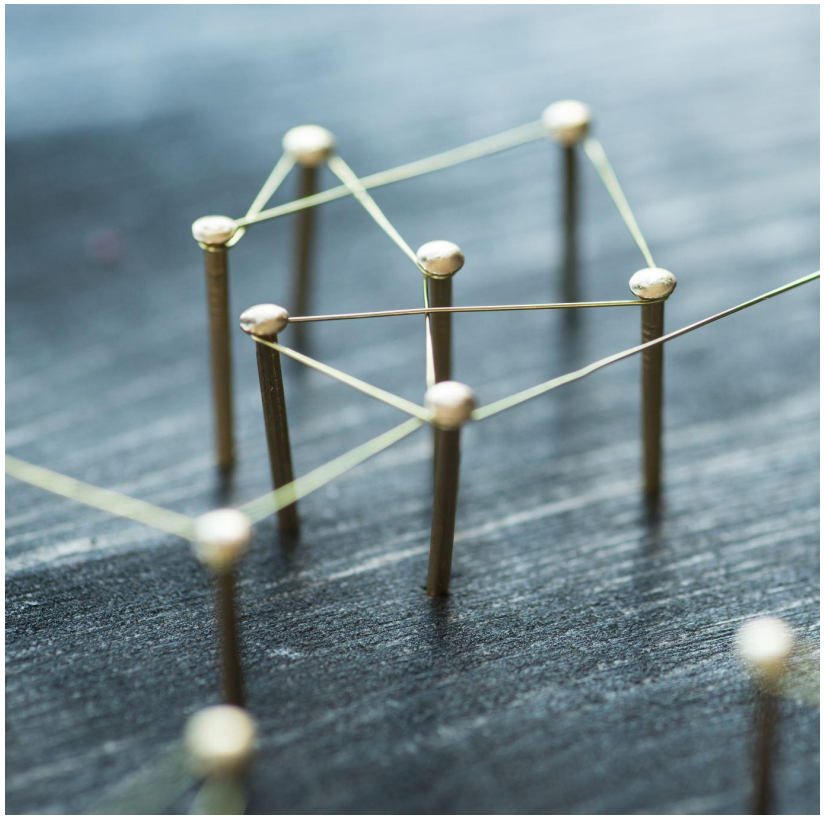
MSE 800

Professional Software Engineering



Course Guideline

- Class
- Functions
- Objects
- Methods
- Formatting
- Conditional

Functions

Functions

- **The key to programming is abstraction**
  - – Abstraction is a process by which concepts are derived from the usage and classification of literal (“real” or “concrete”) concepts.
  - – Naming a concept is a key part of abstraction
- Example: “Hey, I often need to multiply a number by itself.
  - -I know, let’s call that squaring a number”
- **In Python, functions are used for abstracting common procedures and as building blocks**
  - – We’ll see other abstraction methods – modules and classes – later.

Using functions

- **Example:**
  - **Function call** round(x) returns the nearest int to the float value x
  - round(45.6)
    - Here round is the name of the function
    - 45.6 is the argument

Note term returns. In full, we say "When **called** with an **argument** of 45.6 the **round** function **returns** the value 46"

- **We can use functions in expressions**
  - round(4.4) + 3

Build-in functions

▪ Some built-in functions:

- – round(x) returns the nearest int to the float value x
- – abs(x) returns the absolute value of x
- – int(x) converts x into an int
  - If x is a float, it truncates. int(123.45)->123
  - Later we'll see x can also be a string.

▪ You'll meet lots more in due course

- – A lot of functions in **Python libraries (modules)** – see later
- – A lot of functions as **methods** – see later.

Defining new functions

```
def square(x):                  # x is called a "parameter"
    return x * x                # the "body" is indented
```

In this case, it returns a value
---The value of the function

- Used by *calling* (or *invoking*) it, e.g.

```
square(3)           # 3 is called the "argument"
square(37.5)        # Here 37.5 is the argument
square(2 + 3 * 5)   # The argument is an expression
```

- The parameter is set to the value of the argument and then the body of the function is executed

```
def print_message(message):  #
    print(message)

print_message("Hello, World!")
```

parameter

argument

Functions are recipes

- **Recipe: boil_egg**

– Half fill a saucepan with cold water. Place egg in water. Put the saucepan on the stove, with an element set on high. When water boils, turn to low and boil for 3 minutes. Remove egg.

So Now We Can Say:
- boil (egg, 3 minutes)
- boil (potato, 20 minutes)
- boil (boot, 3 days)

Arguments

Functions: a generic recipe

Parameters

Generic recipe: boil(food, duration)

– Half fill a saucepan with cold water. Place food in water. Put the saucepan on the stove, with an element set on high. When water boils, turn to low and boil for the duration. Remove food.

– return food # The return value from the function

What type is the parameter?

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

In many languages we have to specify the parameter type

- e.g. specify whether we are squaring *ints* or *floats*
- That restricts the allowable argument types

Examples

Python has "Duck Typing"

- "If it walks like a duck and quacks like a duck, it's a duck"
- In this case: if the argument allows x * x, it's OK
  - If not, it crashes when we run it

So you can square *ints* and *floats*

- And any other objects we might define that allow '*'
  - We'll do more on this later (future class)

In Java:

```java
public int square(int number) {
    return number * number;
}

public double square(double number) {
    return number * number;
}
```

In Python:

don't need to specify types for

```python
def square(number):
    return number * number
```

In Python 3.5 and later:

use type hints to suggest the expected data types

```python
def square(number: float) -> float:
    return number * number
```

Another function definition

Local variable

```
def fahrenheit(degrees_c):
    degrees_f = (9.0 / 5.0) * degrees_c + 32.0
    return degrees_f


print(fahrenheit(0))    # What ar32.0r do we get?
print(fahrenheit(100.0)) # What 212.0er here?
print(fahrenheit(232 + 7.0 / 9.0)) # Ar451.0ere?
print(fahrenheit("Fred")) # What TypeErrors do?
```

Note multiline body. All lines are indented by the same amount.

Also note the local variable.

Some Simple Definitions:

Multiline body: A code block that spans multiple lines. All lines are indented by the same amount

A Code Block is a section of code that is grouped together and is meant to be executed as a unit.

Local variables

```
def fahrenheit(degrees_c):
    degrees_f = (9.0 / 5.0) * degrees_c + 32.0
    return degrees_f


print(fahrenheit(0))    # What answer do we get?
print(fahrenheit(100.0)) # What answer here?
print(fahrenheit(232 + 7.0 / 9.0)) # And here?
print(fahrenheit("Fred")) # What does this do?
```

- degrees_f is a "local variable" of the fahrenheit function
- Goes in a new dictionary belonging to that function
  - That dictionary exists only while the function is running
    - So the variable disappears when the function returns
- We say the scope of a local variable is the body of the function in which it is used
  - Scope is where a variable can be "seen" from

When do I use functions?

- Always!
- Programming is the art of breaking a problem into small "obviously correct" functions
  - "Divide and conquer"
- Each can be separately debugged
  - To "debug" is to remove the "bugs", i.e., errors, from a program
- Most functions should be less than 10 lines
- No function may be longer than 40 lines in MSE 800
  - Break big functions into smaller functions

The *two sorts* of functions

Procedures ("Write a function that prints…")

- **Don't** return a value to caller
  - No return statement (in Python they implicitly return None)
- **Do** print output (or write files etc)
- Names start with a verb
  - *print_table, display_summary*
- Called as, e.g.
  - `print_table(names, marks)`
  - `display_summary(data)`

Real functions ("Write a function that returns…")

- **Do** return a value to caller
  - Must have a return statement
- **Don't** print output or write files (usually)
- Names are nouns
  - *standard_error, max_rainfall*
- Called as, e.g.,
  - `error = standard_error(data)`
  - `print(max_rainfall(data))`

Class

What is a Class?

- A class is a blueprint for creating objects. It provides definitions for an object's attributes (variables) and behaviors (methods).

What is an Object?

- An object is an instance of a class that has the structure and behavior defined by the class.

Objects

- We've seen that everything in Python is an object
  - *int* objects, *float* objects, *str* objects, *function* objects ...
- An object contains data
  - The *value*, for *int* and *float* objects
  - The sequence of characters, for *str* objects
  - The Python code, for *function* objects
- But wait, there's more ....

A rudimentary Colour class

keyword

Class name

```
class Colour:
    """ An empty class -- all it has is a name """ pass

# ============================================
def display(colour):
    """ Function to print a given colour """
    print("Colour: ", colour.red, colour.green, colour.blue)



colour = Colour()          #  Call the constructor to make a Colour
colour.red = 255           #  Give it some attributes
colour.green = 123
colour.blue = 53

display(colour)
```

Style note: class names start with a capital letter without space, use *CamelCase*

#1: initialisation

```
class Colour:
    def __init__(self, red, green, blue):
        self.red = red
        self.green = green
        self.blue = blue


def display(colour):
    print("Colour: ", colour.red, colour.green, colour.blue)


colour = Colour(255, 123, 53)
display(colour)
```

initializer/constructor

Variables that belong to an *instance* of a class (i.e., an object). Also known as *fields*.

Instance of Colour class

Improvement #2: add a method

```
1   class Colour:
2       def __init__(self, red, green, blue):
3           self.red = red
4           self.green = green
5           self.blue = blue
6
7       def display(self):
8           print("Colour: ", self.red, self.green, self.blue)
9
10
11  colour = Colour(255, 123, 53)
12  colour.display()
```

NB: *convention* to use self as the first parameter to all methods

Not function, method

Each object or *instance* has its own instance variables;
Methods are shared by all instances of the class.

Method

Methods

- Each class of objects has a set of functions that operate on objects of that type.
- These are called methods.
- We call a method with the syntax

  objectName.methodName([argument]...), e.g. name.find('x')

- This is roughly equivalent to

  functionName(objectName, [argument]...) e.g. find_func(name, 'x')

  – i.e., calling a method of a particular object is like calling an equivalent function that takes the object as its first parameter

Method Call

```
class StringManipulator:
    def __init__(self, text):
        self.text = text

    def find_character(self, char):
        return self.text.find(char)

# Create an instance of the StringManipulator class
name = StringManipulator("example")

# Call the find_character method on the object
result = name.find_character('x')
print(result)  # Output: 1
```

Function Call

```
def find_character(string_object, char):
    return string_object.text.find(char)

# Create an instance of the StringManipulato
name = StringManipulator("example")

# Call the find_character function with the
result = find_character(name, 'x')
print(result)  # Output: 1
```

Some string methods

```
s = "hello world"
```

method	usage	output
capitalize()	print(s.capitalize())	Hello world
find(substring , begin , end)	print(s.find("world", 0, 5))	-1 *(not found)*
lower()	print(s.lower())	hello world
upper()	print(s.upper())	HELLO WORLD
startswith(prefix, start, end)	print(s.startswith("world", 6))	True
split(delimiter)	print(s.split(' '))	['hello', 'world'] __see later
format(value, value ...) s = "Hello {name}, you are {age} years old."	print(s.format(name="Alice", age=30))	Hello Alice, you are 30 years old.

More Methods: https://docs.python.org/3/library/stdtypes.html#string-methods

String method example program

A program to read a full name like "natalie ng", break it into two components, correctly capitalize each one, and print a "Hi" message. Handles mixed case, e.g. "nATAlie nG".

```
full_name = input("Enter your full name: ")
pos_of_space = full_name.find(' ')
first_name = full_name[0:pos_of_space]
last_name = full_name[pos_of_space+1:]
corrected_first_name = first_name.capitalize()
corrected_last_name = last_name.capitalize()
print("Hi", corrected_first_name, corrected_last_name)
```

Note: the notation s[start:] is shorthand for s[start : len(s)].
len(s) is the length of the string s.

Formatting complicated output

- To construct strings (e.g. for output), we can do things like:

  s = "Name: " + name + ", Height: " + height

- But: can't control the number of output digits of float values

- The format method of a string achieves this much more easily:

  template = "Name: {0}, height: {1:.2f}"

  s = template.format(name, height)

```
name = "Isabel"
height = 2.445666
s = "Name: {1:.2f}, height: {0}"
.format(name, height)
print(s)
```

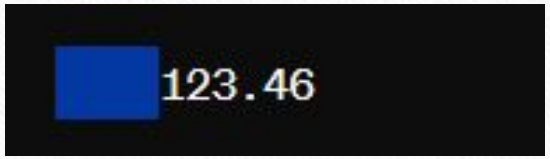
or just s = "Name: {0}, height: {1:.2f}".format(name, height)

Formatting (cont'd)

– We'll do only some simple cases: {n:w.pf} or {n:.pf} or {n:w} or {n}

- 'n' is the argument index.
- 'w' is the field width, which is the fixed-length number of characters to be produced.
- 'p' is the number of digits after the decimal point.
- 'f' indicates that the number is formatted as a fixed-point number.

```
number = 123.456
template = "{0:10.2f}"
formatted_string = template.format(number)
print(formatted_string)
```



4 places

More formatting examples

1. https://docs.python.org/3.6/library/string.html#formatspec
2. https://quiz2018.csse.canterbury.ac.nz/mod/resource/view.php?id=2246

- Conditionals

The boolean type ('bool')

NB: Equality testing is done with "==", not "=" (which is an assignment).

- A boolean expression evaluates to either True or False.
- A boolean variable is either True or False.
- Get booleans by:
  1. Testing relationships using comparison operators
     <, <=, >, >=, ==, !=, is not, not in
  2. Calling functions or methods that return booleans,
     e.g. startswith, endswith
  3. Combining booleans with logical operators, e.g. and, or, not

Truth tables

Since boolean values are either True or False, it's easy to make a table of the value of some boolean expression for all possible parameter values.

– Called a *truth* table

For example, the truth tables for *or*, *and*, and *not* are:

a	b	a or b	a and b	not a
False	False	False	False	True
False	True	True	False	True
True	False	True	False	False
True	True	True	True	False

Operator precedence

- **
- *, /, %
- +, -
- Shifts and bitwise operations (not in MSE 800)
- All comparison operators (all same precedence)
- not
- and
- or

High Precedence

Low Precedence

More details:
https://docs.python.org/3/reference/expressions.html#operator-precedence

Chaining comparison operators

- Suppose we want to check if an *int* i is in the range of low to high inclusive
- In most languages, we would write: (i >= low) and (i <= high)
- Python allows the shorthand: low <= i <= high
- But use this operator chaining only in the usual mathematical ways or it might surprise you, e.g.:
  - 5 < 10 == False evaluates to False
  - 5 < 10 == True also evaluates to False!
  - **Reason**:
    - They are shorthands for (5 < 10) and (10 == False) and (5 < 10) and (10 == True)
    - Objects of different types (*int* and *bool*) usually test unequal (but don't do it!)

Temporary boolean variables

- Consider checking if a (row, column) specification for a square on a chessboard is valid. Could implement as:

```
if not ((row >= 0 and row < 8) and (column >= 0 and column < 8)):
    print("Invalid square")
```

- But more readable as

```
is_valid_row = row >= 0 and row < 8
is_valid_column = column >= 0 and column < 8
if not (is_valid_row and is_valid_column):
    print("Invalid square")
```

- Or use de Morgan to rewrite *if* statement as

```
if not is_valid_row or not is_valid_column:
    print("Invalid square")
```

Style rule

Use names beginning with prefixes like *is_* or *can_* for boolean variables and functions

– e.g. is_valid, is_end_of_file, can_move ...

if statements

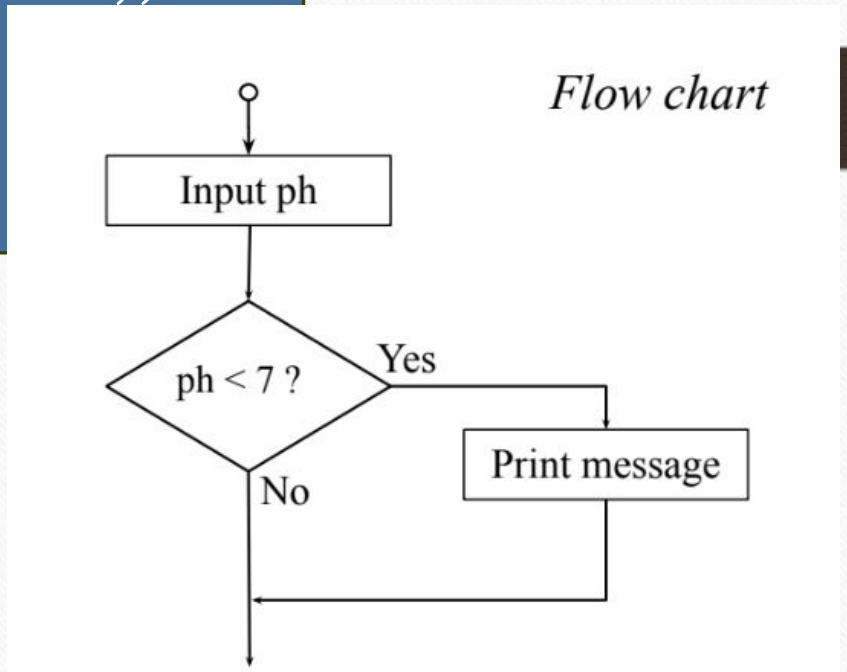
- Syntax (the most general form):

```
if bool_expression:
    block
elif bool_expression:
    block
elif bool_expression:
    block
…
else:
    block
```

A block is an indented sequence of statements

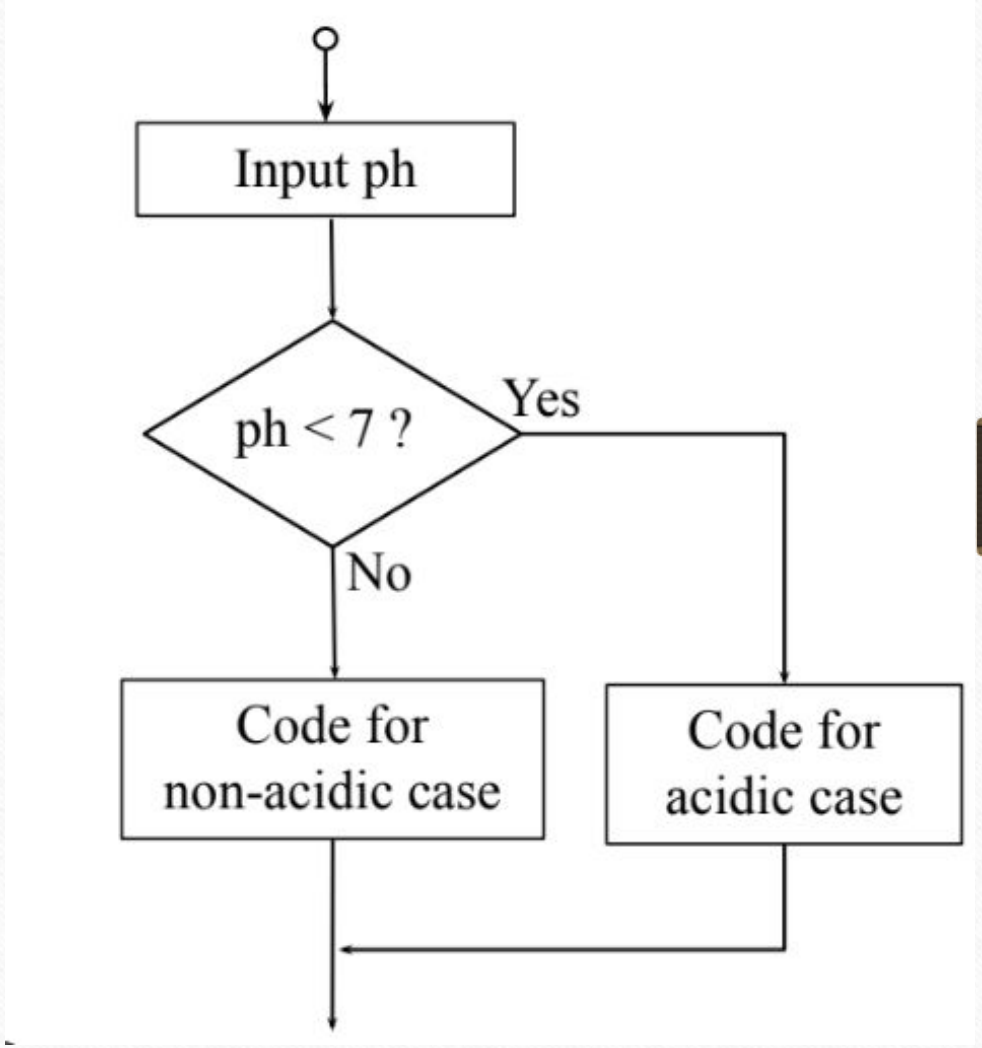
A basic if statement

```
substance = input("What substance? ")
ph = float(input("Enter the measured pH: "))
if ph < 7.0:
    print(substance + " is acidic")
```

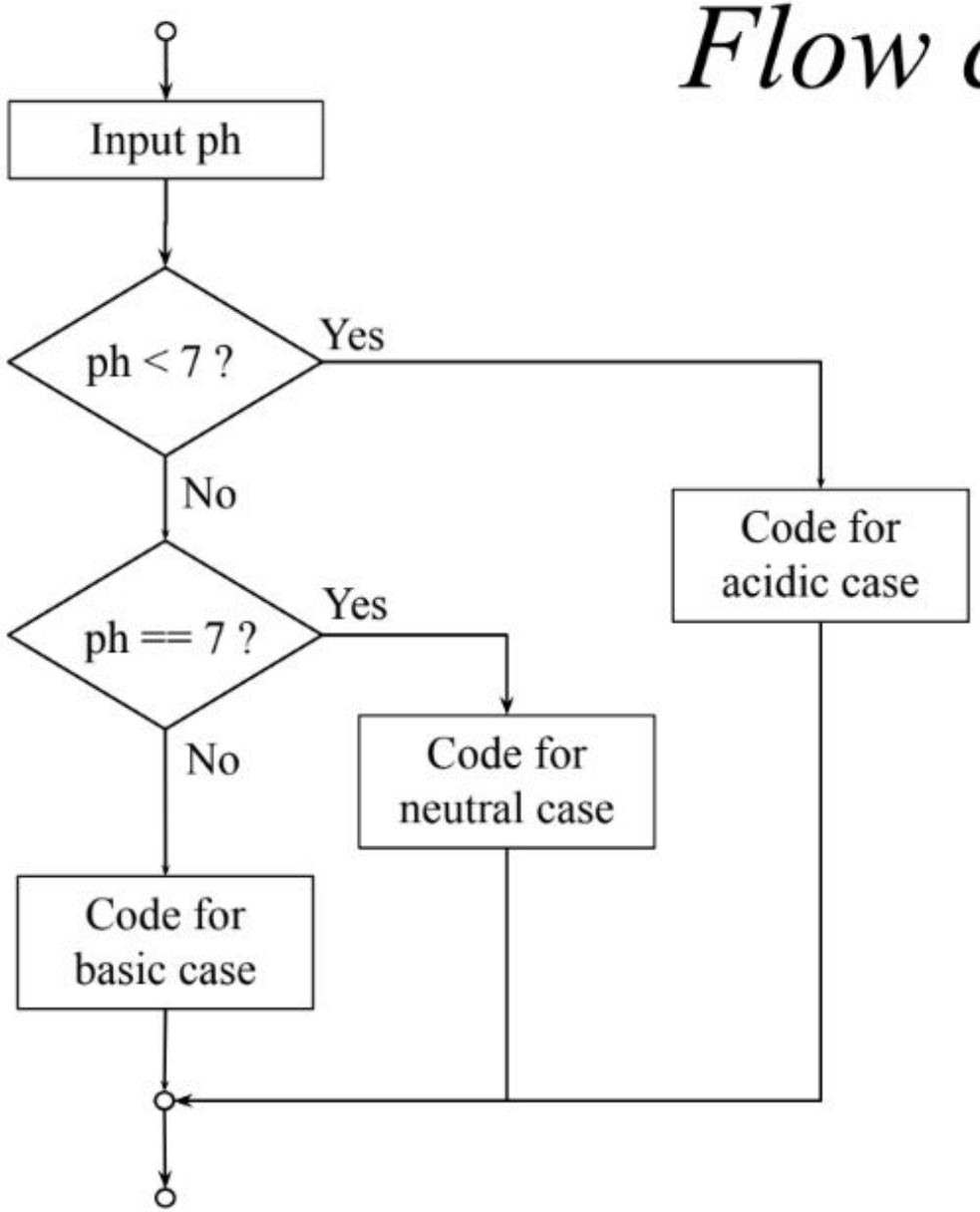


The *else* part

```
substance = input("What substance? ")
ph = float(input("Enter the measured pH: "))
if ph < 7.0:
    print(substance + " is acidic")
    print("Be careful with that!")
else:
    print(substance + " is not acidic")
    print("But that doesn't mean it's safe!")
```



Flow chart



- It is clear that only one of the three blocks can be executed.
  - i.e., cases are all *mutually exclusive*.

Using elifs

```
substance = input("What substance? ")
ph = float(input("Enter the measured pH: "))
if ph < 7.0:
    print(substance + " is acidic")
    print("Be careful with that!")
elif ph == 7.0:
    print(substance + " is neutral")
else:
    print(substance + " is basic")
    print("It might be caustic!")
```

```
substance = input("What substance? ")
ph = float(input("Enter the measured pH: "))

if ph < 7.0:
    print(substance + " is acidic")
    print("Be careful with that!")
else:
    if ph == 7.0:
        print(substance + " is neutral")
    else:
        print(substance + " is basic")
        print("It might be caustic!")
```

Pseudocode

- When writing programs we often draft the general algorithm we will use in pseudocode
  - Shows the main code blocks, if statements, and loops
  - Omits much of the coding details

```
Input a, b and c
if a is 0:
        print("Not a quadratic")
else:
        Compute the discriminant b² – 4ac
        if discriminant is not negative:
                Compute and print roots
        else:
                print("Roots are imaginary")
```

Exercise

Temperature converter

For this project, I want to build a temperature converter that transforms user-entered temperatures between Fahrenheit and Celsius. The input for Fahrenheit temperatures should start with an uppercase 'F', and for Celsius, it should start with an uppercase 'C'. Hence, the project needs to include validation and interpretation of user input.

If the input is in Fahrenheit (e.g., 'F51'), the program should convert it to Celsius, rounding to two decimal places, and output: "F51 degrees Fahrenheit is converted to XX.XX degrees Celsius", where 'XX.XX' is the converted temperature value. Conversely, if the input is in Celsius (e.g., 'C11'), the program should convert it to Fahrenheit, rounding to two decimal places, and output: "C11 degrees Celsius is converted to YY.YY degrees Fahrenheit", where 'YY.YY' is the converted temperature value.

Should the user enter an incorrect format or use the wrong prefix, the program should prompt them with: "Invalid input. Please enter the temperature with the correct 'C' or F' prefix."

```python
def convert_to_celsius(degrees_f):
    """Convert Fahrenheit to Celsius, rounding to two decimal places."""
    return round((degrees_f - 32) * 5 / 9, 2)

def convert_to_fahrenheit(degrees_c):
    """Convert Celsius to Fahrenheit, rounding to two decimal places."""
    return round((degrees_c * 9 / 5) + 32, 2)

def validate_input(user_input):
    """Check if the input has the correct format and extract the temperature."""
    if user_input.startswith('F') and user_input[1:].isdigit():
        return int(user_input[1:]), 'F'
    elif user_input.startswith('C') and user_input[1:].isdigit():
        return int(user_input[1:]), 'C'
    else:
        return None, None

def main():
    """Prompt user for temperature, perform conversion, and print result."""
    user_input = input("Enter the temperature (e.g., 'F51' for Fahrenheit or 'C11' for Celsius): ").strip().upper()
    temp, scale = validate_input(user_input)

    if scale == 'F':
        converted_temp = convert_to_celsius(temp)
        print(f"{user_input} degrees Fahrenheit is converted to {converted_temp:.2f} degrees Celsius.")
    elif scale == 'C':
        converted_temp = convert_to_fahrenheit(temp)
        print(f"{user_input} degrees Celsius is converted to {converted_temp:.2f} degrees Fahrenheit.")
    else:
        print("Invalid input. Please enter the temperature with the correct 'C' or 'F' prefix.")

if __name__ == "__main__":
    main()
```

Thank you