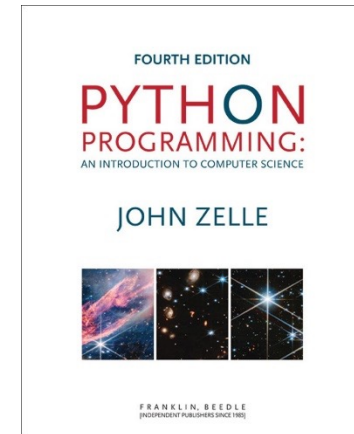


Python Programming: An Introduction to Computer Science



Chapter 6 Decision Structures



Objectives

- To understand the *simple decision* programming pattern and its implementation using a Python `if` statement.
- To understand the *two-way decision* programming pattern and its implementation using a Python `if-else` statement.



Objectives

- To understand the *multi-way decision* programming pattern and its implementation using a Python `if-elif-else` statement.
- To understand the idea of exception handling and be able to write simple exception handling code that catches standard Python run-time errors.



Objectives

- To understand the concept of Boolean expressions and the `bool` data type.
- To be able to read, write, and implement algorithms that employ decision structures, including those that employ sequences of decisions and nested decision structures.



Simple Decisions

- So far, we've viewed programs as sequences of instructions that are followed one after the other.
- While this is a fundamental programming concept, it is not sufficient in itself to solve every problem. We need to be able to alter the sequential flow of a program to suit a particular situation.



Simple Decisions

- *Control structures* allow us to alter this sequential program flow.
- In this chapter, we'll learn about *decision structures*, which are statements that allow a program to execute different sequences of instructions for different cases, allowing the program to “choose” an appropriate course of action.



Example: Temperature Warnings

- Let's return to our Celsius to Fahrenheit temperature conversion program from Chapter 2.

```
# convert.py
#         A program to convert Celsius temps to Fahrenheit
# by: Susan Computewell

def main():
    celsius = float(input("What is the Celsius temperature? "))
    fahrenheit = 9/5 * celsius + 32
    print("The temperature is", fahrenheit, "degrees Fahrenheit.")
```



Example: Temperature Warnings

- Let's say we want to modify the program to print a warning when the weather is extreme.
- Any temperature over 90 degrees Fahrenheit and lower than 30 degrees Fahrenheit will cause a hot and cold weather warning, respectively.



Example: Temperature Warnings

```
Input the temperature in degrees Celsius  
(call it celsius)
```

```
Calculate fahrenheit as 9/5 celsius + 32
```

```
Output fahrenheit
```

```
If fahrenheit > 90  
    print a heat warning
```

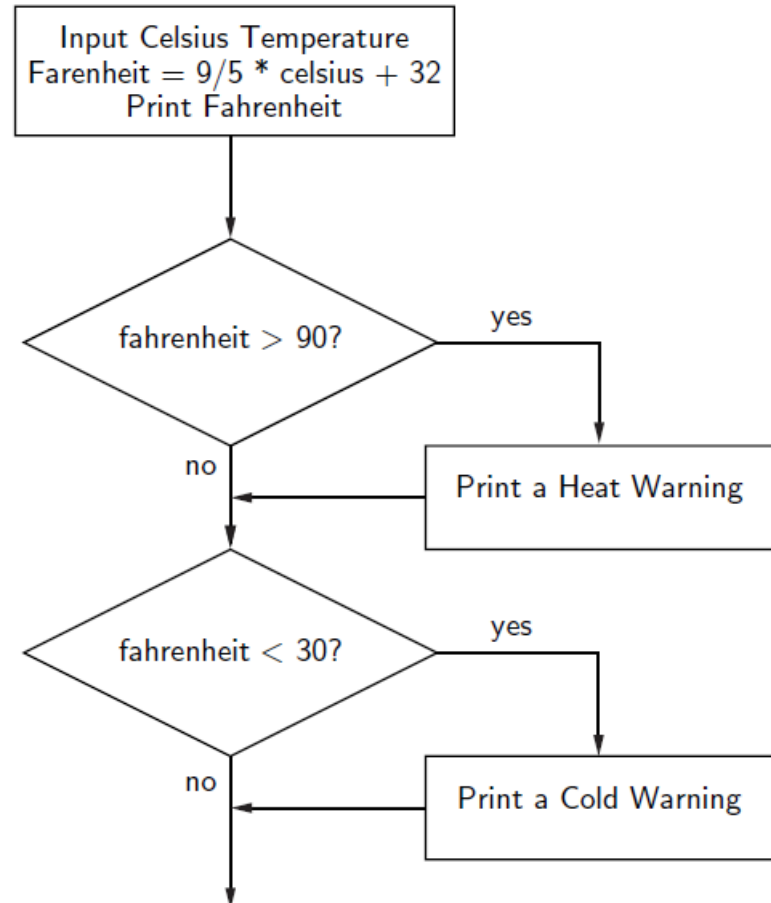
```
If fahrenheit < 30  
    print a cold warning
```



Example: Temperature Warnings

- This new algorithm has two simple *decisions* at the end. The indentation indicates that a step should be performed only if the condition listed in the previous line is true.

Example: Temperature Warnings





Example: Temperature Warnings

```
# convert2.py
#     A program to convert Celsius temps to Fahrenheit.
#     This version issues heat and cold warnings.

def main():
    celsius = float(input("What is the Celsius temperature? "))
    fahrenheit = 9 / 5 * celsius + 32
    print("The temperature is", fahrenheit, "degrees fahrenheit.")
    if fahrenheit > 90:
        print("It's really hot out there, be careful!")
    if fahrenheit < 30:
        print("Brrrrrr. Be sure to dress warmly")

main()
```



Example: Temperature Warnings

- The Python `if` statement is used to implement the decision.
- ```
if <condition>:
 <body>
```
- The body is a sequence of one or more statements indented under the `if` heading.

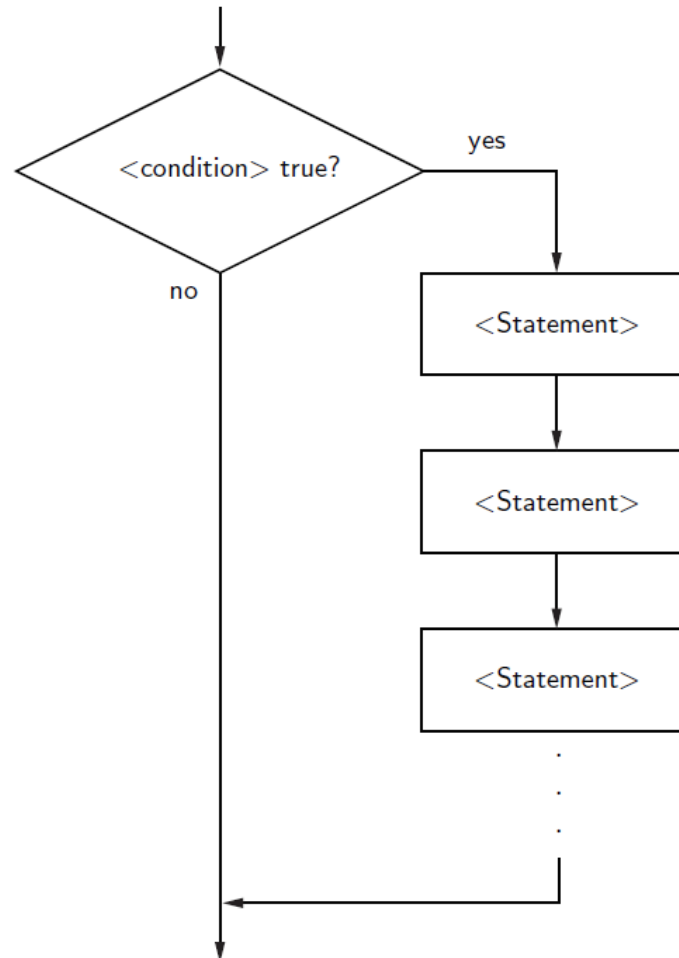


# Example: Temperature Warnings

---

- The semantics of the `if` should be clear.
  - First, the condition in the heading is evaluated.
  - If the condition is true, the sequence of statements in the body is executed, and then control passes to the next statement in the program.
  - If the condition is false, the statements in the body are skipped, and control passes to the next statement in the program.

# Example: Temperature Warnings





# Example: Temperature Warnings

---

- The body of the `if` either executes or not depending on the condition. In any case, control then passes to the next statement after the `if`.
- This is a *one-way* or *simple* decision.





# Forming Simple Conditions

---

- What does a condition look like?
- At this point, let's use simple comparisons.
- `<expr> <relop> <expr>`
- `<relop>` is short for *relational operator*



# Forming Simple Conditions

| Python | Mathematics | Meaning                  |
|--------|-------------|--------------------------|
| <      | <           | Less than                |
| <=     | ≤           | Less than or equal to    |
| ==     | =           | Equal to                 |
| >=     | ≥           | Greater than or equal to |
| >      | >           | Greater than             |
| !=     | ≠           | Not equal to             |



# Forming Simple Conditions

---

- Notice the use of `==` for equality. Since Python uses `=` to indicate assignment, a different symbol is required for the concept of equality.
- A common mistake is using `=` in conditions!



# Forming Simple Conditions

---

- Conditions may compare either numbers or strings.
- When comparing strings, the ordering is *lexicographic*, meaning that the strings are sorted based on the underlying Unicode. Because of this, all upper-case Latin letters come before lower-case letters. (“Bbbb” comes before “aaaa”)



# Forming Simple Conditions

---

- Conditions are based on *Boolean* expressions, named for the English mathematician George Boole.
- When a Boolean expression is evaluated, it produces either a value of *True* (meaning the condition holds), or it produces *False* (it does not hold).
- Some computer languages use 1 and 0 to represent “true” and “false”. Python does NOT.



# Forming Simple Conditions

---

- Boolean conditions are of type `bool` and the Boolean values of true and false are represented by the literals `True` and `False`.

```
>>> 3 < 4
```

```
True
```

```
>>> 3 * 4 < 3 + 4
```

```
False
```

```
>>> "hello" == "hello"
```

```
True
```

```
>>> "Hello" < "hello"
```

```
True
```



# Example: Conditional Program Execution

---

- There are several ways of running Python programs.
  - Some modules are designed to be run directly. These are referred to as programs or scripts.
  - Others are made to be imported and used by other programs. These are referred to as libraries.
  - Sometimes we want to create a hybrid that can be used both as a stand-alone program and as a library.



# Example: Conditional Program Execution

---

- When we want to start a program once it's loaded, we include the line `main()` at the bottom of the code.
- Since Python evaluates the lines of the program during the import process, our current programs also run when they are imported into an interactive Python session or into another Python program.





# Example: Conditional Program Execution

---

- Generally, when we **import** a module, we don't want it to execute!
- In a program that can be either run stand-alone or loaded as a library, the call to `main` at the bottom should be made conditional, e.g.

```
if <condition>:
 main()
```



# Example: Conditional Program Execution

---

- Whenever a module is imported, Python creates a special variable in the module called `__name__` to be the name of the imported module.

- Example:

```
>>> import math
>>> math.__name__
'math'
```



# Example: Conditional Program Execution

---

- When imported, the `__name__` variable inside the `math` module is assigned the string `'math'`.
- When Python code is run directly and *not* imported, the value of `__name__` is `'__main__'`. E.g.:

```
>>> __name__
'__main__'
```



# Example: Conditional Program Execution

---

- To recap: if a module is imported, the code in the module will see a variable called `__name__` whose value is the name of the module.
- When a file is run directly, the code will see the value `'__main__'`.
- We can change the final lines of our programs to:  

```
if __name__ == '__main__':
 main()
```
- Virtually every Python module ends this way!



# Two-Way Decisions

---

- In Python, a two-way decision can be implemented by attaching an `else` clause onto an `if` clause.
- This is called an `if-else` statement:

```
if <condition>:
 <statements>
else:
 <statements>
```



# Two-Way Decisions

---

- When Python encounters this structure, it first evaluates the condition. If the condition is true, the statements under the `if` are executed.
- If the condition is false, the statements under the `else` are executed.
- In either case, the statements following the `if-else` are executed after either set of statements are executed.



# Multi-Way Decisions

---

- Imagine if we needed to make a five-way decision using nesting. The `if-else` statements would be nested four levels deep!
- There is a construct in Python that achieves this, combining an `else` followed immediately by an `if` into a single `elif`.



# Multi-Way Decisions

---

```
■ if <condition1>:
 <case1 statements>
elif <condition2>:
 <case2 statements>
elif <condition3>:
 <case3 statements>

...
else:
 <default statements>
```





# Multi-Way Decisions

---

- This form sets off any number of mutually exclusive code blocks.
- Python evaluates each condition in turn looking for the first one that is true. If a true condition is found, the statements indented under that condition are executed, and control passes to the next statement after the entire `if-elif-else`.
- If none are true, the statements under `else` are performed.
- The `else` is optional. If there is no `else`, it's possible no indented block would be executed.



# Exception Handling

---

- The `try` statement has the following form:

```
try:
 <body>
except <ErrorType>:
 <handler>
```

- When Python encounters a `try` statement, it attempts to execute the statements inside the body.
- If there is no error, control passes to the next statement after the `try...except`.



# Exception Handling

---

- If an error occurs while executing the body, Python looks for an except clause with a matching error type. If one is found, the handler code is executed.
- The original program generated this error with a negative discriminant:

```
Traceback (most recent call last):
 File "C:\Documents and Settings\Terry\My
Documents\Teaching\W04\CS120\Textbook\code\chapter3\quadratic.py", line 21, in -toplevel-
main()
 File "C:\Documents and Settings\Terry\My Documents\Teaching\W04\CS
120\Textbook\code\chapter3\quadratic.py", line 14, in main
 discRoot = math.sqrt(b * b - 4 * a * c)
ValueError: math domain error
```



# Exception Handling

---

- Instead of crashing, the exception handler prints a message indicating that there are no real roots.
- The `try...except` can be used to catch *any* kind of error and provide for a graceful exit.
- A single `try` statement can have multiple `except` clauses.



# Exception Handling

---

- The multiple `excepts` act like `elifs`. If an error occurs, Python will try each `except` looking for one that matches the type of error.
- The bare `except` at the bottom acts like an `else` and catches any errors without a specific match.
- If there was no bare `except` at the end and none of the `except` clauses match, the program would still crash and report an error.



# Exception Handling

---

- Exceptions themselves are a type of object.
- If you follow the error type with an identifier in an `except` clause, Python will assign to that identifier the actual exception object.



# Study in Design: Max of Three

---

- Now that we have decision structures, we can solve more complicated programming problems. The negative is that writing these programs becomes harder!
- Suppose we need an algorithm to find the largest of three numbers.



# Study in Design: Max of Three

---

```
def main():
 x1 = float(input("Please enter three values: "))
 x2 = float(input("Please enter three values: "))
 x3 = float(input("Please enter three values: "))

 # missing code sets max to the value of the largest

 print("The largest value is", maxval)
```





# Strategy 1: Compare Each to All

---

- This looks like a three-way decision, where we need to execute *one* of the following:

```
maxval = x1
```

```
maxval = x2
```

```
maxval = x3
```

- All we need to do now is preface each one of these with the right condition!



# Strategy 1: Compare Each to All

---

- Let's look at the case where  $x_1$  is the largest.
- ```
if x1 >= x2 >= x3:  
    maxval = x1
```
- Is this syntactically correct?
 - Many languages would not allow this *compound condition*
 - Python does allow it, though. It's equivalent to $x_1 \geq x_2 \geq x_3$.



Strategy 1: Compare Each to All

- Whenever you write a decision, there are two crucial questions:
 1. When the condition is true, is executing the body of the decision the right action to take?
 - x1 is at least as large as x2 and x3, so assigning maxval to x1 is OK.
 - Always pay attention to borderline values!!



Strategy 1: Compare Each to All

2. Secondly, ask the converse of the first question, namely, are we certain that this condition is true in all cases where x_1 is the max?
 - Suppose the values are 5, 2, and 4.
 - Clearly, x_1 is the largest, but does $x_1 \geq x_2 \geq x_3$ hold?
 - We don't really care about the relative ordering of x_2 and x_3 , so we can make two separate tests: $x_1 \geq x_2$ *and* $x_1 \geq x_3$.



Strategy 1: Compare Each to All

- We can separate these conditions with *and*!

```
if x1 >= x2 and x1 >= x3:  
    maxval = x1  
elif x2 >= x1 and x2 >= x3:  
    maxval = x2  
else:  
    maxval = x3
```

- We're comparing each possible value against all the others to determine which one is largest.



Strategy 1: Compare Each to All

- What would happen if we were trying to find the max of five values?
- We would need four Boolean expressions, each consisting of four conditions *anded* together.
- Yuck!



Strategy 2: Decision Tree

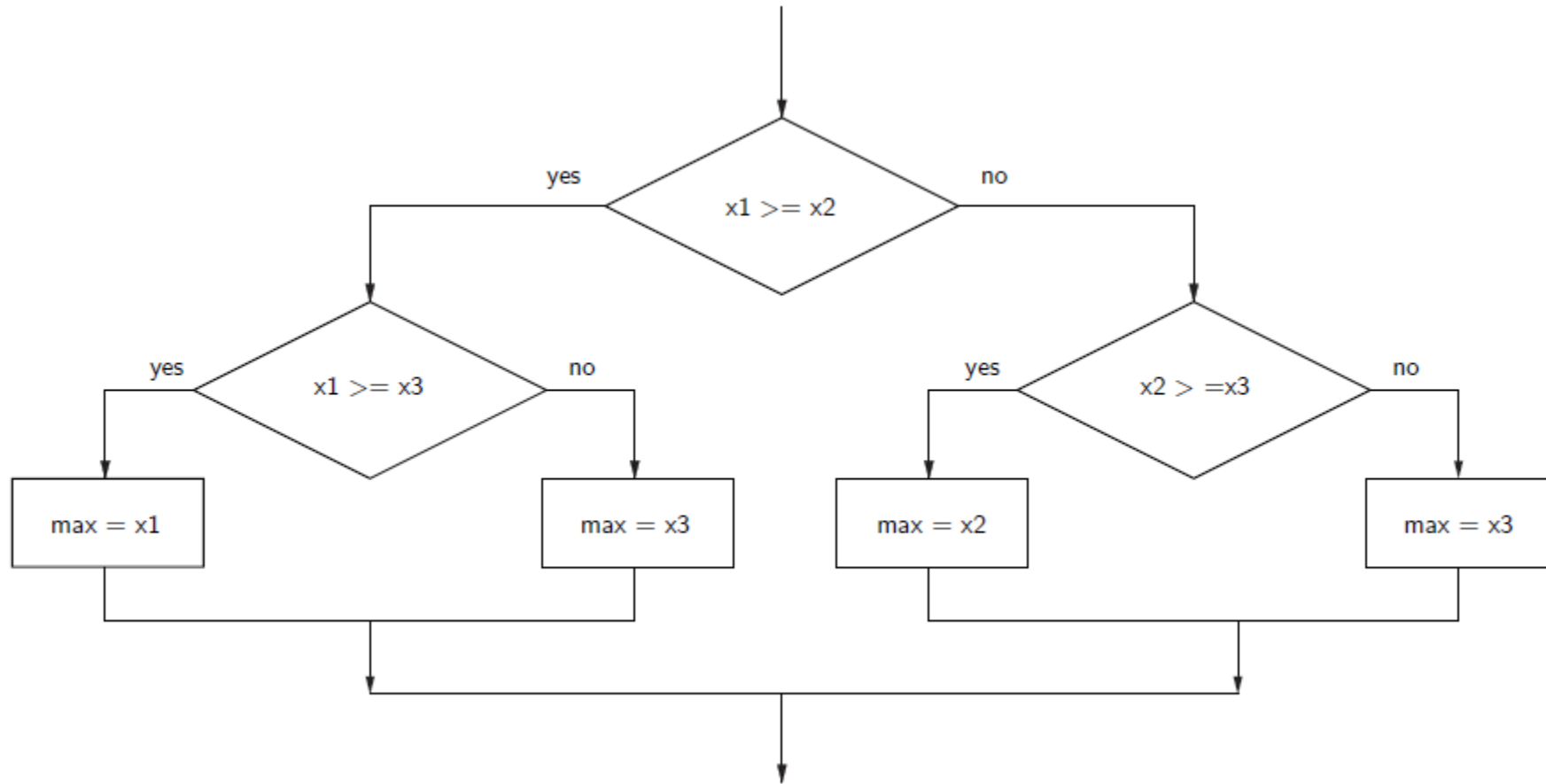
- We can avoid the redundant tests of the previous algorithm using a *decision tree* approach.
- Suppose we start with $x_1 \geq x_2$. This knocks either x_1 or x_2 out of contention to be the max.
- If the condition is true, we need to see which is larger, x_1 or x_3 .



Strategy 2: Decision Tree

```
if x1 >= x2:
    if x1 >= x3:
        maxval = x1
    else:
        maxval = x3
else:
    if x2 >= x3:
        maxval = x2
    else:
        maxval = x3
```


Strategy 2: Decision Tree





Strategy 2: Decision Tree

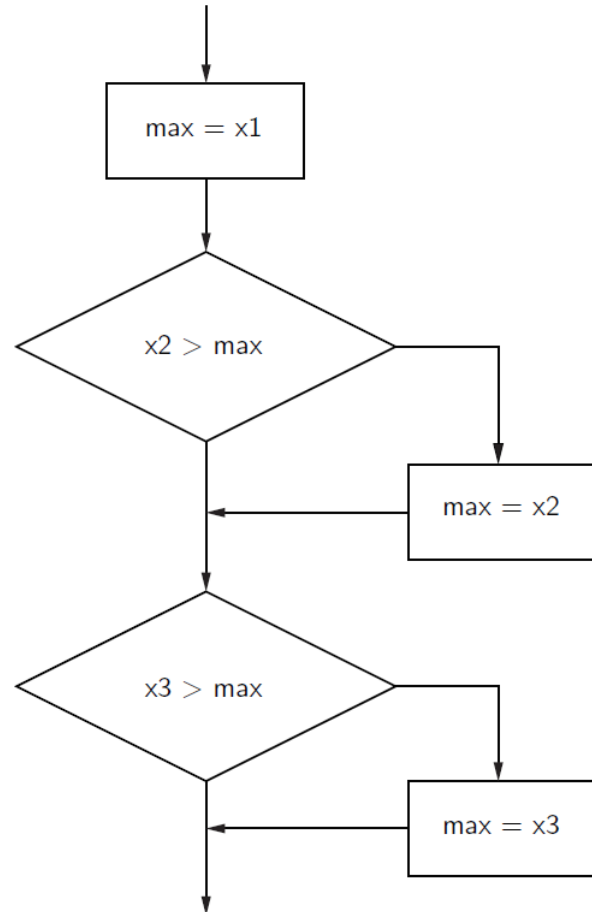
- This approach makes exactly two comparisons, regardless of the ordering of the original three variables.
- However, this approach is more complicated than the first. To find the max of four values you'd need `if-elses` nested three levels deep with eight assignment statements!

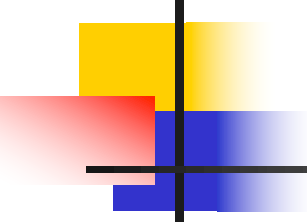


Strategy 3: Sequential Processing

- How would you solve the problem?
- You could probably look at three numbers and just *know* which is the largest. But what if you were given a list of a hundred numbers?
- One strategy is to scan through the list looking for a big number. When one is found, mark it, and continue looking. If you find a larger value, mark it, erase the previous mark, and continue looking.

Strategy 3: Sequential Processing





Strategy 3: Sequential Processing

- This idea can easily be translated into Python.

```
maxval = x1
if x2 > maxval:
    maxval = x2
if x3 > maxval:
    maxval = x3
```



Strategy 3: Sequential Programming

- This process is repetitive and lends itself to using a loop.
- We prompt the user for a number, we compare it to our current max, if it is larger, we update the max value, repeat.



Strategy 3: Sequential Programming

```
# program: maxn.py
#   Finds the maximum of a series of numbers

def main():
    n = int(input("How many numbers are there? "))

    # Set max to be the first value
    max = float(input("Enter a number >> "))

    # Now compare the n-1 successive values
    for i in range(n-1):
        x = float(input("Enter a number >> "))
        if x > max:
            max = x

    print("The largest value is", max)
```



Strategy 4: Use Python

- Python has a built-in function called `max` that returns the largest of its parameters.

- ```
def main():
 x1, x2, x3 = eval(input("Please enter three values: "))
 print("The largest value is", max(x1, x2, x3))
```





# Some Lessons

---

- There's usually more than one way to solve a problem.
  - Don't rush to code the first idea that pops out of your head. Think about the design and ask if there's a better way to approach the problem.
  - Your first task is to find a correct algorithm. After that, strive for clarity, simplicity, efficiency, scalability, and elegance.



# Some Lessons

---

- Be the computer.
  - One of the best ways to formulate an algorithm is to ask yourself how you would solve the problem if you had the sensibilities of a computer.
  - This straightforward approach is often simple, clear, and efficient enough.



# Some Lessons

---

- Generality is good.
  - Consideration of a more general problem can lead to a better solution for some special case.
  - If the max of  $n$  program is just as easy to write as the max of three, write the more general program because it's more likely to be useful in other situations.



# Some Lessons

---

- Don't reinvent the wheel.
  - If the problem you're trying to solve is one that lots of other people have encountered, find out if there's already a solution for it!
  - As you learn to program, designing programs from scratch is a great experience!
  - Truly expert programmers know when to borrow.