Practical Programming
in Python

Inspired by 'Practical Programming' by Paul Gries, Jennifer Campbell, Jason Montojo

Lecture 5
Making Choices

The Boolean Type, Boolean Operators, Relational Operators, Choosing which Statements to Execute

Kurt Rinnert, Kate Shaw

Physics Without Frontiers

The Abdus Salam International Centre for Theoretical Physics

Abstract

"The best thing about a boolean is even if you are wrong, you are only off by a bit."

– Anonymous

This lecture introduces another fundamental concept: making choices. We do this whenever we want our program to behave differently depending on the data it's working with.

We'll introduce statements for making choices called *control flow* statements.

These statements involve a Python type that is used to represent truth and falsehood.

Overview

- The Boolean type
- Boolean operators
- Relational Operators
- Control flow statements

Reacting differently to varying inputs makes programs much more useful.

The Boolean Type

- Python has a type called `bool`
- `bool` is a type just like `int` or `float`
- Unlike the numeric types, `bool` has only two values: **`True`** and **`False`**
- **`True`** and **`False`** are values just like `0`, `-11.3` and `'Grace Hopper'`

George Boole – A Little History

In the 1840's the mathematician George Boole expressed classical logic in purely mathematical form, using only two values: *true* and *false*.
A century later the inventor of information theory, Claude Shannon, used this system to optimize the design of electromechanical phone switchboards. This work led to the use of *Boolean logic* to design computer circuits.
In honor of Boole's work, most modern programming languages have a type named after him to track what is true and what is false.

We use the type `bool` to describe logical states.

Boolean Operators

- There are only three Boolean operators in Python
- In order of precedence they are: **`not`**, **`and`**, **`or`**
- Here is how they work:

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

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

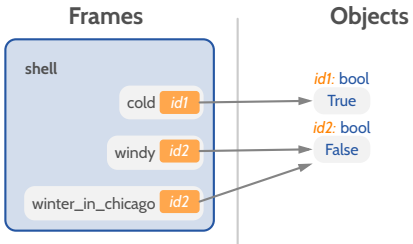
```
>>> True or True # or is inclusive!
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

Beware: unlike in English, `or` is *inclusive*.

Boolean Expressions & Variables

```
>>> cold = True
>>> windy = False
>>> winter_in_chicago = cold and windy
```

- We can build Boolean expressions from Boolean values and operators
- Expressions yield values that can be assigned to variables
- Boolean variables refer to objects of type `bool`



The only special thing about `bool` is that there are only two possible values.

Boolean Operators & Expressions – A Truth Table

cold	windy	cold and windy	cold or windy	(not cold) and windy	not(cold and windy)
`True`	`True`	`True`	`True`	`False`	`False`
`True`	`False`	`False`	`True`	`False`	`True`
`False`	`True`	`False`	`True`	`True`	`True`
`False`	`False`	`False`	`False`	`False`	`True`

Relational Operators

- Usually Boolean values are produced by expressions
- Most of the time using *relational operators*
- Comparisons are common relational operators

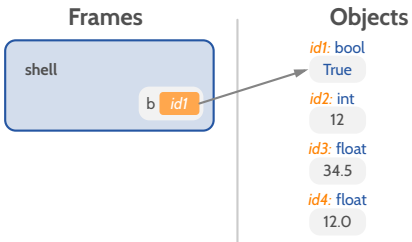
```
>>> 7 < 9
True
>>> 17 > 64
False
>>> 11 == 11
True
>>> 11 != 11
False
>>> 34 <= 34
True
```

Symbol	Operation
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
==	equal to
!=	not equal to

Note Python uses == for equality, = is reserved for assignments.

Comparing Different Types

```
>>> b = 12 < 34.5
>>> b
True
```



- We can compare objects of types `int` and `float`
- Python first converts the `int` object to `float`
- Then Python compares the two `float` values
- This is equivalent to:

```
>>> b = float(12) < 34.5
```

The `float` conversion can cause problems due to rounding errors.

Relational Operators in Practice

- It does not make sense to compare numbers we know in advance
- Usually we compare variables
- We give a simple function as an example
- Note that it follows the recipe defined in lecture 3
- The docstring describes what the function does and gives examples for testing
- The term "iff" means "if and only if" (this is mathematican's jargon)

```
def is_strictly_positive(x):
    """
    Return True iff x is strictly positive.

    Examples:

        >>> is_strictly_positive(3)
        True
        >>> is_strictly_positive(-1.2)
        False
        >>> is_strictly_positive(0)
        False
    """
    return x > 0
```

We will learn ways to make comparisons even more useful soon.

Combining Comparisons

- So far we have seen three types of operators
- In order of precedence these are:
  - Arithemtic operators (`+`, `-` and so on)
  - Relational operators (`>`, `==` and so on)
  - Boolean operators (`not`, `and`, `or`)
- The precedences are designed to make combinations easy to write and read:

```
>>> x = 3
>>> y = 6
>>> z = y
>>> x < y and y < z + 1
True
```

- This is fine, but consider using parentheses even in simple cases:

```
>>> (x < y) and (y < (z + 1))
```

Make your code readable. Avoid misunderstandings. Use parentheses.

Applying Boolean Operators to Numbers and Strings

- Numbers and strings can be used with Boolean operators
- Python treats `0` and `0.0` as **`False`** and all other numbers as **`True`**:

```
>>> not 0
True
>>> not 1
False
>>> not -43.2
False
```

- The empty string is treated as **`False`** and all other strings are treated as **`True`**:

```
>>> not ''
True
>>> not 'bad'
False
```

Don't be obscure. Treating an `int` with value `0` or an empty string as `False` is fine.

Short-Circuit Evaluation

- Python evaluates Boolean expressions from left to right
- Python stops the evaluation as soon as the result is determined
- For example:

```
>>> (2 < 3) or (1 // 0)
True
```

- What would happen without the short-circuit here?

Short-circuit saves time. Python programmers (you) have to know what's going on.

Comparing Strings

- We can compare strings just like we can compare numbers
- Behind the scenes, characters in strings a represented by integer numbers
- The details can be messy
- But this is the mechanism that allows to compare strings
- Things are arranged such that string comparisons reflect alphabetical order:

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

Strings are *ordered* alphabetically.

String Contents

- Comparisons are not the only relational operators
- Python also provides the operator **`in`**
- We can use it to check whether a string is contained in another string:

```
>>> 'Jan' in '01 Jan 1838'
True
```

- The **`in`** operator is case sensitive:

```
>>> 'A' in 'abc'
False
>>> 'a' in 'abc'
True
```

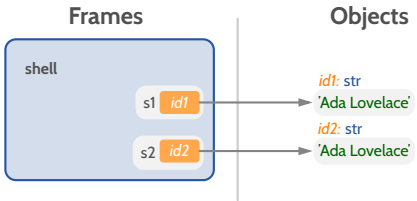
- The empty string is always contained:

```
>>> '' in 'abc'
True
>>> '' in ''
True
```

The `in` operator also works for other types. We will learn about them later.

Object Identity

```
>>> s1 = 'Ada Lovelace'
>>> s2 = 'Ada Lovelace'
>>> s1 == s2
True
>>> s1 is s2
False
```



- Python provides the **`is`** operator to test whether two objects are *identical*
- Objects can have equal values while *not* being identical
- Identical objects always have equal values

It is important to understand the difference between *identity* and *equivalence*.

Choosing Which Statements to Execute

- Python provides the **`if`** statement
- The **`if`** statement lets you choose which code is executed
- Its general form is:

```
if condition:
    block
```

- The *condition* is an expression, often (but not necessarily) Boolean:

```
if color != 'orange':
    print('The color is not orange.')

if x < y:
    pass # the pass statement is a statement that does nothing. It is surprisingly useful.
```

Like a function block, an if-block must be indented correctly.

Choosing Which Statements to Execute

- For example, we can print a message only when the user enters a pH level that is acidic:

```
>>> ph = float(input('Enter a pH level: '))
>>> if ph < 7.0:
...     print(ph, 'is acidic')
...
6.0 is acidic
```

Category Example

pH Level	Solution Category
0 – 4	strong acid
5 – 6	weak acid
7	neutral
8 – 9	weak base
10 – 14	strong base

Try what happens if you *don't* indent the block!

Control Flow: The `if` Statement

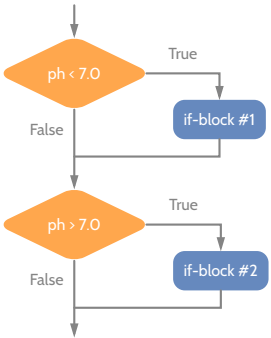
- The **`if`** statement executes a code block if the condition evaluates to **`True`**
- The picture on the right is called a *flow chart*

```
def acidity_if():
    ph = float(input("Enter a pH level: "))
    if ph < 7.0:
        print(ph, "is acidic.")

    if ph > 7.0:
        print(ph, "is basic.")
```

```
>>> acidity_if()

  Enter a pH level: 8.5
  8.5 is basic.
```



Each condition is *always* checked, even though *we* know only one will hold.

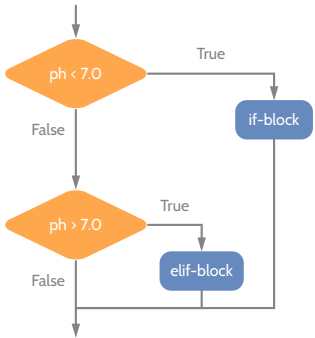
Control Flow: The `elif` Statement

- The **`if`** statement has to come first and behaves as before
- The **`elif`** statement is only evaluated if *all* previous conditions are **`False`**

```
def acidity_elif():
    ph = float(input("Enter a pH level: "))
    if ph < 7.0:
        print(ph, "is acidic.")
    elif ph > 7.0:
        print(ph, "is basic.")
```

```
>>> acidity_elif()

  Enter a pH level: 8.5
  8.5 is basic.
```



This expresses our a priori knowledge much better.

Control Flow: Beware of `elif` vs. multiple `if` Logic

- Behaviour is different when the **`if`** statement modifies the tested object

```
def acidity_if():
    ph = float(input("Enter a pH level: "))
    if ph < 7.0:
        ph = 8.0

    if ph > 7.0:
        print(ph, "is basic.")
```

```
def acidity_elif():
    ph = float(input("Enter a pH level: "))
    if ph < 7.0:
        ph = 8.0
    elif ph > 7.0:
        print(ph, "is basic.")
```

- Behaviour is different when the conditions overlap

```
def acidity_if():
    ph = float(input("Enter a pH level: "))
    if ph < 7.0:
        print(ph, "is acidic.")

    if ph < 4.0:
        print(ph, "is strongly acidic.")
```

```
def acidity_elif():
    ph = float(input("Enter a pH level: "))
    if ph < 7.0:
        print(ph, "is acidic.")
    elif ph < 4.0:
        print(ph, "is strongly acidic.")
```

We strongly recommend you avoid these scenarios whenever possible.

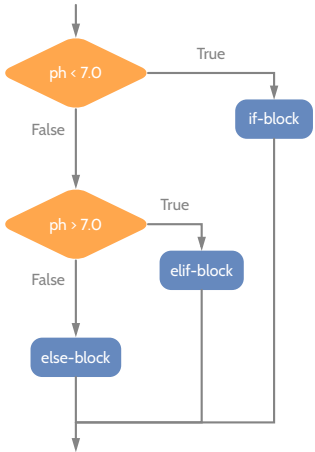
Control Flow: The `else` Statement

- The **`if`** and **`elif`** statements behave as before
- The **`else`** statement has to come last and is only evaluated if *all* previous conditions are **`False`**

```
def acidity_elif_else():
    ph = float(input("Enter a pH level: "))
    if ph < 7.0:
        print(ph, "is acidic.")
    elif ph > 7.0:
        print(ph, "is basic.")
    else:
        print(ph, "is neutral.")
```

```
>>> acidity_elif_else()

  Enter a pH level: 7.0
  7.0 is neutral.
```

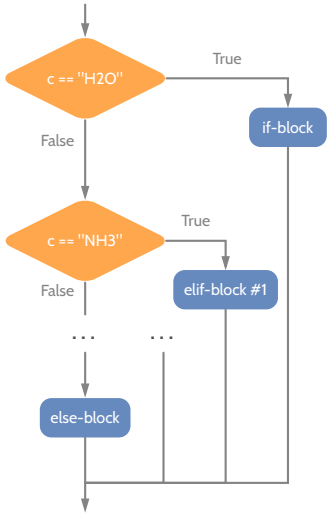


Think of `else` as covering the default case.

Control Flow: Multiple `elif` Statements

- There has to exactly one **`if`** statement
- There can be multiple **`elif`** statements at the same level
- There can only be one **`else`** statement

```
def compounds():
    c = input("Enter the compound: ")
    if c == "H2O":
        print("Water")
    elif c == "NH3":
        print("Ammonia")
    elif c == "CH4":
        print("Methane")
    else:
        print("Unknown compound")
```

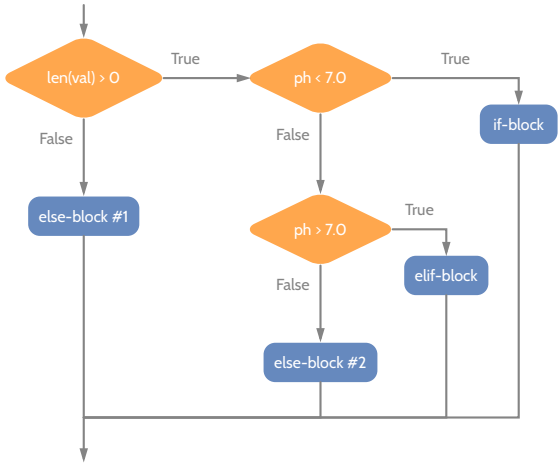


It is good practice to use multiple `elif` statements for *distinct* cases.

Control Flow: Nested `if` Statements

- An **`if`**-block can contain any statement
- In particluar another **`if`** statement

```
def acidity_elif_else():
    val= input("Enter a pH level: ")
    if len(val) > 0:
        ph = float(val)
        if ph < 7.0:
            print(ph, "is acidic.")
        elif ph > 7.0:
            print(ph, "is basic.")
        else:
            print(ph, "is neutral.")
    else:
        print('No pH value given!')
```



Nesting control statements is common. We will see more examples later.

Exercises Lecture 5