Chapter 2 - Data Types

CS 171 - Computer Programming 1
Lanzhou University

These slides use many elements provided in the main bibliographic reference for these lectures:

Programming in Python 3
A Complete Introduction to the Python Language,
2nd Edition,
Mark Summerfield

Outline

Outline

Identifiers and Keywords

- When we create a data item, we can
  - assign it to a variable;
  - insert it into a collection;
- The names of our object references are called *identifiers* or simply *names*;

Identifiers and Keywords

A valid Python identifier:

- is a nonempty, of any length, sequence of characters, that consists:

First rule:

- An identifier is composed of a *start* character
  - any Unicode letter
    - ASCII letters (“a”, “b”, ..., “z”, “A”, “B”, ..., “Z”)
    - underscore (“_”)
    - letters from most non-English languages
- and zero or more *continuation* characters
  - any character allowed as start character
  - plus Unicode digits
    - (“0”, “1”, ..., “9”)
  - the Catalan character “·”

Identifiers and Keywords

Identifiers are case sensitive:

- These are all different identifiers
  - TAXRATE
  - Taxrate
  - TaxRate
  - taxRate
  - taxrate

Second rule:

- Python's keywords are not allowed as identifiers:

```
and    as      assert  break   class  continue  except
def    False   del     finally elif   for       else
from   global  lambda  if      None   import    nonlocal
in     not     True    is      or     try       pass
while  raise   with    return  yield
```

Identifiers and Keywords

Looking at an example

- Written by a Spanish-speaking programmer;
- Assuming that
  - `import math` was declared
  - `radio` and `vieja_área` have been created earlier

```
π = math.pi
ϵ = 0.0000001
nueva_área = π * radio * radio
if abs(nueva_área - vieja_área) < ϵ:
      print("las áreas han convergido")
```

By the way,

- What does the above program do?

Identifiers and Keywords

- It is easy to check the validity of an identifier on a Python interpreter:

```
>>> stretch-factor = 1
(...)
SyntaxError: can't assign to operator # - is not Unicode

>>> 2miles = 2
(...)
    2miles = 2
         ^
SyntaxError: invalid syntax
      # start character is not Unicode letter or underscore
>>> l'impot = 4
(...)
    l'impot = 4
              ^
SyntaxError: EOL while scanning string literal
      # a quote is not Unicode letter, digit, or underscore
```

Outline

Integral Types

- Python provides two built-in types
  - `int`
  - `bool`
- Both are immutable
  - but given Python's augmented assignment operator this is hardly noticeable;
- When used in a Boolean context,
  - `0` and `False` are interpreted as `False`
  - any other integer and `True` are interpreted as `True`
- When used in an integer context,
  - `True` evaluates to `1`
  - `False` evaluates to `0`

Integers

Numeric Operations and Functions

Syntax	Description
`x + y`	Adds number x and number y
`x - y`	Subtracts y from x
`x * y`	Multiplies x by y
`x / y`	Divides x by y; always produces a float (or a complex if x or y is complex)
`x // y`	Divides x by y; truncates any fractional part so always produces an int result
`x % y`	Produces the modulus (remainder) of dividing x by y
`x ** y`	Raises x to the power of y
`-x`	Negates x; changes x's sign if nonzero, does nothing if zero
`+x`	Does nothing; is sometimes used to clarify code
`abs(x)`	Returns the absolute value of x
`divmod(x, y)`	Returns the quotient and remainder of dividing x by y as a tuple of two ints
`pow(x, y)`	Raises x to the power of y; the same as the ** operator
`pow(x, y, z)`	A faster alternative to `(x ** y) % z`

Integers

- All the binary numeric operators
  - +, -, /, //, %, and **
- have an augmented assignment version
  - +=, -=, /=, //=, %=, and **=
- where `x op= y` is the same as `x = x op y`

Integers

- Integer Literals are written using base 10 by default, but others are possible:

```
>>> 14600926                                # decimal
14600926
>>> 0b11011110110010101011011110            # binary
14600926
>>> 0o67545336                              # octal
14600926
>>> 0xDECADE                                # hexadecimal
14600926
```

Integers

Integer Conversion Functions

Syntax	Description
`bin(i)`	Returns the binary representation of int i as a string, e.g., `bin(1980) == '0b11110111100'`
`hex(i)`	Returns the hexadecimal representation of i as a string, e.g., `hex(1980) == '0x7bc'`
`int(x)`	Converts object x to an integer; raises ValueError on failure—or TypeError if x's data type does not support integer conversion. If x is a floating-point number it is truncated.
`int(s, base)`	Converts str s to an integer; raises ValueError on failure. If the optional base argument is given it should be an integer between 2 and 36 inclusive.
`oct(i)`	Returns the octal representation of i as a string, e.g., `oct(1980) == '0o3674'`

Booleans

- There are two built-in Boolean objects:
  - `True`
  - `False`

```
>>> x = bool()
>>> x
False
>>> y = bool(True)
>>> y
True
>>> x and y
False
```

- Three logical operators are available:
  - `and` and `or` use short-circuit logic and return the operator that determined the result;
  - `not` always returns either `True` or `False`

Outline

Floating-Point Types

- Python provides three kinds of floating-point values:
  - the built-in `float` and `complex` types
  - the `decimal.Decimal` from the standard library

Type float

- Holds double-precision floating point numbers
- They have limited precision and cannot reliably be compared for equality
- This means that some numbers (e.g., 0.1) can not be stored exactly
  - which is not specific to Python at all!
- Examples include

```
>>> 0.0, 5.4, -2.5, 8.9e-4
(0.0, 5.4, -2.5, 0.00087)
```

Floating-Point Types

If we really need high precision:

- one should use Python's `decimal.Decimal` numbers
- These perform calculations that are accurate to the level of precision we specify
  - by default, to 28 decimal places
- and can represent periodic numbers like `0.1` exactly
- `decimal.Decimal` numbers are suitable for financial calculations

Floating-Point Numbers

- All the numeric operations in slide 12 can be used with a `float`
- This includes the augmented assignment versions

```
>>> x = float()
>>> x
0.0
>>> y = float(3.5)
>>> y
3.5
```

- One may define a function to compare `floats` for equality:

```
def equal_float(a, b):
        return abs(a - b) <= sys.float_info.epsilon
```

Floating-Point Numbers

- Floating-point numbers can be converted to other types:

```
>>> x = float(3.5)
>>> x
3.5
>>> y = int(x)
>>> y
3
>>> z = round(x)
>>> z
4
```

Floating-Point Numbers

- The `math` module provides many more operations on `floats`:

```
>>> import math
>>> math.pi * (5 ** 2)
78.53981633974483
>>> math.hypot(5, 12)
13.0
>>> math.modf(13.732)
(0.7319999999999993, 13.0)
```

- `math.hypot(x, y)` calculates the distance from the origin to the point (x,y)

Complex Numbers

- The `complex` data is an immutable type that holds a pair of `floats`
  - one representing the real part of the number
  - the other representing the imaginary part of the number
- Examples include:

```
3.5+2j      0.5j      4+0j      -1-3.7j
```

- The separate parts if a `complex` are available as attributes `real` and `imag`

```
>>> z = -89.5+2.125j
>>> z.real, z.imag
(-89.5, 2.125)
```

- All the numeric operators and functions in slide 12 can be used on `complex`
  - Except for `//`, `%`, `divmod()` and the three-argument `pow()`
- By the way, what is the behavior of the `complex()` function?

Decimal Numbers

- In many applications, the numerical inaccuracies of `floats` don't matter
  - and are actually outweighed by the speed of calculating with `floats`
- But sometimes we really need to chose accuracy over speed
- `decimal` provides `Decimal` numbers that are as accurate as we specify
- To create a `Decimal` we must import the `decimal` module

```
>>> import decimal
>>> a = decimal.Decimal(9876)
>>> b = decimal.Decimal("54321.01234567898765432l")
>>> a + b
Decimal('64197.01234567898765432l')
```

- `decimal.Decimal()` can not be passed a `float`

Decimal Numbers

- One can observe the accuracy differences between `floats` and `decimal.Decimal`

```
>>> 23/1.05
21.904761904761905
>>> decimal.Decimal(23) / decimal.Decimal("1.05")
Decimal('21.904761904761904761904761904')
```

Outline

Strings

- Are represented by the `str` data type
- Hold sequences of Unicode characters
- Are created using quotes
  - Single quotes `'this is a string'`
  - Double quotes `"this is a string"`
  - Triple quotes, start and end either with `'''` or `"""`

```
>>> text = """A triple quoted string like this can include 'quotes' and
... "quotes" without formality. We can also escape newlines \
... so this particular string is actually only two lines long."""
>>> print(text)
A triple quoted string like this can include 'quotes' and
"quotes" without formality. We can also escape newlines so this particular str
```

Strings

- We can use quotes inside a normal quoted string if they are different from the delimiting quotes

```
a = "Single 'quotes' are fine; \"doubles\" must be escaped."
b = 'Single \'quotes\' must be escaped; "doubles" are fine.'
```

- Python uses newline as its statement terminator, except inside
  - inside parentheses (())
  - square brackets ([])
  - braces ({})
  - triple quoted strings
- Newlines can be used without formality in triple quoted strings
- We can include newlines in any string using the \n escape sequence

```
>>> text = "first line \n second line"
>>> print(text)
first line
 second line
```

Comparing Strings

- Strings support the usual comparison operators
  - <
  - <=
  - ==
  - !=
  - >
  - >=
- These operators compare strings byte by byte in memory
  - this may sometimes raise issues, e.g., when sorting lists of strings
    - for example, when strings come from external sources

  We will see later how we can customize Python's sort methods

Slicing and Striding Strings

- We know we can use [] to access individual items in a sequence

```
>>> str = "Light ray"
>>> str[0]
'L'
>>> str[4]
't'
```

- But actually, the access operator [] has 3 possible syntaxes:
  - `seq[_start_] # as above, but also:`

```
>>> str = "Light ray"
>>> str[-1] # This is useful to access the last character
'y'
>>> str[-3]
'r'
```

  - `[_start:end_]`
  - `[_start:end:step_]`

Slicing and Striding Strings

- Let's look at how to use:
  - [_start:end_]

```
>>> str = "The maxwork man"
>>> str[4:11]  # string starting(ending) in character at index 4(11)
'maxwork'
>>> str[4:]    # _end_ can be omitted and is understood as len(seq)
'maxwork man'
>>> str[:4]    # _start_ can be omitted and is understood as 0
'The '
>>> str[:]     # _start_ and _end_ both can be omitted
'The maxwork man'
```

  - [_start:end:step_]

```
>>> str = "abababababababababab"
>>> str[0::2]  # from 0 til the end, every other character
'aaaaaaaaa'
>>> str[1::2]
'bbbbbbbbb'    # again, _end_ is being omitted
```

Slicing and Striding Strings

The operator []

- Can be used not only with strings, but with any sequence,
  - such as a list or a tuple;
- `_start_` and `end` must be integers, or variables holding integers

- We can mix slicing with concatenation to insert a substring in a string:

```
>>> str = "The maxwork man"
>>> s = str[:12] + "wo" + str[12:]
>>> s
'The maxwork woman'
```

String Operators and Methods

- Strings are immutable sequences
  - All the functionality that can be used with immutable strings can be used with strings
  - This includes
    - membership testing `in`
    - concatenation `'`
    - appending `+=`
    - replication `*`
    - augmented assignment replication `*=`
- The list of methods that are available for strings is very rich
  - We will use examples to explain many such methods
  - For a complete list, we refer to the main bibliographic reference

String Operators and Methods

```
>>> str = "I am a string with 32 characters"
>>> len(str)    # len() counts the number of characters in a string
32
```

```
>>> treatises = ["Arithmetica", "Conics", "Elements"]
>>> " ".join(treatises)
'Arithmetica Conics Elements'   # join() concatenates the elements in a
                                # sequence into a string with the string
                                # the method was called on between them
                                # (" " in this case)
```

join() is preferable to concatenation (+) when dealing with many strings

```
>>> s = "=" * 5    # * provides string replication
>>> print(s)
=====
>>> s *= 10        # we are using the augmented assignment version
>>> print(s)
==================================================
```

String Operators and Methods

```
>>> str = "I am a string"
>>> "am" in str
True
>>> "not" in str
False
```

```
>>> str = "abcdefghijklmnopqrstuvwxyz"

>>> str.index('d')
3
>>> str.index('A')   # an exception is raised:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found

>>> str.find('efg')
4
>>> str.find('efgx')
-1
```

String Formatting

- The `str.format()` method provides a flexible way to create strings
- `str.format()` returns a new string with the *replacement fields* in its string replaced with its arguments formatted:

```
>>> "The novel '{0}' was published in {1}".format("Hard Times", 1854)
"The novel 'Hard Times' was published in 1854"
```

- If we try to concatenate a string and a number, Python raises a `TypeError` exception; we can use `str.format()` for that:

```
>>> "{0}{1}".format("The amount due is $", 200)
'The amount due is $200'
```