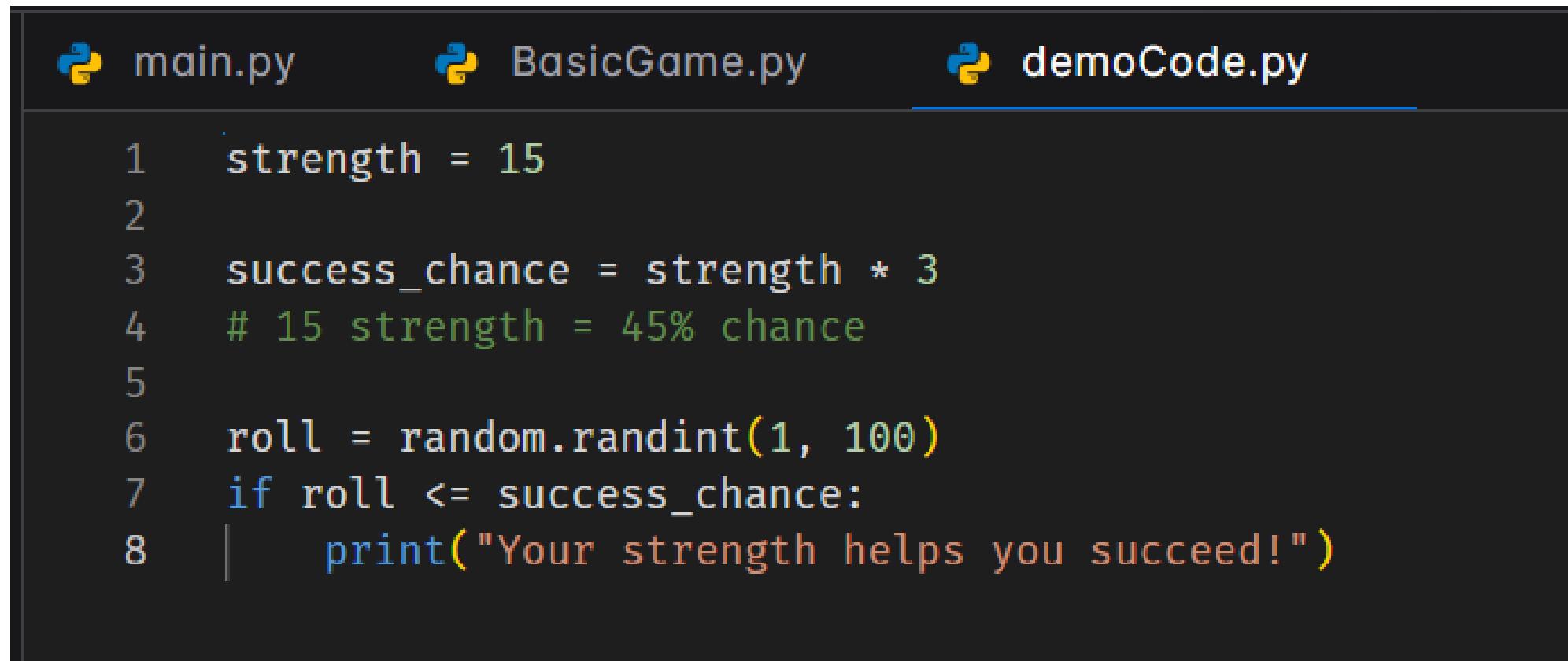


```
main.py          BasicGame.py          demoCode.py

1  if health > 50 and gold >= 100:
2      print("You can afford expensive healing!")
3
4  if level >= 5 or has_special_key:
5      print("You can enter the chamber!")
```

Stat-Based Chances

- Higher stats = better chances!



```
main.py          BasicGame.py          demoCode.py
1 strength = 15
2
3 success_chance = strength * 3
4 # 15 strength = 45% chance
5
6 roll = random.randint(1, 100)
7 if roll <= success_chance:
8     print("Your strength helps you succeed!")
```

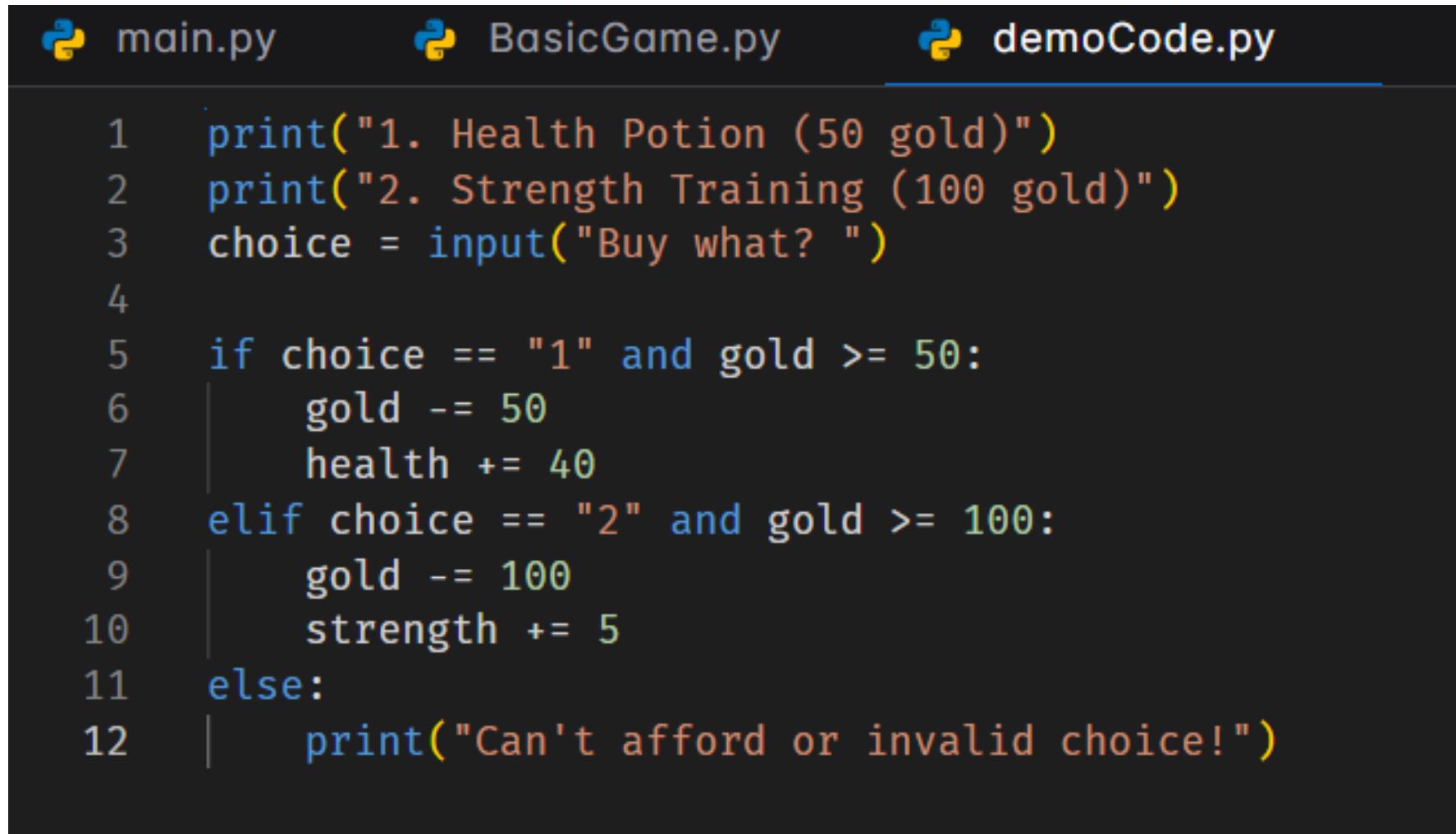
Complex Example

- `min(health + 40, 100)`
- heals 40 but caps total health at 100

```
main.py          BasicGame.py          demoCode.py
1  gold = 60
2  health = 70
3
4  if gold >= 50:
5      gold -= 50
6      health = min(health + 40, 100)
7      print(f"Potion bought and used! Health is now {health}")
8 elif gold < 50:
9     print("Not enough gold!")

TERMINAL
Potion bought and used! Health is now 100
```

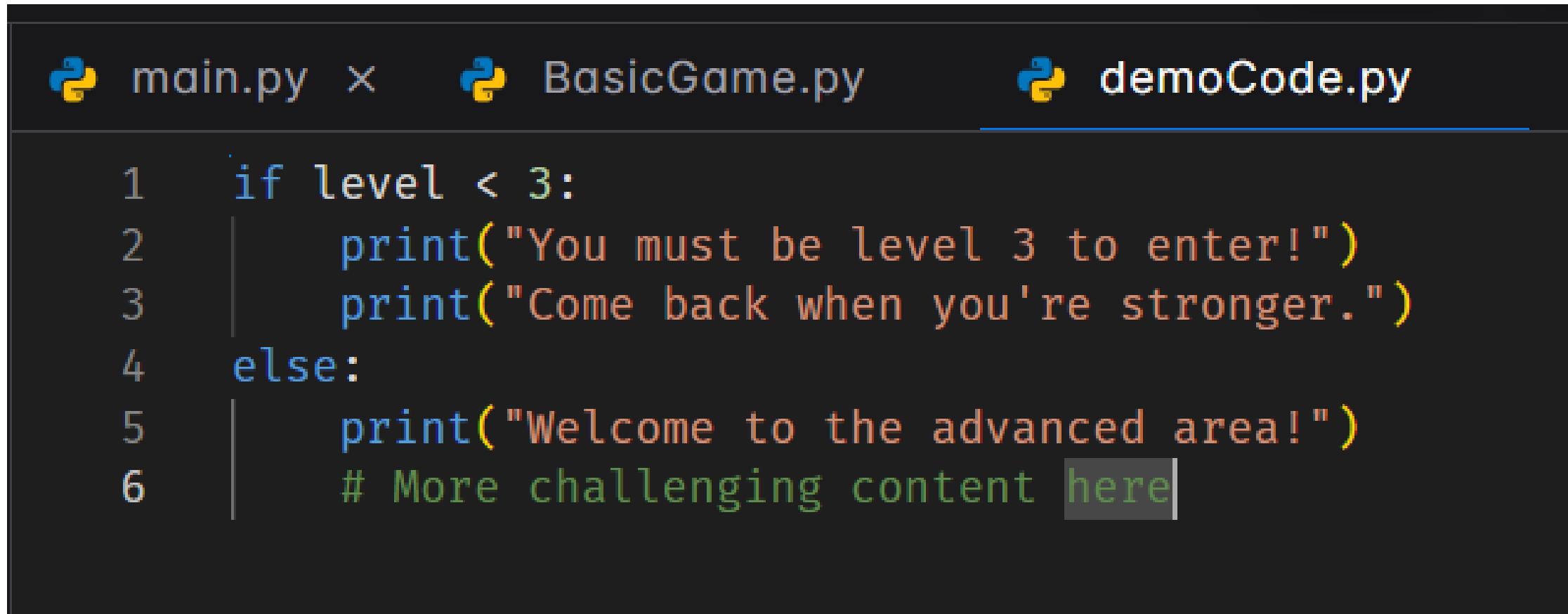
The Shop Pattern



The screenshot shows a code editor with three tabs at the top: "main.py", "BasicGame.py", and "demoCode.py". The "demoCode.py" tab is currently selected, indicated by a blue underline. The code in "demoCode.py" is a script for a shop pattern:

```
1 print("1. Health Potion (50 gold)")  
2 print("2. Strength Training (100 gold)")  
3 choice = input("Buy what? ")  
4  
5 if choice == "1" and gold >= 50:  
6     gold -= 50  
7     health += 40  
8 elif choice == "2" and gold >= 100:  
9     gold -= 100  
10    strength += 5  
11 else:  
12     print("Can't afford or invalid choice!")
```

Level-Gating Content



```
main.py × BasicGame.py demoCode.py

1 if level < 3:
2     print("You must be level 3 to enter!")
3     print("Come back when you're stronger.")
4 else:
5     print("Welcome to the advanced area!")
6     # More challenging content here
```



Challenge 5: The Boss Fight

Create a boss encounter with:

- Player AND boss stats
- 2-3 combat options
- Requirements for special moves
- Stat-based success chances
- Multiple possible outcomes
- Bonus: Add a shop before the boss fight (like in Hades!)

A photograph of a person's hand pointing towards a computer screen. The screen displays a block of Python code. The code is related to a character inventory system, specifically a 'mirror' operator. It includes logic for selecting objects based on their type and name, setting properties like 'use_x', 'use_y', and 'use_z' to true or false, and handling user input for selecting exactly one object. The code is color-coded with syntax highlighting.

Learning Intention:

Character Inventory Systems: From Lists to Libraries

Success Criteria:

- I can use lists to build a basic inventory system
- I can use libraries to build more robust and varied inventory systems

What is an Inventory System?

A way to store and manage items your character carries



Items



Storage



Actions

Simple Approach: Lists

```
# Create an inventory
inventory = []

# Add items
inventory.append("sword")
inventory.append("shield")
inventory.append("potion")

# Check inventory

print(inventory)

# Output: ['sword', 'shield',
'potion']
```

- Simple and easy to understand
- Good for basic item tracking

Lists in Action

```
# Remove an item
inventory.remove("potion")
# Check if item exists
if "sword" in inventory:
    print("You have a sword!")

# Count items
item_count = len(inventory)
print(f"You have {item_count} items")

# Display all items

for item in inventory:

    print(f"-{item}")
```

Advanced: Dictionary Library

```
# Create inventory with item details
inventory = {
    "sword": {"damage": 10, "type": "weapon"},

    "shield": {"defense": 8, "type": "armor"},

    "potion": {"healing": 20, "type": "consumable"}
}

# Access item properties

print(inventory["sword"]["damage"])

# Output: 10
```

Using a dictionary will allow us to

- Store multiple properties per item
- Make more complex inventory systems

Dictionary in Action

```
# Add new item
inventory[
    "bow"] = {"damage": 8, "type": "weapon"}
# Remove item
del inventory["potion"]
# Check if item exists
if "sword" in inventory:
    print("Sword equipped!")

# Loop through all items
for item, stats in inventory.items():

    print(f"{item}:{stats}")
```

When to Use Each?

Lists

- Simple item names
- No extra details needed
- Quick and easy
- Great for beginners

```
[ "sword", "shield" ]
```

Dictionaries

- Item properties
- Stats and details
- More organized
- Better for RPGs

```
{ "sword": { "dmg": 10 } }
```

Your Turn!

Practice Challenge

Create an inventory system for a game character with at least 5 items

Choose your approach:

- Simple: Use a list
- Advanced: Use a dictionary with item stats



Bonus: Add functions to add and remove items!

```
mirror_mod = modifier_obj
# Set mirror object to mirror
mirror_mod.mirror_object = selected_obj

operation = "MIRROR_X":
mirror_mod.use_x = True
mirror_mod.use_y = False
mirror_mod.use_z = False

operation == "MIRROR_Y":
mirror_mod.use_x = False
mirror_mod.use_y = True
mirror_mod.use_z = False

operation == "MIRROR_Z":
mirror_mod.use_x = False
mirror_mod.use_y = False
mirror_mod.use_z = True

# Selection at the end - add
# _ob.select= 1
# other_ob.select=1
context.scene.objects.active = eval("Selected" + str(modifier))
mirror_ob.select = 0
bpy.context.selected_objects.append(data.objects[one.name].select)

int("please select exactly one object")
# - OPERATOR CLASSES ----

types.Operator:
    # X mirror to the selected object.mirror_mirror_x"
    mirror_X"

context):
    # context.active_object is not None
```

Learning Intention:

LEVELING UP - Object Oriented Programming

Creating editable characters

Success Criteria:

- I can use classes in my code
- I can create, call and modify objects

What Are Objects?

Think about real-world objects...



Car

Has: color, speed, fuel

Does: accelerate,
brake



Phone

Has: battery, apps

Does: call, message

Objects have PROPERTIES and can DO things!

Your First Class

A **class** is a blueprint for creating objects

```
class Dog:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    def bark(self):  
        print(f"{self.name} says Woof!")
```

__init__ sets up the object

self refers to this object

Attributes vs Methods

Attributes

What an object **HAS**

- Properties of the object
- Data it stores

```
dog.name  
dog.age
```

Methods

What an object can **DO**

- Actions it can perform
- Functions inside the class

```
dog.bark()  
dog.sit()
```

Creating Multiple Objects

One class can create **many objects!**

```
# Create different dogs
```

```
buddy = Dog("Buddy", 3)
```

```
max = Dog("Max", 5)
```

```
bella = Dog("Bella", 2)
```

```
# Each dog is unique!
```

```
buddy.bark() # Buddy says Woof!
```

```
max.bark() # Max says Woof!
```

Key Idea: Same blueprint, different objects!

What is Inheritance?

Create **new classes** based on existing ones!

Parent Class

Has basic features



Child Class 1

Inherits + adds own

Child Class 2

Inherits + adds own

Let's Build Game Characters!

Use OOP to create different character types



Warrior

- High health
- Strong attacks
- Special: Power Strike



Mage

- Lower health
- Magic attacks
- Special: Cast Spell



Archer

- Medium health
- Quick attacks
- Special: Multi-shot



Warrior Class

```
class Warrior(Character):
    def __init__(self, name):
        super().__init__(name, health=150, strength=20)
        self.armor = 10

    def special_attack(self, target):
        print(f"{self.name} uses POWER STRIKE!")
        target.take_damage(self.strength * 2)
```

Inherits from Character class

`super()` calls parent's `__init__`

Special Attack: Double damage! 



Mage Class

```
class Mage(Character):
    def __init__(self, name):
        super().__init__(name, health=80, strength=10)
        self.mana = 100

    def cast_spell(self, target):
        if self.mana >= 20:
            print(f"{self.name} casts FIREBALL!")
            target.take_damage(30)
            self.mana -= 20
```

Lower health than Warrior

Mana resource for spells

Magic Attack: Costs mana to cast! ⚡

Let's Battle!

Create characters

```
knight = Warrior("Sir Lancelot")  
wizard = Mage("Merlin")
```

Battle!

```
knight.attack(wizard)  
wizard.cast_spell(knight)
```

```
knight.special_attack(wizard) # POWER STRIKE!
```

What We Learned

- Classes create objects
- Objects have attributes & methods
- Inheritance shares code



Your Turn!

Design your own character classes!

```
1  class Dog:
2      def __init__(self, name, age):
3          self.name = name
4          self.age = age
5
6      def bark(self):
7          print(f"{self.name} says woof!")
8
9  scooby = Dog("scooby", 67)
10 scooby.bark()
11
12 class Big_Dog(Dog):
13     def __init__(self, name, age, long_tail):
14         super().__init__(name, age)
15         self.long_tail = long_tail
16
17     def bite(self):
18         print(f"{self.name} bites!")
19
20 big_scooby = Big_Dog("scooby", 67, True)
21 big_scooby.bite()
22 print(f"Long tail?: {big_scooby.long_tail}")
```

TERMINAL

```
scooby says woof!
scooby bites!
Long tail?: True
```

** Process exited - Return Code: 0 **