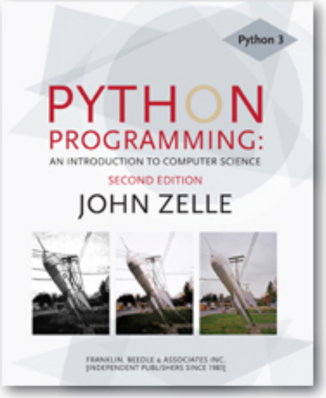
Python Programming: An Introduction to Computer Science



Chapter 7

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.

decisions_01_simple.py

- *simple decision with if*

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 *lexigraphic*, meaning that the strings are sorted based on the underlying Unicode. Because of this, all upper-case 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".

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
```

decisions_05_two_way.py

- *two-way decision with if-else*

decisions_10_multi_way.py

- *basic multi-way decision with if-elif-else*

decisions_15_multi_way_extended.py

- *extended multi-way decision with if-elif-else*

decisions_20_lookup.py

- *simple lookup coded inline*

decisions_25_lookup_in_function.py

- *simple lookup after refactored to function*

decisions_30_nested_inline.py

- *inline coding of complex decision using nested if-else*

decisions_nested_in_function.py

- *complex decision using nested if-else*
- *decision logic factored into function*
- *main() used to run multiple test cases*

decisions_40_try.py

- *Use try-except block to detect bad input*

decisions_45_raise.py

- *Use try-except block with raise to catch input error*
- *exception raised in called function*

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, x2, x3 = eval(input("Please enter three values: "))

    # missing code sets max to the value of the largest

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

Strategy 1:
Compare Each to All

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

```
max = x1
max = x2
max = 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 x1 is the largest.

-
  ```
  if x1 >= x2 >= x3:
      max = x1
  ```
- Is this syntactically correct?
  - Many languages would not allow this *compound condition*
  - Python does allow it, though. It’s equivalent to $x1 \geq x2 \geq x3$.

Strategy 1:
Compare Each to All

- Whenever you write a decision, there are two crucial questions:
  - 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 max to x1 is OK.
    - Always pay attention to borderline values!!

Strategy 1:
Compare Each to All

- Secondly, ask the converse of the first question, namely, are we certain that this condition is true in all cases where x1 is the max?
  - Suppose the values are 5, 2, and 4.
  - Clearly, x1 is the largest, but does x1 ≥ x2 ≥ x3 hold?
  - We don’t really care about the relative ordering of x2 and x3, so we can make two separate tests: x1 >= x2 *and* x1 >= x3.

Strategy 1:
Compare Each to All

- We can separate these conditions with *and*!

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

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

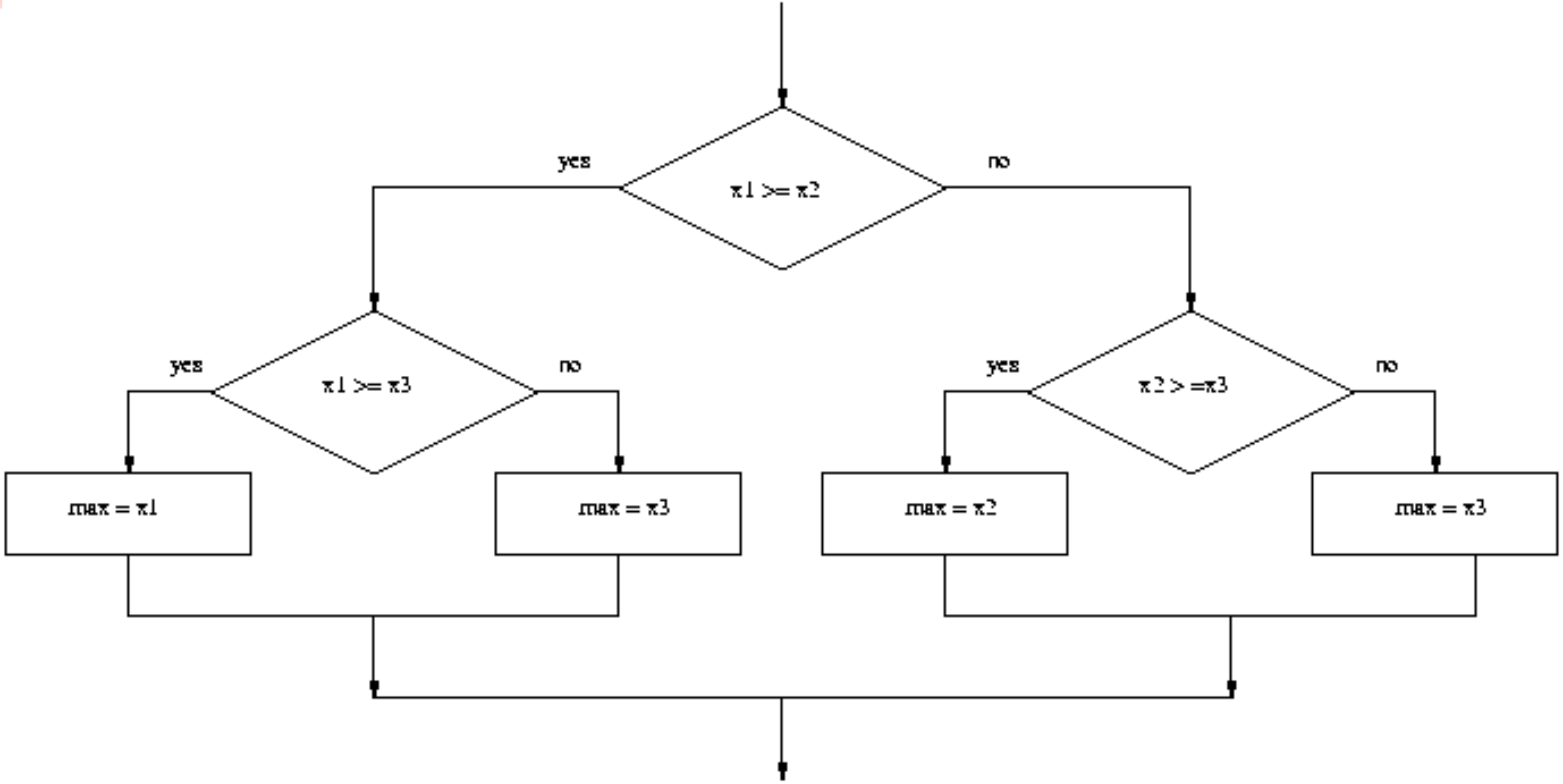
Strategy 1:
Compare Each to All

- 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 *and*ed 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 `x1 >= x2`. This knocks either `x1` or `x2` out of contention to be the max.
- If the conidition is true, we need to see which is larger, `x1` or `x3`.

Strategy 2: Decision Tree



Strategy 2: Decision Tree

-
  ```
  if x1 >= x2:
      if x1 >= x3:
          max = x1
      else:
          max = x3
  else:
      if x2 >= x3:
          max = x2
      else
          max = x3
  ```

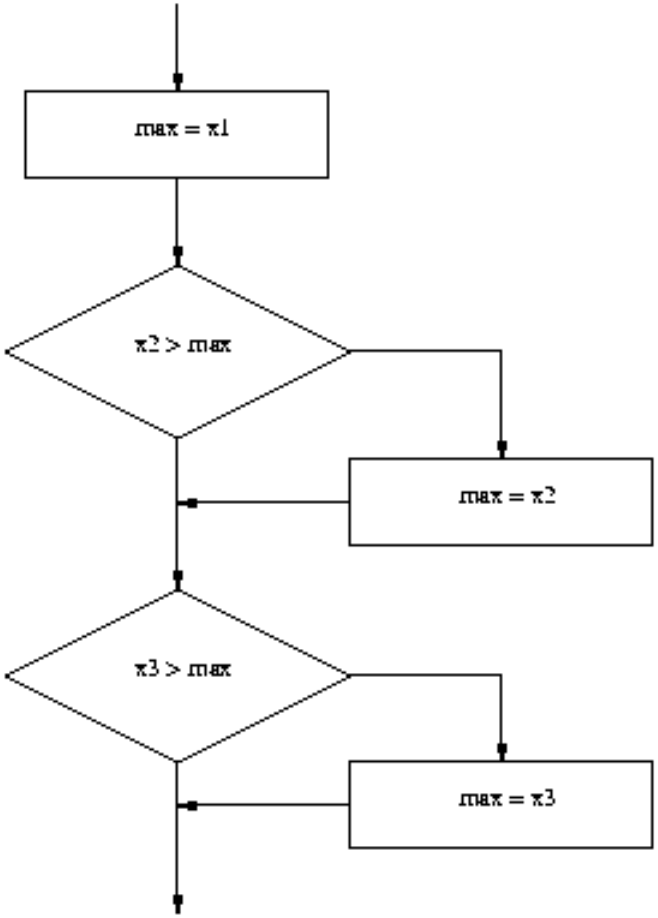
Strategy 2: Decision Tree

- 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-else`s 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.

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

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

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

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

    # Now compare the n-1 successive values
    for i in range(n-1):
        x = eval(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.
  - 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 a 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.