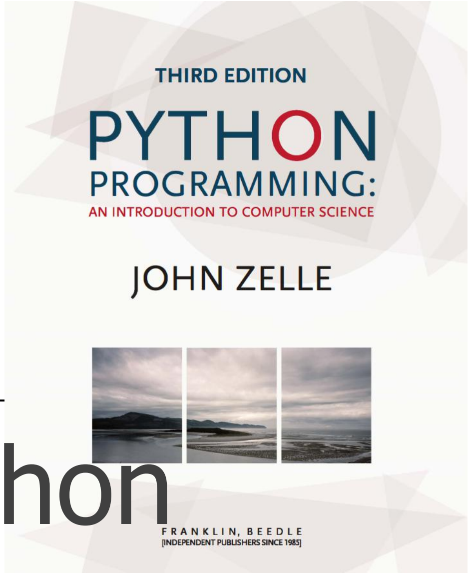


An Introduction to Python Programming

Chapter 7: Decision Structures

Objectives

- To understand **the decision programming pattern** and its implementation using a Python **`if`/ `if-else` /`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.
- To understand the concept of **Boolean expressions** and theb ool data type.

Simple Decisions

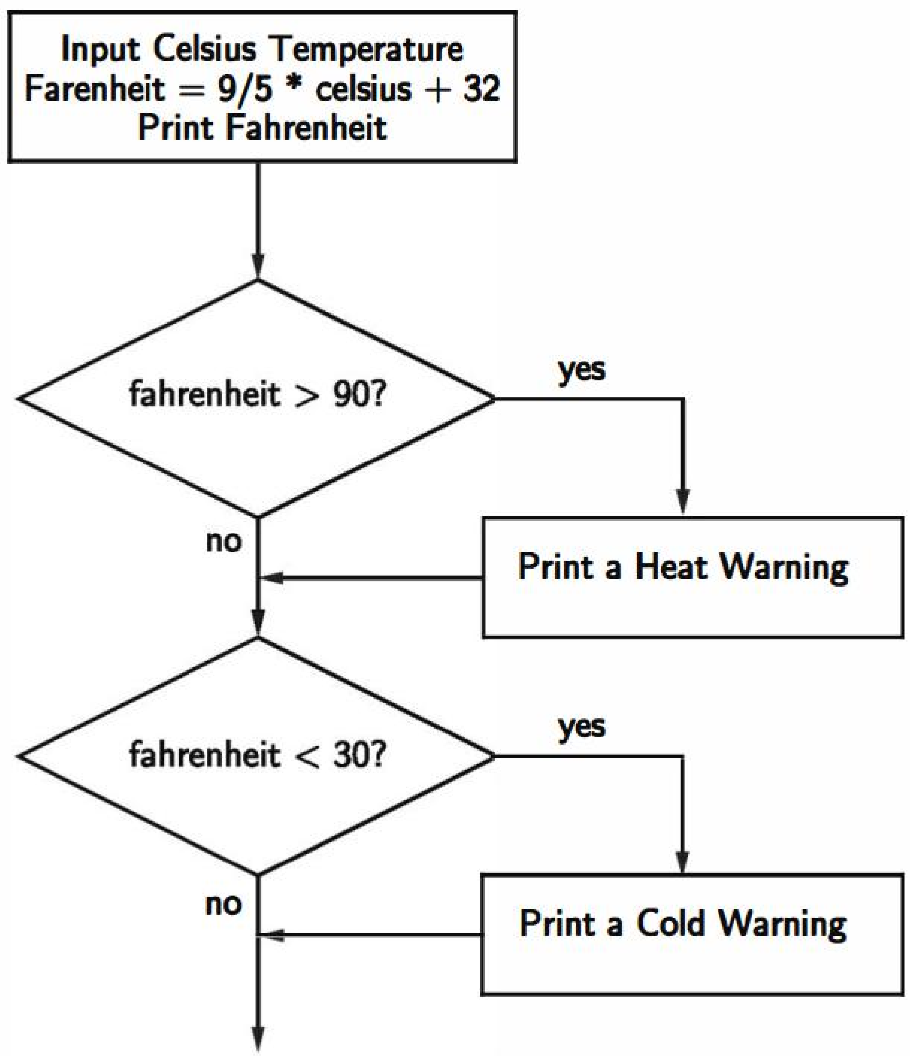
- Often, a fundamental programming is not to solve every problem. We need to be able to **alter the sequential flow** of a program to suit a particular situation.
- ***Control structures*** allow us to alter this sequential program flow.

Example:
Temperature Warnings

- Let's return to our ***convert.py***.
- Any temperature **over 90 degrees** Fahrenheit and **lower than 30 degrees** Fahrenheit will cause a hot and cold weather **warning**, respectively.

```
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



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.")

    # Print warnings for extreme temps
    if fahrenheit > 90:
        print("It's really hot out there. Be careful!")
    if fahrenheit < 30:
        print("Brrrrr. 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.
- This is a ***one-way* or *simple* decision**.

Forming Simple Conditions

- What does a **condition** look like in python?
- Let's use **simple comparisons**:

    `<expr> <relop> <expr>`

- **`<relop>`** is short for ***relational operator.*** There are **six** relational operators in Python,

Forming Simple Conditions

Python	mathematics	meaning
<	$<$	less than
<=	$\leq$	less than or equal to
==	$=$	equal to
>=	$\geq$	greater than or equal to
>	$>$	greater than
!=	$\neq$	not equal to

- Conditions may compare either **numbers or strings**.
- When comparing strings, they are sorted based on the **underlying ASCII** codes. So, all upper-case letters come before lower-case letters. ("Bbbb" comes before "aaaa")

Forming Simple Conditions

- Conditions are based on ***Boolean* expressions**.
- A Boolean expression produces ***true*** (or 1, meaning the condition holds), or it produces ***false*** (or 0, it does not hold).

```
>>> 3 < 4
True
>>> 3 * 4 < 3 + 4
False
>>> "hello" == "hello"
True
>>> "hello" < "hello"
False
>>> "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

- Most programs have a line **at the bottom** using **`main()`** to invoke the main function.
- 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.

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

Example: Conditional Program Execution

- 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

- Let’s look at the quadratic program.

```
def main():
    print("This program finds the real solutions to a quadratic\n")

    a = float(input("Enter coefficient a: "))
    b = float(input("Enter coefficient b: "))
    c = float(input("Enter coefficient c: "))

    discRoot = math.sqrt(b * b - 4 * a * c)
    root1 = (-b + discRoot) / (2 * a)
    root2 = (-b - discRoot) / (2 * a)

    print("\nThe solutions are:", root1, root2 )

main()
```

Two-Way Decisions

- when b^2*-4ac < 0*, the program tries to take the square root of a negative number, and then crashes.
- We can check for this situation and try our first attempt.

```
def main():
    print("This program finds the real solutions to a quadratic\n")
    a = float(input("Enter coefficient a: "))
    b = float(input("Enter coefficient b: "))
    c = float(input("Enter coefficient c: "))

    discrim = b * b - 4 * a * c
    if discrim >= 0:
        discRoot = math.sqrt(discrim)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2 )
```

Two-Way Decisions

- Look carefully at the program. What's wrong with it? Hint: What happens when there are **no real roots**?
- This is **almost worse** than the version that crashes, because we don't know what went wrong!
- In Python, a two-way decision can be implemented by attaching an **`else`** clause onto an **`if`** clause.

Two-Way Decisions

- This is called an **`if-else`** statement:

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

Multi-Way Decisions

- The newest program is great, but it still has some quirks!

```
>>> main()
This program finds the real solutions to a quadratic

Enter coefficient a: 1
Enter coefficient b: 2
Enter coefficient c: 1

The solutions are: -1.0 -1.0
```

  - **Double roots occur** when the discriminant is exactly **0**, and then the roots are ***–b/2a***.

Multi-Way Decisions

- We can do this with **two if-else statements**, one inside the other.
- Putting one compound statement inside of another is called ***nesting***.

```
if discrim < 0:
    print("Equation has no real roots")
else:
    if discrim == 0:
        root = -b / (2 * a)
        print("There is a double root at", root)
    else:
        # Do stuff for two roots
```

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:

```
if <condition1>:
    <case1 statements>
elif <condition2>:
    <case2 statements>
elif <condition3>:
    <case3 statements>
...
else:
    <default statements>
```

Exception Handling

- For many programs,**decision structures** are used to **protect** against rare but possible **errors**.

```
discRt = otherSqrt(b*b - 4*a*c)
if discRt < 0:
    print("No real roots.")
else:
```

- Sometimes programs get so many checks for special cases that the algorithm becomes **hard to follow**.
- Programming language designers have come up with a mechanism to **handle *exception handling*** to solve this design problem.

Exception Handling

```
def main():
    print("This program finds the real solutions to a quadratic\n")

    try:
        a = float(input("Enter coefficient a: "))
        b = float(input("Enter coefficient b: "))
        c = float(input("Enter coefficient c: "))
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2)
    except ValueError:
        print("\nNo real roots")
```

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 the **`try`**…**`except.`** If **an error occurs**, Python looks for an **except clause** with a matching **error type**. If one is found, the **handler** code is executed.

Exception Handling

- The original program **without** the **exception handling** produced the following error:

```
Traceback (most recent call last):
  File "quadratic.py", line 23, in <module>
    main()
  File "quadratic.py", line 16, in main
    discRoot = math.sqrt(b * b - 4 * a * c)
ValueError: math domain error
```

  - In the last line of this error message, **ValueError** indicates the **type of error** that was generated.

Exception Handling

- The updated version of the program **provides an except clause** to catch the ***ValueError***:

```
Enter coefficient a: 1
Enter coefficient b: 2
Enter coefficient c: 3

No real roots
```

Exception Handling

- About E**xceptions:**

```
>>>
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
>>> 2+3*x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>>
>>> '2'+2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
>>>
>>> _
```

Exception Handling

- A single `try` statement can have **multiple** `except` clauses.

```
try:
    x = int(input('please input an integer:'))
    if 30 / x > 5:
        print('Hello World!')
except ValueError:
    print('That was no valid number. Try again...')
except ZeroDivisionError:
    print('The divisor can not be zero, Try again...')
except:
    print('Handling other exceptions...')

#if __name__ == '__main__':
main()
```

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 that **identifier** the **actual exception object**.
- You can observing the **error messages** that Python prints and designing ***except clauses*** to catch and handle them.

Study in Design: Max of Three

- Suppose we need an algorithm to **find the largest** of three numbers.

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

    # missing code sets maxval to the value of the largest

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

Strategy 1:
Compare Each to All

- Let's look at the case where **x1 is the largest**.

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

- Python does allow it! But, there are two crucial questions:
  - When the condition is true, is **the executing body** of the decision act right?
  - Is this condition will be true **in all cases** where x1 is the max?(Suppose the values are 5, 2, and 4)

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
```

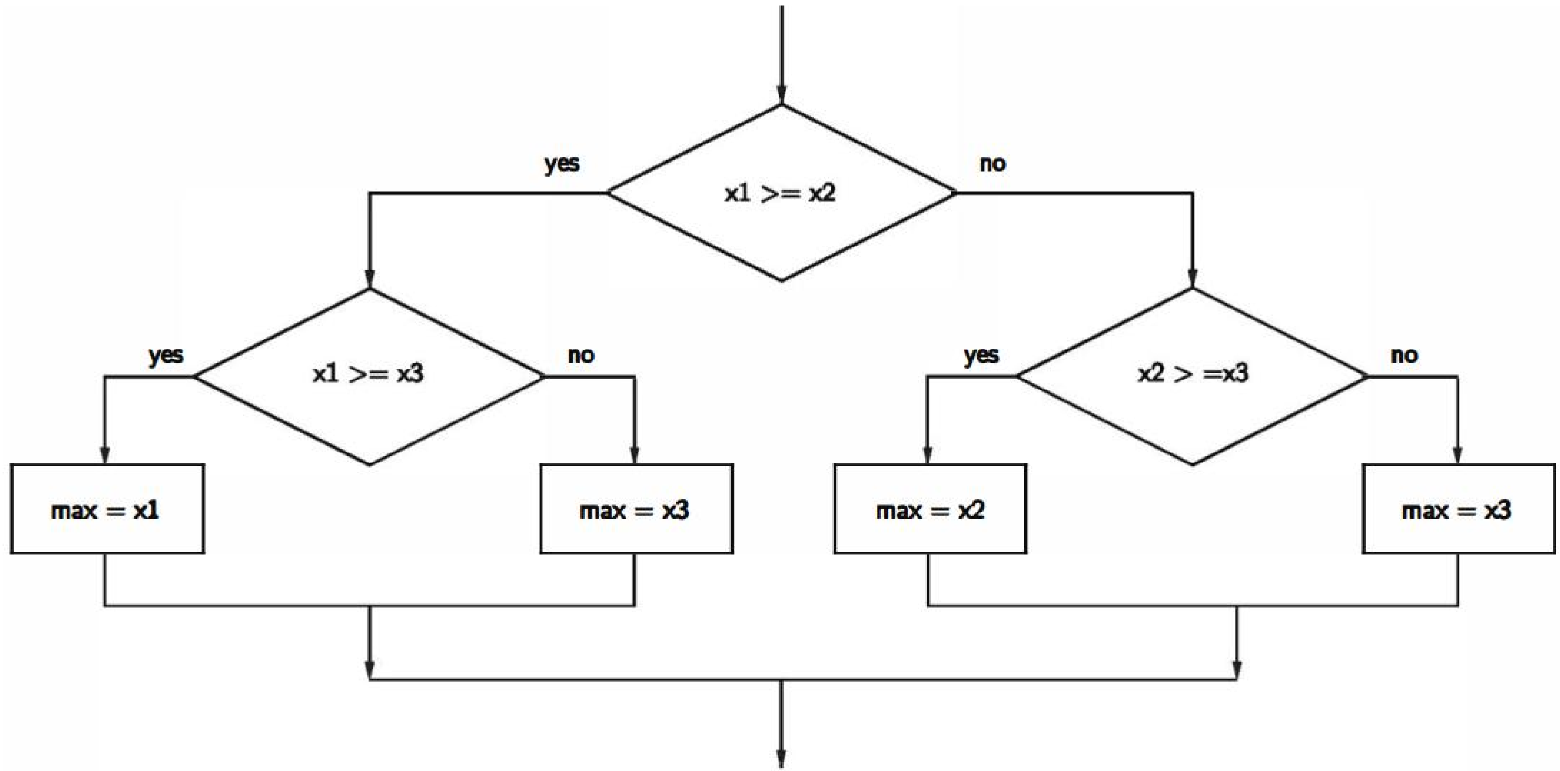
- What would happen if we were trying to **find the max of five values**?

Strategy 2: Decision Tree

- We can avoid the redundant tests of the previous algorithm using **a *decision tree*** approach.

```
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

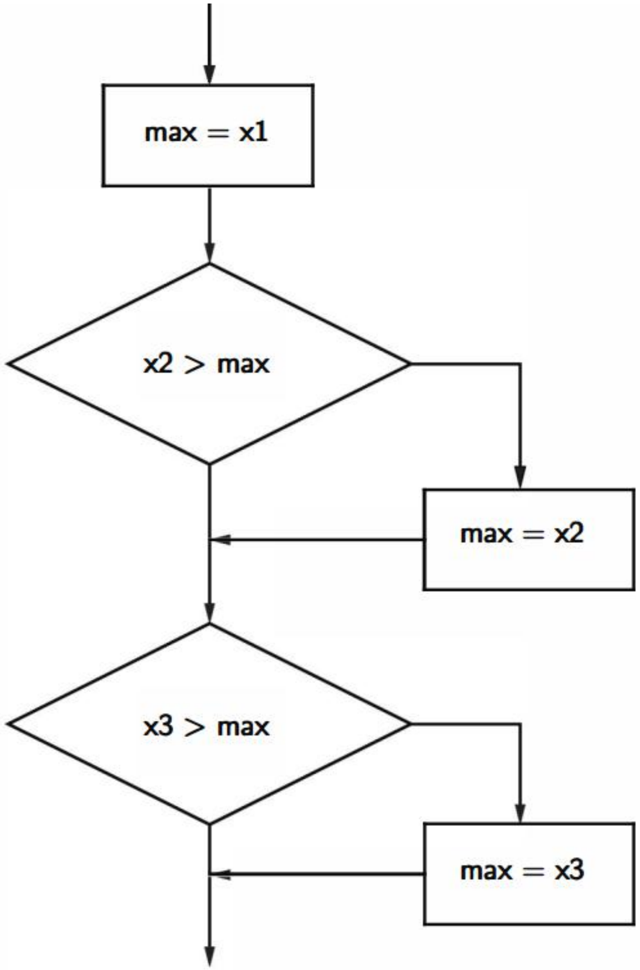
- However, this approach is **more complicated** than the first. To find the **max of four values** you'd need **`if-elses` nested three levels** deep with eight assignment statements.

Strategy 3: Sequential Processing

- If you were given **a list of a hundred numbers**,How would you solve the problem?
- Scan through the list, looking for a big number, mark it, continue looking, find a larger value, mark it, erase the previous mark, and continue looking...

Strategy 3: Sequential Processing

```
maxval = x1
if x2 > maxval:
    maxval = x2
if x3 > maxval:
    maxval = x3
if x4 > maxval:
    maxval = x4
```



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 ***maxval***, 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
    maxval = 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 > maxval:
            maxval = x

    print("The largest value is", maxval)

main()
```

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))
```

  - This version didn't require any algorithm development at all.

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.***
  - 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.
  - That way you get the maximum utility from your programming effort.

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!