Conditionals

Learning Objectives

- Create and evaluate boolean expressions that answer questions about the state of a program's data
- Use `if`, `elif` and `else` keywords to build conditional statements that control the flow of a program
- Choose among several enumerated possibilities using the match & case keywords

Conditions & Conditionals

Conditions & Conditionals

Like humans, programs should be able to make decisions based on **conditions**

- **Conditions** are the states of the data in your program
- **Conditional statements** allow a program to decide to execute some code if a condition is `True` and another part if it is `False`
  - e.g. the *Pedestrian Program:* "if the light is green, *walk*; else, *stop*"
  - first example of modifying **control flow**, or order of program execution

Conditions as Boolean Expressions

Boolean expressions evaluate to `bool` values, i.e. either `True` or `False`.

```
3 > 4 and 9 == (81 / 9)                    # always True
not True and True or False and not False   # always False
```

We are also able to write boolean expressions that contain variables.

```
x % 3 == 2 and x > 5                       # not always True or False!
```

This expression's value changes based on the value of `x`!

Can you think of a value of `x` that would cause the expression to evaluate to `True`? What about `False`?

Testing the State of the World

When we use variables as part of boolean expressions, we are able to test conditions about the state of the world that our program represents.

- Compare values with *relational operators*
- Combine boolean expressions with *logical operators.*

The Boolean Expression Toolkit

Relational Operators:

Operator/method	Input Types	Description
`<` / `<=`	`int`, `float`, `str`	less than / less than or equal to
`>` / `>=`	`int`, `float`, `str`	greater than / greater than or equal to
`==` / `!=`	`int`, `float`, `str`	equal to / not equal to

The Boolean Expression Toolkit

Logical Operators:

Operator/method	Input Types	Description
`and`	`bool`	evaluates to `True` only if both inputs are `True`
`or`	`bool`	evaluates to `True` as long as at least one input is `True`
`not`	`bool`	negates a single `bool` value to its opposite

Writing Expressions to Test Conditions

Conditions are most useful when they model real-world concepts:

- *"Is the concert tonight sold out?"*
- *"Is the user's suggested password valid?"*

Sometimes the answers will be "yes" and sometimes "no", all depending on the values stored in the underlying variables.

Writing Expressions to Test Conditions

As usual, in programming, we want to be as specific as possible!

Original	➡️	Rephrased
"Is the concert tonight sold out?"	➡️	*"Is the number of tickets sold equal to the capacity for the venue?"*
"Is the user's suggested password valid?"	➡️	*"Is the user's password long enough to be valid and is it different from their username?"*

Writing Expressions to Test Conditions

Being specific lets us write expressions in terms of variables & relational operators.

Rephrased	➡️	Code
"Is the number of tickets sold equal to the capacity for the venue?"	➡️	`num_tickets == venue_capacity`
"Is the user's password long enough to be valid and is it different from their username?"	➡️	`len(password) >= 8 and password != username`


if

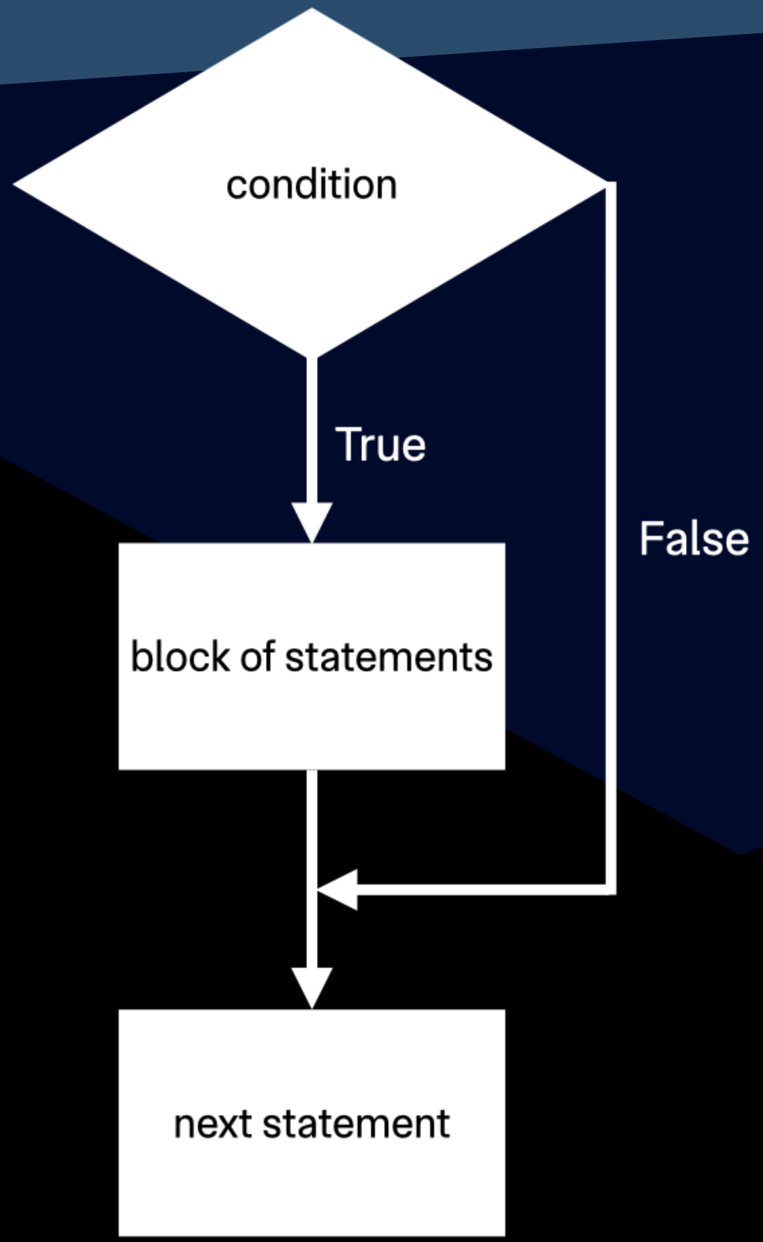
The `if` Statement

> *"`if` music be the food of love, play on."* — William Shakespeare

The `if` statement allows us to specify a portions of our program that should be run **only in the case that** a certain condition is met.

```
if my_boolean_expression:
        statement_one
        statement_two
        ...
        statement_last
```

Control Flow & `if`



- Test the condition...
  - **if** it is `True`, execute the block of statements
  - otherwise, proceed to the next statement.

Using `if`

```
num = input("Pick a number: ")
num = int(num)
print("Printing a message if {num} is divisible by 5...")
if num % 5 == 0:
        print("Yes!")
print("All done.")
```

```
$ python pick_a_number.py
Pick a number: 20
Printing a message if 20 is divisible by 5...
Yes!
All done.
```

Using `if`

```
num = input("Pick a number: ")
num = int(num)
print("Printing a message if {num} is divisible by 5...")
if num % 5 == 0:
        print("Yes!")
print("All done.")
```

```
$ python pick_a_number.py
Pick a number: 13
Printing a message if 13 is divisible by 5...
All done.
```

Multiple `if` Statements

Multiple `if` statements in a program are evaluated independently and in order.

```
username = "inspector_norse"
password = "0451"
if len(password) < 8:
        print("Bad Password: Not long enough!")
if password == username:
        print("Bad Password: Same as username!")
```

🖨️👇

```
Bad Password: Not long enough!
```

Nesting `if` Statements

`if` statements are statements, so they can be members of the bodies of other `if` statements!

```
if month <= 7:
        if month % 2 == 1:
                print(f"Month {month} has 31 days.")
        if month % 2 == 0 and month != 2:
                print(f"Month {month} has 30 days.")
        if month == 2:
                print(f"Month {month} has 28 days.")
if month > 7:
        if month % 2 == 1:
                print(f"Month {month} has 30 days.")
        if month % 2 == 0:
                print(f"Month {month} has 31 days.")
```

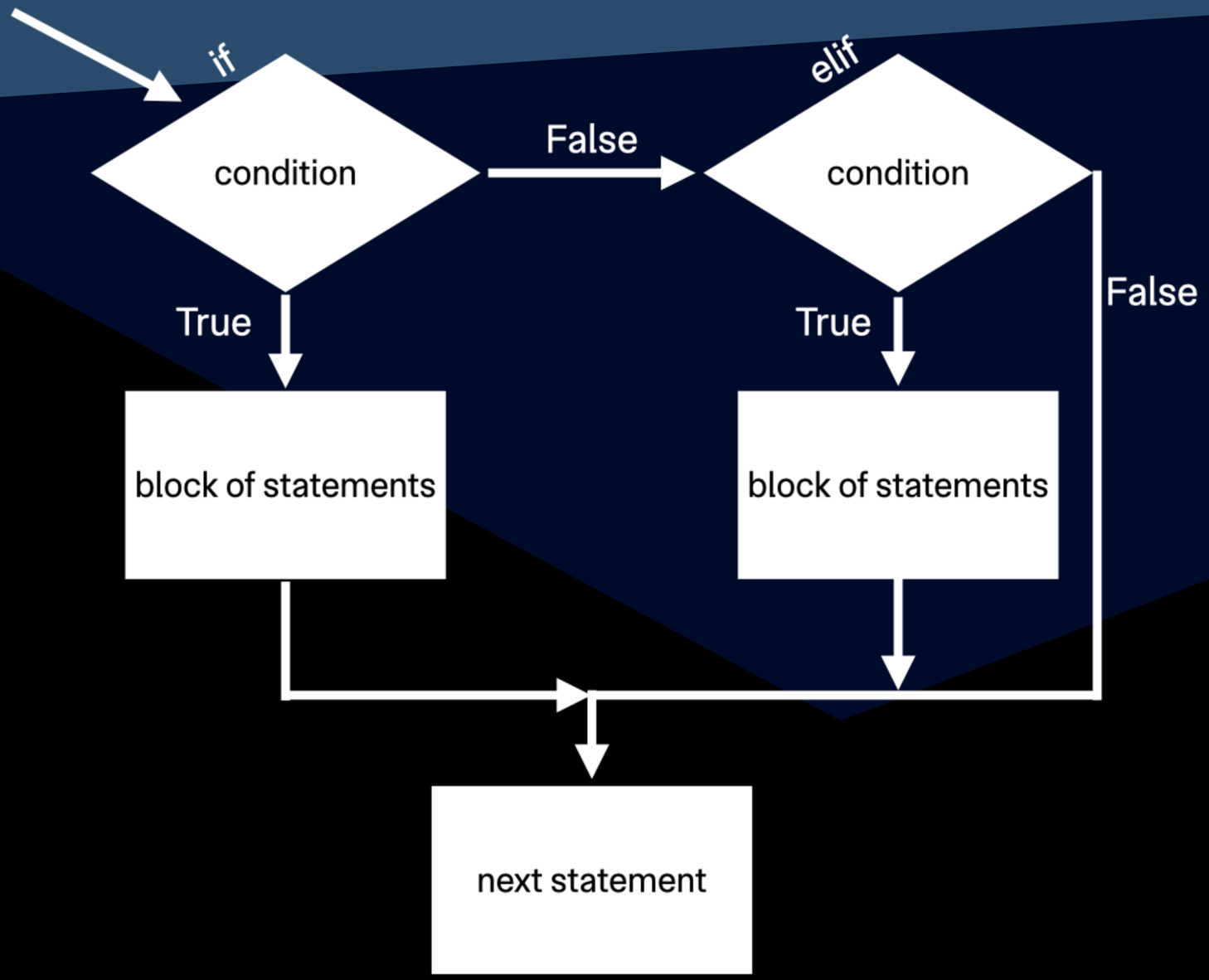

elif

`elif`: Choosing One Of Several Options

`elif` allows you to specify an *alternative condition* that is be tested *only when all previous conditions were* `False`.

The `elif` syntax:

```
if first_boolean_expression:
        statement_one
        statement_two
        ...
        statement_last
elif alternative_boolean_expression:
        statement_a
        statement_b
        ...
        statement_z
```



Control Flow & `elif`

`if` and `elif` statements represent *mutually exclusive* choices: we may execute the body of one, the other, or neither, but *never both*.

`elif`: Outdoor Activities

```
temperature = 90
if temperature > 85:
        print("Go to the beach. 🏖️")
elif temperature > 55:
        print("Go hiking. 🥾")
```

🖨️ 👇

```
Go to the beach. 🏖️
```

Possible outcomes:

- `temperature > 85` ➡️ 🏖️
- `85 >= temperature > 55` ➡️ 🥾
- `55 >= temperature` ➡️ nothing!

Assigning Letter Grades

```
exam_score = 94
letter_grade = "F"
if exam_score > 90:
        letter_grade = "A"
if exam_score > 80:
        letter_grade = "B"
if exam_score > 70:
        letter_grade = "C"
if exam_score > 60:
        letter_grade = "D"
print(f"Your exam score of {exam_score} earns: {letter_grade}.")
```

🖨️👇

```
Your exam score of 94 earns: D.
```

🤯🤯🤯🤯

Assigning Letter Grades

```
exam_score = 94
letter_grade = "F"
if exam_score > 90:
        letter_grade = "A"
elif exam_score > 80:
        letter_grade = "B"
elif exam_score > 70:
        letter_grade = "C"
elif exam_score > 60:
        letter_grade = "D"
print(f"Your exam score of {exam_score} earns: {letter_grade}.")
```

🖨️👇

```
Your exam score of 94 earns: A.
```

✅✅✅✅

Multiple Conditional Chains

```
transaction_completed = False
if account_balance < item_price:
        print("Insufficient funds to complete transaction. Transaction cancelled.")
elif account_balance > item_price:
        transaction_completed = True
        print(f"Completing transaction; dispensing change amount of {account_balance - item_price}")
elif account_balance == item_price:
        transaction_completed = True
        print("Completing transaction. Have a nice day.")

if transaction_completed and item_price > 10.00:
        print("Printing $2.50 coupon for your next visit.")
elif transaction_completed and item_price > 5.00:
        print("Printing $1.00 coupon for your next visit.")
```

else

Outdoor Activities (On A Cold Day)

```
if temperature > 85:
        print("Go to the beach. 🏖️")
elif temperature > 55:
        print("Go hiking. 🥾")
```

🖨️ 👇

Possible outcomes:

- `temperature > 85` ➡️ 🏝️
- `85 >= temperature > 55` ➡️ 🥾
- `55 >= temperature` ➡️ nothing!

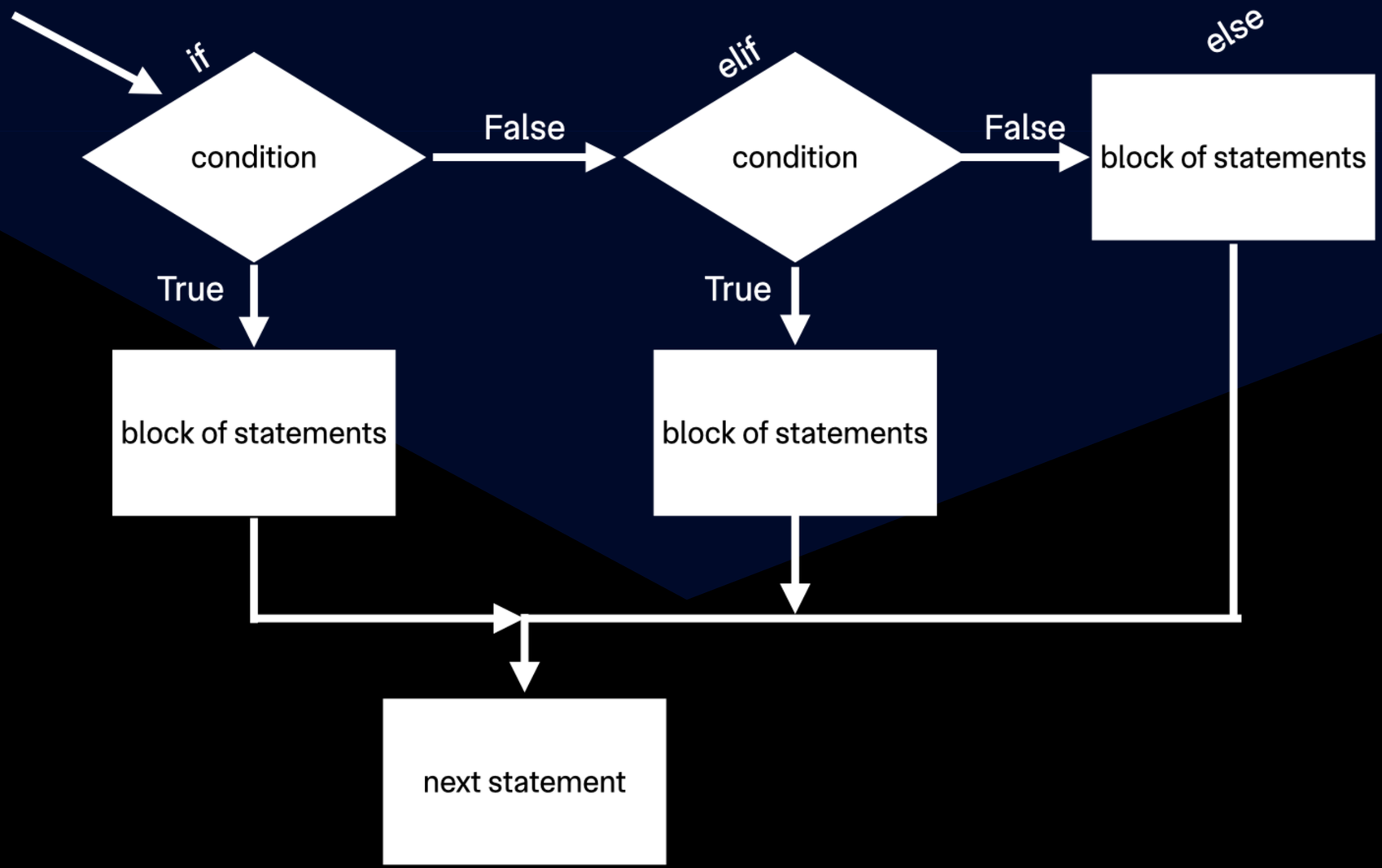
What to do on a cold day?

`else`: Provide A Default Outcome

The `else` keyword allows us to define a body of statements that will be run *no matter what* in the case that all previous conditions were not met.

```
if first_boolean_expression:
    block_one
elif alternative_boolean_expression:
    block_two
# optionally many elif statements provided here...
else:
    block_three
```

Look: no new condition provided!


if
condition
False
elif
condition
False
else
block of statements
True
block of statements
True
block of statements
next statement

Nesting with `else` / `elif`

Being a part of conditional statements, `elif` and `else` statements can be found nested within the bodies of other conditionals. The indentation of the block indicates which conditional the `elif` and `else` statements correspond to.

```
if am_hungry:
    if is_morning:
        print("Making pancakes! 🥞")
    else:
        print("Making soup! 🍜")
```

```
if am_hungry:
    if is_morning:
        print("Making pancakes! 🥞")
else:
    print("Making soup! 🍜")
```

Putting it All Together

Recipe for any conditional:

1. Always start with an `if`. Each `if` comes with a boolean expression to test. This expression is always tested.
2. As many `elif` statements as desired. Each comes with a boolean expression. Each expression only tested if all previous are `False`.
3. An `else` statement, or not. No boolean expression provided. Body executed if all previous expressions are `False`.

case/match

`case` / `match`: Another Way to Choose

What to do at a traffic light:

```
if traffic_light == "red":
        print("Stop!")
elif traffic_light == "yellow":
        print("Slow down.")
elif traffic_light == "green":
        print("Proceed carefully.")
```

Perfectly valid code, but a conditional requires *studying* to understand.

`case` / `match`: Another Way to Choose

What to do at a traffic light, take two:

```
match traffic_light:
    case "red":
        print("Stop!")
    case "yellow":
        print("Slow down.")
    case "green":
        print("Proceed carefully.")
```

- `match` allows us to compare an expression's value to several different cases.
- Each `case` gives a value to compare to and a block of code to execute if there's a match.

Multiple Matches & Default Cases

```
match status:
    case 200:
        print("The request worked, here's the page you wanted.")
    case 301:
        print("What you were looking for has been moved, but here's the link to the new spot.")
    case 403 | 404:
        print("You asked for something you can't have.")
    case 500:
        print("Something went wrong. Sorry!")
    case _:
        print("Something complicated happened.")
```

- Use `|` to provide multiple options per `case`
- Use `case _` at the end to specify a default behavior.