Scientific Computing 272

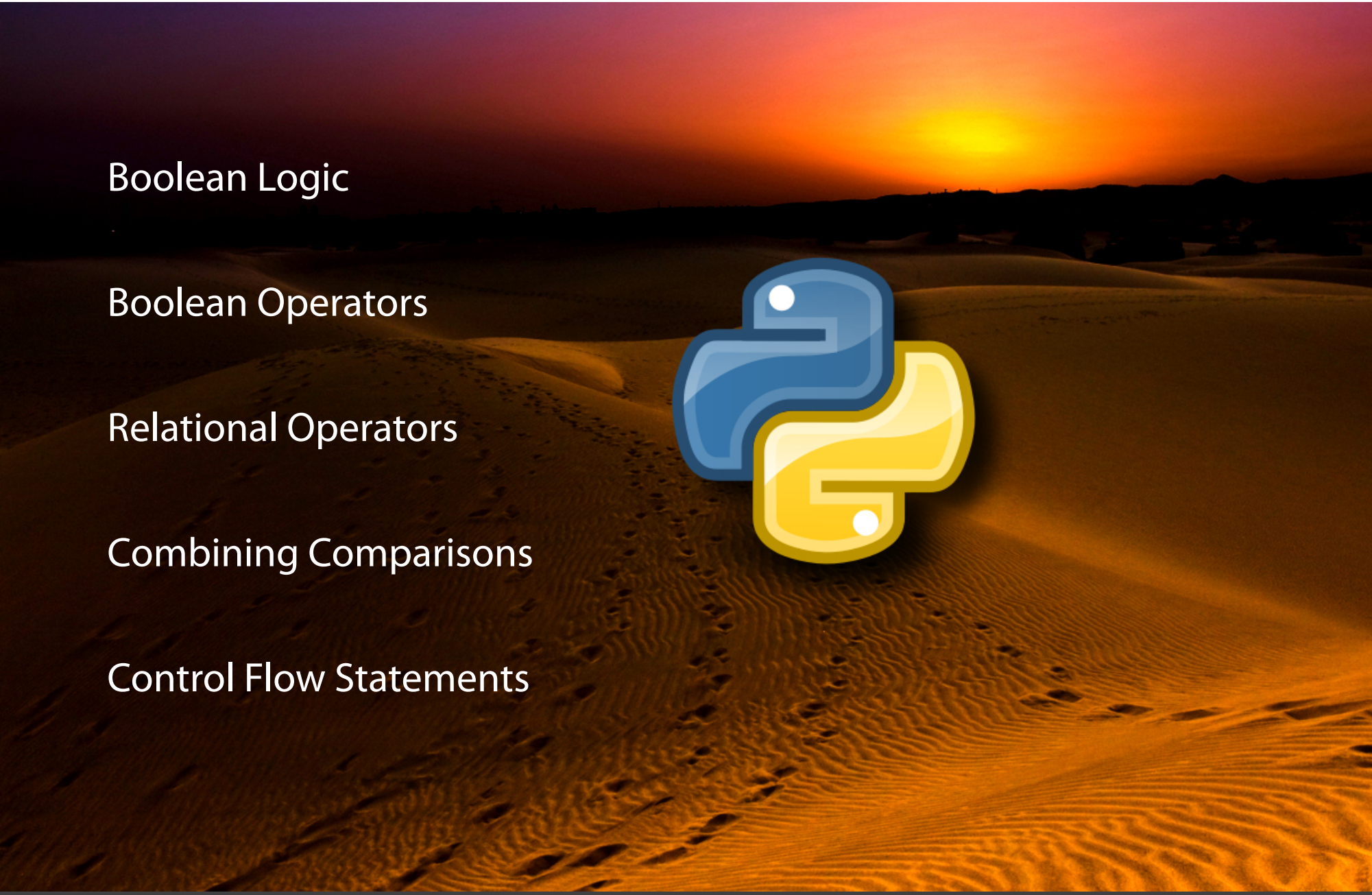
Section 5: Making Choices in Python

Last updated: 29 May 2019

UNIVERSITEIT STELLENBOSCH UNIVERSITY

Willem Bester

Section Outline



Boolean Logic

Boolean Operators

Relational Operators

Combining Comparisons

Control Flow Statements

Boolean Type and Operators

- George Boole showed in the 1840s that the rules of classical logic can be expressed in purely mathematical form using only the two values “true” and “false”
- Claude Shannon, the inventor of information theory, realised that Boole’s work could be used to optimise electromechanical telephone switches
- Boolean logic is used, inter alia, in the design of electronic circuits and computer programs
- Most programming languages provide a boolean type or at least statements that treat other values as boolean values

Boolean Operators

- Python has a boolean type called `bool`
- `bool` has only two possible values: `True` and `False`
- In normal speech, “true” and “false” are adjectives, but `True` and `False` are Python values, just as much as the `int` value `0` or the `float` value `-17.3`
- Only three basic boolean operators, given in order of precedence (low to high): `or`, `and`, and `not`
- They have meanings in line with common usage
- `and` and `or` are **binary operators**
- `not` is a **unary operator**

Boolean Operators

Example

```
>>> not True
False
>>> not False
True
>>> False and False
False
>>> False and True
False
>>> True and False
False
>>> True and True
True
```

Example

```
>>> False or False
False
>>> False or True
True
>>> True or False
True
>>> True or True
True
```

Do these results make sense?

Truth Tables

- We define boolean operators with **truth tables**

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

x	y	x or y
False	False	False
False	True	True
True	False	True
True	True	True

- `and` evaluates to true $\iff$ both its operands evaluate to true
- `or` evaluates to false $\iff$ both its operands evaluate to false
- Since `or` evaluates to false if either or both its operands evaluate to false, it is **inclusive**
- Sometimes we need an **exclusive or**, but then we have to create our own exclusive operation

Relational Operators

- Most often, boolean values are created in expressions
- The most common way is to use **relational operators**

Table: Relational Operators

Symbol	Operation
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

- If an operator consists of two symbols, there is no space between them

Relational Operators

- Relational expressions evaluate to true or false, just like we can say whether a mathematical statement is true or false
- Note that Python uses `==` for equality, and not `=`
- All relational operators are **binary**: They compare **two** values

Example

```
>>> 45 > 73
False
>>> 45 > 23
True
>>> 45 < 73
True
>>> 45 < 23
False
```

Relational Operators

- We can compare `int`s to `float`s
- `int`s are converted to `float`s in comparisons

Example

```
>>> 23.1 >= 23
True
>>> 23.1 >= 23.1
True
>>> 23.1 <= 23.1
True
>>> 23.1 <= 23
False
```

Example

```
>>> 67.3 == 87
False
>>> 67.3 == 67
False
>>> 67.0 == 67
True
>>> 67.0 != 67
False
>>> 67.0 != 23
True
```

Relational Operators

- It doesn't make much sense to compare two numbers you know in advance
- Relational operators almost always involve variables

Example

```
>>> def is_positive(x):
...     return x > 0
...
>>> is_positive(3)
True
>>> is_positive(-2.4)
False
>>> is_positive(0)
False
```

Combining Comparisons

Rules for Combining Operators

1. Arithmetic operators have higher precedence than relational operators
2. Relational operators have higher precedence than boolean operators
3. All relational operators have the same precedence

- For example, `+` and `/` are evaluated before `<` or `>`
- Also, comparisons are evaluated before `and`, `or`, and `not`
- For example, `1 + 3 > 7` is evaluated as `(1 + 3) > 7`

Combining Operators

- Often, we may omit the parentheses in complicated expressions
- However, for clarity, we'd rather leave them in

Example

```
>>> x = 2
>>> y = 5
>>> z = 7
>>> x < y and y < z
True
>>> (x < y) and (y < z)
True
```

Range Checking

- We often need to check whether a value lies in a given range
- Python lets us **chain** comparisons

Example

```
>>> x = 3
>>> (1 < x) and (x <= 5)
True
>>> x = 7
>>> (1 < x) and (x <= 5)
False
>>> x = 3
>>> 1 < x <= 5   # ≡ (1 < x) and (x <= 5)
True
```

Range Checking

Example

```
>>> 3 < 5 != True
True
>>> 3 < 5 != False
True
```

- `3 < 5 != True` is equivalent to `(3 < 5) and (5 != True)`
- Similarly, `3 < 5 != False` ≡ `(3 < 5) and (5 != False)`
- Since 5 is neither `True` nor `False`, the second half evaluates to `True` in each instance $\implies$ the expression is true as a whole
- Only chain expressions that make sense mathematically
- Use parentheses to make your meaning clear

Converting Numbers to Boolean

- Python converts `int`s to `float`s in mixed expressions
- Python also "converts" numbers to `bool`s
- `0` and `0.0` are treated as `False`
- All other numbers are treated as `True`

Example

```
>>> not 0
True
>>> not 1
False
>>> not 32.2
False
>>> not -87
False
```

Truth Value Testing

- Any object can be tested for a truth value
- By default, any object is considered `True`*
- The following built-in objects are considered `False`
  - Constants defined to be so:
    `None` and `False`
  - Zeros of any numeric type:
    `0  0.0  0j  Decimal(0)  Fraction(0, 1)`
  - Empty sequences and collections:
    `''  ()  []  {}  set()  range(0)`
- Operations and built-in functions that return a boolean may
  - return `0` for `False`, or
  - return `1` (but no other number) for `True`
- But `or` and `and` always return one of their operands

---

*Unless, for object `x`, its class defines special methods so that `bool(x)` returns `False` or `len(x)` returns zero; how to do this, we look at next year.

Boolean Operators: `and`, `or`, and `not`

Table: The boolean operators, ordered by ascending priority

Operation	Result	Short-circuiting
`x or y`	If *x* is false, then *y*, else *x*	Evaluates *y* only if *x* is false
`x and y`	If *x* is false, then *x*, else *y*	Evaluates *y* only if *x* is true
`not x`	If *x* is false, then `True`, else `False`	

- Since `not` has lower priority than non-boolean operators: `not a == b` is interpreted as `not (a == b)`
- Also, `a == not b` is a syntax error; the expression must rather be: `a == (not b)`

Expression Short-circuiting

- As soon as Python knows enough for an answer, it stops evaluating a boolean expression
- If the first argument of `and` is **false**, the expression is **false**
- If the first argument of `or` is **true**, the expression is **true**
- In either case, Python knows enough for an answer, so it doesn't evaluate the expression any further

Example

```
>>> True and 7
7
>>> False and 7
False
```

Example

```
>>> True or 0
True
>>> False or 18.2
18.199999999999999
```

Expression Short-circuiting

Example

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> True or 1 / 0
True
```

- Don't be too clever: Programs are meant to be readable
- Don't use `result = test and first or second` for

```
if test:
    result = first
else:
    result = second
```

Comparing Strings

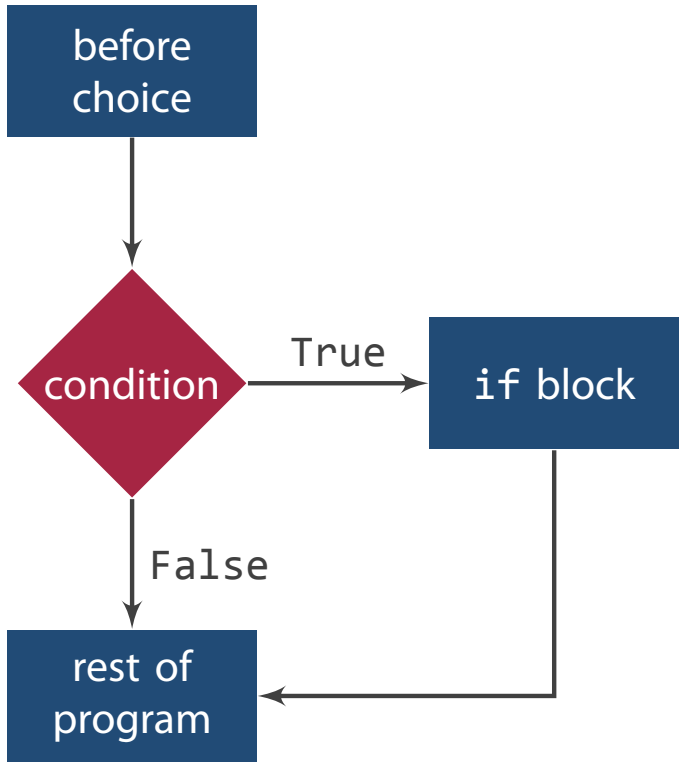
- Strings can be compared on their **lexicographic order**
- Uppercase letters come before the lower letters
- If a string `s1` is a prefix of another, longer string `s2`, then `s1` is "less than" `s2`

Example

```
>>> 'A' < 'a'
True
>>> 'A' > 'z'
False
>>> 'abc' < 'abd'
True
>>> 'abc' > 'abcd'
False
```

`if` statements

- Use an `if` statement to make a choice
- General form:
  `if` ⟨**condition**⟩`:`
      ⟨**block**⟩
- If the ⟨**condition**⟩ is **true**, then the ⟨**block**⟩ is **executed**
- However, if the ⟨**condition**⟩ is **false**, then the ⟨**block**⟩ is **skipped**
- Note that the ⟨**block**⟩ must be indented



Problem Statement

Example

Table: Solution Categories Based on pH Level

pH Level	Solution Category
0–4	Strong acid
5–6	Weak acid
7	Neutral
8–9	Weak base
10–14	Strong base

We can make Python execute certain statements when the pH level represented by some variable falls into a certain category.

Example Solution

Example

```
ph = float(input())
5.7
>>> if ph < 5.0:
...     print("Strong acid")
...
>>> if ph < 7.0:
...     print("Weak acid")
...
Weak acid
```

- The body of the first `if` statement is not executed
- But the body of the second one is
- What about a pH of 3.7…?

Example Solution

Example

```
ph = float(input())
3.7
>>> if ph < 5.0:
...     print("Strong acid")
...
Strong acid
>>> if ph < 7.0:
...     print("Weak acid")
...
Weak acid
```

- Oops, a pH of 3.7 triggers both bodies
- What can we do?

Example Solution

Example

```
>>> ph = float(input())
3.7
>>> if 0.0 <= ph < 5.0:
...     print("Strong acid")
...
Strong acid
>>> if 5.0 <= ph < 7.0:
...     print("Weak acid")
...
>>>
```

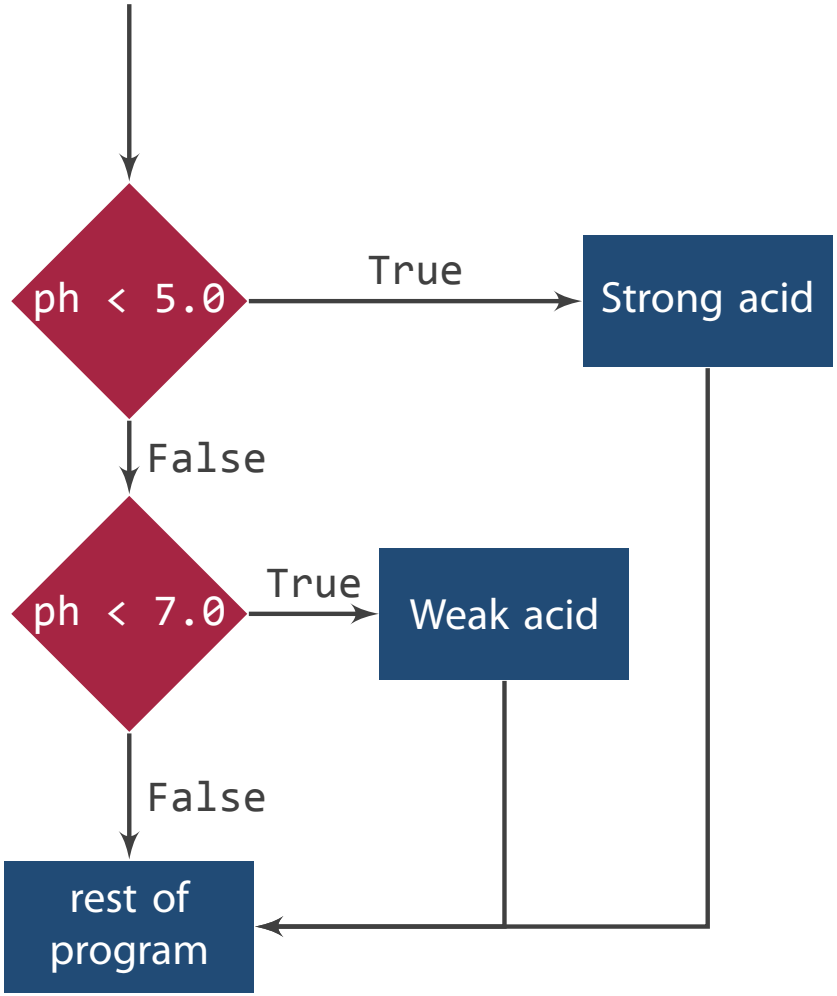
- This works
- Or we could use an `elif` clause....

Example Solution

Example

```
>>> if ph < 5.0:
...     print("Strong acid")
... elif ph < 7.0:
...     print("Weak acid")
...
Strong acid
>>>
```

Note: If the condition of the `if` is true, then neither the `elif` nor its block is executed.



`elif` clauses

- A condition-and-block pair is called a **clause**
- `elif` is for "else if"
- General form:
  `elif` ⟨**condition**⟩`:`
  ⟨**block**⟩
- If the ⟨**condition**⟩ is **true**, then the ⟨**block**⟩ is **executed**; if the ⟨**condition**⟩ is **false**, then it is **skipped**
- As usual the ⟨**block**⟩ must be indented
- An `elif` clause may be preceded by other `elif` clauses, and the top one of these must be preceded by an `if` clause
- An `if` clause may be followed by any number (including 0) of `elif` clauses

`else` clauses

- Use an `else` clause for a default action
- General form:

  `else:`

  ⟨**block**⟩
- An `if` statement can have at most one `else` clause
- The `else` clause must be the final clause in the statement
- Note that:

`if` ⟨**condition**⟩`:`
    ⟨**if block**⟩
`else:`
    ⟨**else block**⟩

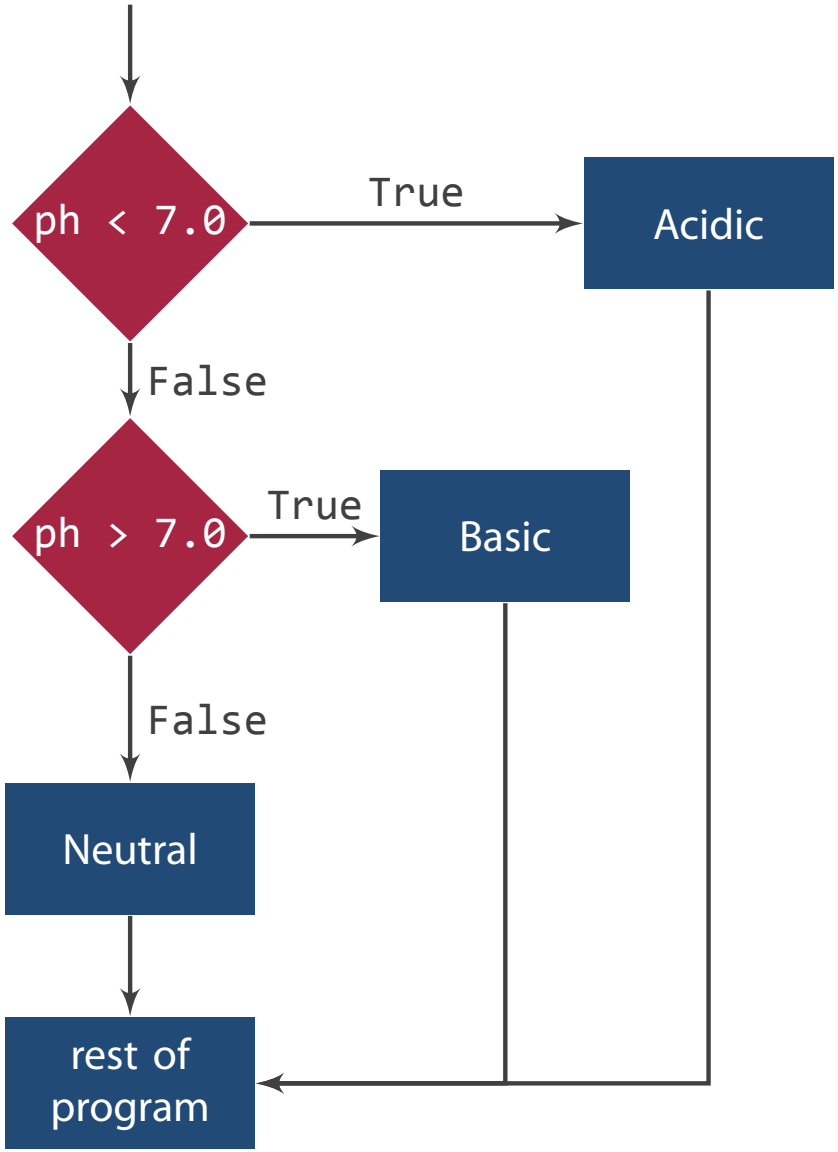
is equivalent to

`if` ⟨**condition**⟩`:`
    ⟨**if block**⟩
`if not` ⟨**condition**⟩`:`
    ⟨**else block**⟩

`elif` clauses

Example

```
>>> ph = float(input())
7.0
>>> if ph < 7.0:
...     print("Acidic")
... elif ph > 7.0:
...     print("Basic")
... else:
...     print("Neutral")
...
Neutral
>>>
```



Example

- An `if` statement inside another is called **nested**
- Is the logic of the following correct?

Example (`ph.py`)

```
value = input("pH value: ")
if len(value) > 0:
    ph = float(value)
    if ph < 0.0 or ph > 14.0:
        print("Invalid pH value")
    elif ph > 7.0:
        print("Acidic")
    elif ph > 14.0:
        print("Basic")
    else:
        print("Neutral")
else:
    print("No pH value given")
```

Example

- We can store the result of a boolean expression in a variable
- For example, to what does the expression `x = 15 > 5` evaluate?

Example

BMI	Age	
	< 45	≥ 45
< 22	Low	Medium
≥ 22	Medium	High

Figure: Risk of heart disease, based on age and body mass index

Solution with Stored Conditionals

Example

```
>>> young = age < 45
>>> slim = bmi < 22.0
>>> if young and slim:
...     risk = 'low'
... elif young and not slim:
...     risk = 'medium'
... elif not young and slim:
...     risk = 'medium'
... elif not young and not slim:
...     risk = 'high'
...
```

Stored Conditionals as List Indices

Example

```
>>> table = [['medium', 'high'],
...          ['low',    'medium']]
>>> young = age < 45
>>> heavy = bmi >= 22.0
>>> risk = table[young][heavy]
```