CSC110 LEC9201 Lecture 2 Notes

Anatoly Zavyalov

September 15, 2020

1 Categorizing Data

1.1 Types of data

- Numeric Data: Natural numbers ($\mathbb{N}$), integers ($\mathbb{Z}$), rationals ($\mathbb{Q}$), reals ($\mathbb{R}$)
- Booleans (`True` / `False`)
- Strings, such as `"Anatoly is cool!"`

Collection data types:

- Sets (unordered, **cannot** contain duplicates)
- Lists (ordered, **can** contain duplicates)
- Mappings (pairs of associations between **keys** and their **corresponding values**)

2 Data types in Python

2.1 Representing values in code

A **literal** is the simplest piece of Python code: code that represents the exact value as written.

Literal	Type
`1`	`int`
`-2.3`	`float`
`True`	`bool`
`"Anatoly"`	`str`
`{1, 2, 3}`	`set`
`[1, 2, 3]`	`list`
`(1, 2, 3)`	`tuple`
`{'a': 1.5, 'b': 2.3, 'c': 6.9}`	`dict`

3 Operations

3.1 Numeric `int` / `float` operations

Symbol	Meaning
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`**`	Exponentiation
`/`	Division
`//`	Integer division
`%`	Remainder/modulo
`==`	Equal to
`!=`	Not equal to
`<`	Less than
`<=`	Less than or equal to
`>`	Greater than
`>=`	Greater than or equal to

Note that `/` returns a `float`, while `//` returns an `int`, e.g. `6 // 2` would equal to `3`, while `6 / 2` would equal to `3.0`.

3.1.1 Floating-point error

`(1 / 3 * 10) == 10 / 3` will return `False`. This is because `1 / 3 * 10` returns `3.333333333333333`, while `10 / 3` returns `3.3333333333333335`.

3.2 Some terminology

An **expression** is a piece of Python code that produces a value. We **evaluate** an expression to determine its value. Every literal is an expression, but not every expression is a literal.

A **statement** is a piece of code representing an instruction to the computer. Every expression is a statement, but not every statement is an expression.

Some examples of literals:

- `1`
- `[1, 2, 3]`
- `{1, 2, 3}`

Some examples of expressions:

- `1 + 5`
- `0.1 * 3.5 + 100`
- `64 * 2 <= 2 ** 7 + 1`

3.3 `bool` operations

3.3.1 `and` operator

Returns `True` if both statements evaluate to `True`. Otherwise, returns `false`.

For example:

```
>>> True and False
False
>>> (1 < 5) and (7 > 2)
True
>>> 9 == 4 and 2 > -5
False
```

3.3.2 `or` operator

Returns `True` if at least one of the statements evaluate to `True`. Otherwise, returns `False`.

For example:

```
>>> True or False
True
>>> 2 == 5 or 5 + 2 == 7
True
>>> 9 < 1 or 2 + 7583578945348957 < 0
False
```

3.3.3 `not` operator

Returns `True` if the statement evaluates to `False`. Otherwise, returns `False`.

For example:

```
>>> not True
False
>>> not 2 == 5
True
```

4 `str` operations

4.0.1 `==` (equality)

Returns `True` if the strings are the same (every character at every index is the same). Otherwise, returns `False`.

For example:

```
>>> "Hello!" == "Hello!Hello!"
False
>>> "Pie" == "Pie"
True
```

4.0.2 `+` (concatenation)

Returns the concatenation of the two strings as a string, i.e. appends the right string to the left one.

For example:

```
>>> 'Hello! ' + 'world!'
'Hello world!'
>>> 'My name is ' + 'Anatoly'
'My name is Anatoly'
```

4.0.3 `[<int>]` (indexing)

Returns the character at the `<int>-1`th (indices in Python start at `0`).

For example:

```
>>> 'Anatoly'[2]
'a'
>>> 'Bob'[0]
'B'
```

4.0.4 `*<int>` (repetition)

Returns the repetition `<int>` times of the string as a new string.

For example:

```
>>> "Cake" * 10
'CakeCakeCakeCakeCakeCakeCakeCakeCakeCake'
>>> '1234567890' * 4
'1234567890123456789012345678901234567890'
```

4.1 `set` operations

4.1.1 `==` (equality)

Returns `True` if the two sets contain the same elements. Otherwise, returns `False`.

For example:

```
>>> {1, 2, 3} == {3, 2, 1}
True
>>> {0, "hi", 2, 5} == {"yo", "sup", "asdfghjk"}
False
```

4.1.2 `in` (element of)

Returns `True` if a specified value is present in a set. Otherwise, returns `False`.

For example:

```
>>> "BananaBanana" in {"Apple", "Banana" * 2, "
    Grapes"}
True
>>> 5 in {False, 1, 2, 9 - 3}
False
```

4.2 `list` / `tuple` operations

Same operations as both strings and sets!

4.3 `dict` operations

4.3.1 `[_]` (key lookup)

Returns the **corresponding value** of a key in a dictionary.

For example:

```
>>> {1: 2, 3: 4}[3]
4
>>> {"Anatoly": 18, "Polina": 23}["Anatoly"]
18
```

4.3.2 `in` (key element of)

Returns `True` if a specified **key** is in a dictionary. Otherwise, returns `False`.

For example:

```
>>> 1 in {1: 2, 3: 4}
True
>>> "Bob" in {"Marley": "Bob", "Astley": "Rick"}
False
```

Note that `in` checks whether a **key** is in a dictionary, not a key's corresponding value.

5 Variables

5.1 Storing values

A **variable** is a piece of code that **refers** to a value, created using **assignment statemenets:**

`<variable> = <expression>`

Note that an assignment statement does not produce a value.

5.1.1 Executing an assignment statement

1. Python evaluates `<expression>`.
2. Python associates the resulting value to `<variable>`.

5.2 Keeping track of variables

```
    a = 3
    b = [1, 2, a]
    c = b * a
    d = c[0] + a
```

Variable	Value
`a`	`3`
`b`	`[1, 2, 3]`
`c`	`[1, 2, 3, 1, 2, 3, 1, 2, 3]`
`d`	`4`

5.3 Value-based memory model

A **memory model** is a structured way of represending variables and data in a program.

Our previous example of a table of values is a value-based memory model.

6 Comprehensions

In math, we use **set builder notation** to express large (possibly infinite!) sets:

$$\{x^2 \mid x \in \mathbb{N}\}$$

In Python, we can use `set comprehensions` to express sets like this by doing:

```
>>> nats = {0, 1, 2, 3, 4, 5, 6, 7}
>>> {x ** 2 for x in nats}
{0, 1, 4, 36, 9, 16, 49, 25}
```

We can also use `list comprehensions` to make a list like this by doing:

```
>>> nats = {0, 1, 2, 3, 4, 5, 6, 7}
>>> [x ** 2 for x in nats]
[0, 1, 4, 36, 9, 16, 49, 25]
```

Dictionary comprehensions look like this:

```
>>> nats = {0, 1, 2, 3, 4, 5, 6, 7}
>>> {x : x ** 2 for x in nats}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49}
```

7 `range`: a sequence of numbers

For integers `m` and `n`, `range(m, n)` represents the sequence of numbers `m`, `m + 1`, ..., `n-1` (notice that the last number is not `n`).

This can be used in a comprehension:

```
>>> {x**2 for x in range(0, 10)}
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
```