KOLT Python

Review 1: Connect Four

Ahmet Uysal

Wednesday 6th March, 2019

Agenda

1. Variables & I/O
2. Basic Operators
3. Branching
4. Lists
5. Strings
6. Loops
7. Connect Four

Comments

```
# Single line comments start with a '#'

"""
Multiline comments can be written between
three "s and are often used as function
and module comments.
"""
print('Hello, stranger!')
```

Python will basically ignore comments, they are purely written **for humans**!

Variables

Type	Explanation	Examples
`int`	represent **integers**	3, 4, 17, -10
`float`	represent **real numbers**	3.0, 1.11, -109.123123
`bool`	represent **boolean** truth values	`True`, `False`
`str`	A sequence of characters.	'Hello', '', '3'
`NoneType`	special and has one value, None	`None`

- How to create a variable? `variable_name = value`
- How about types? use `type()`
- Can a variable change type? **Yes!** Just assing a new value with any type.
- What if we if want to convert a value between types, i.e, '2'$\rightarrow$ 2

Casting

- `int('2')` $\rightarrow$ 2
- Any possible reasons for casting? -taking user input -reading numbers from a file?
- Can we cast every value to every type? **NO!** try `int('hello')`

Console I/O(Input/Output)

`print(*args, sep=' ', end='\n')`

- Can take arbitrary number of arguments
- Separates elements with space by default
- Adds newline character `'\n'` to end by default

`input([prompt])`

- Prints the prompt to Console
- Program is paused until user enters something
- **returns an `str` object!**

`bool` Operators

How to represent logical operations in Python? (and, or, not)

A	B	A or B	A and B	not A
`True`	`True`	`True`	`True`	`False`
`True`	`False`	`True`	`False`	`False`
`False`	`True`	`True`	`False`	`True`
`False`	`False`	`False`	`False`	`True`

`True or False and False` ⇒ **`True`**

WHY?

- **`and`**
- **`or`**
- **`not`**

Operator Precedence

Logical operators are evaluated in this order:

1. `not`
2. `and`
3. `or`

You can override this order with parentheses

`(True or False) and False` ⇒ **`False`**

Short-Circuit Evaluation

`X`: Any boolean value

`True or X` ⇒ **`True`**

`False and X` ⇒ **`False`**

Python is smart enough to take advantage of this!

```
1/0  # => ZeroDivisionError
True or 1/0 # => True
False and 1/0 # => False
1/0 or True # => ZeroDivisionError
1/0 and False # => ZeroDivisionError
```

Arithmetic Operators

These operations are applicable on Numeric types: `int` and `float`

- `+`: Addition
- `-`: Subtraction
- `*`: Multiplication
- `/`: Division
- `//`: Floor(integer) Division
- `%`: Modulo
- `**`: Power

```
3.2 + 1.4   # => 4.6
3.2 - 1   # => 2.2
3.2 * 1.2   # => 3.84
3.5 / 1.5   # => 2.3333333333333335
3.5 // 1.5  # => 2.0
3.5 % 1.5   # => 0.5
2 ** 10   # => 1024
```

Comparison Operators

- <: Strictly less than
- <=: Less than or equal
- >: Strictly greater than
- >=: Greater than or equal
- ==: Equal
- !=: Not equal

```
3.0 == 3   # => True
3.0 >= 3   # => True
# Small-case characters
# have bigger ASCII value
'Aa' > 'aa' # => False
'hi' == 'hi'   # => True
'a' == None   # => True
3 > 'a'   # => TypeError
3 == 'a'   # => False
```

Chained Comparisons

`1 < 2 < 3` $\Rightarrow$ **`True`**

You can chain arbitrarily many comparison operations together.

v_i: variables/values, op_i: comparison operators

$v_1 \;\; op_1 \;\; v_2 \;\; op_2 \;\; v_3 \;\; \ldots \;\; op_{n-1} \;\; v_n$ is equivalent to:

$v_1 \;\; op_1 \;\; v_2$ **`and`** $v_2 \;\; op_2 \;\; v_3$ **`and`** $\ldots v_{n-1} \;\; op_{n-1} \;\; v_n$

```
3 > 2 == 1 < 5 > 4   # => False
3 > (2 == 1) < 5 > 4   # => True
3 > True > False # => True
3 > 5 < 1/0   # => False
3 < 5 < 1/0   # => ZeroDivisionError
```

Assignment Operators

We have already seen `'='`: `variable_name = value`
Frequently we will update variables' values based on their **old value**.
Ex: Increment a number: `num = num + 1`
Python has shorthand representations for these updates with arithmetic operators.
`num += 1` is equivalent to `num = num + 1`
`result *= 2` is equivalent to `result = result * 2`

Assignment Operators

Operator	Usage	Equivalent
`+=`	`val += 3`	`val = val + 3`
`-=`	`val -= 3`	`val = val - 3`
`*=`	`val *= 3`	`val = val * 3`
`/=`	`val /= 3`	`val = val / 3`
`%=`	`val %= 3`	`val = val % 3`
`**=`	`val **= 3`	`val = val ** 3`
`//=`	`val //= 3`	`val = val // 3`

Branching

```
if <condition>:
    <expression>
    <expression>
    ...
```

```
if <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

```
if <condition>:
    <expression>
    <expression>
    ...
elif <condition>:
    <expression>
    <expression>
    ...
...
else:
    <expression>
    <expression>
    ...
```

- `<condition>` has a **`bool`** value (`True` or `False`)
- Which expressions will be evaluated in which conditions?

Lists

- Group values together.

Lists

- Group values together. `my_values = [1, 'a', None]`

Lists

- Group values together. `my_values = [1, 'a', None]`
- You can think of each element as a variable, accessed by **indexing**

Lists

- Group values together. `my_values = [1, 'a', None]`
- You can think of each element as a variable, accessed by **indexing**
- You can do everything you do to variables to list elements:

Lists

- Group values together. `my_values = [1, 'a', None]`
- You can think of each element as a variable, accessed by **indexing**
- You can do everything you do to variables to list elements:
  - Assign new values: `my_values[0] = 3`

Lists

- Group values together. `my_values = [1, 'a', None]`
- You can think of each element as a variable, accessed by **indexing**
- You can do everything you do to variables to list elements:
  - Assign new values: `my_values[0] = 3`
  - Use shorthand assignment operators: `my_values[1] += 'bc'`

Lists

- Group values together. `my_values = [1, 'a', None]`
- You can think of each element as a variable, accessed by **indexing**
- You can do everything you do to variables to list elements:
  - Assign new values: `my_values[0] = 3`
  - Use shorthand assignment operators: `my_values[1] += 'bc'`
  - Learn their type: `type(my_values[2]) # => <class 'NoneType'>`

Lists

- Group values together. `my_values = [1, 'a', None]`
- You can think of each element as a variable, accessed by **indexing**
- You can do everything you do to variables to list elements:
  - Assign new values: `my_values[0] = 3`
  - Use shorthand assignment operators: `my_values[1] += 'bc'`
  - Learn their type: `type(my_values[2]) # => <class 'NoneType'>`
  - Change their type: `my_values[2] = True`

Lists

- Group values together. `my_values = [1, 'a', None]`
- You can think of each element as a variable, accessed by **indexing**
- You can do everything you do to variables to list elements:
  - Assign new values: `my_values[0] = 3`
  - Use shorthand assignment operators: `my_values[1] += 'bc'`
  - Learn their type: `type(my_values[2]) # => <class 'NoneType'>`
  - Change their type: `my_values[2] = True`
  - Compare their value: `if my_values[0] == my_values[1]: ...`

Lists

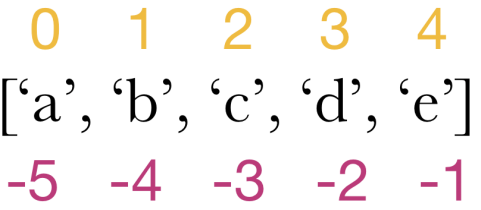
- Group values together. `my_values = [1, 'a', None]`
- You can think of each element as a variable, accessed by **indexing**
- You can do everything you do to variables to list elements:
  - Assign new values: `my_values[0] = 3`
  - Use shorthand assignment operators: `my_values[1] += 'bc'`
  - Learn their type: `type(my_values[2]) # => <class 'NoneType'>`
  - Change their type: `my_values[2] = True`
  - Compare their value: `if my_values[0] == my_values[1]: ...`
- What happens when we call `my_values[3] = 3`?

Lists

- Group values together. `my_values = [1, 'a', None]`
- You can think of each element as a variable, accessed by **indexing**
- You can do everything you do to variables to list elements:
  - Assign new values: `my_values[0] = 3`
  - Use shorthand assignment operators: `my_values[1] += 'bc'`
  - Learn their type: `type(my_values[2]) # => <class 'NoneType'>`
  - Change their type: `my_values[2] = True`
  - Compare their value: `if my_values[0] == my_values[1]: ...`
- What happens when we call `my_values[3] = 3`? `# =>` **`IndexError`**

List Indexing

Access elements at a particular index




```
x = [1, 2, 'a', 'hello']
x[0] # => 1
x[1] # => 2
x[2] # => 'a'
x[3] # => 'hello'
x[-1] # => 'hello'
x[-2] # => 'a'
x[-3] # => 2
```

List Slicing

Access collection of elements by specifying `[start:stop:step]`

List Slicing

Access collection of elements by specifying `[start:stop:step]`
Gives a list, even when number of elements is not bigger than 1.

List Slicing

Access collection of elements by specifying **`[start:stop:step]`**
Gives a list, even when number of elements is not bigger than 1.

```
numbers = [0, 1, 2, 3, 4, 5]
"""
ASCII art analogy :)
 +---+---+---+---+---+---+
 | 0 | 1 | 2 | 3 | 4 | 5 | => Indices
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6 => Borders
-6  -5  -4  -3  -2  -1
"""
```

```
numbers[0::2]   # => [0, 2, 4]
numbers[:]   # => [0, 1, 2, 3, 4, 5]
numbers[1:]   # => [1, 2, 3, 4, 5]
numbers[-2:]   # => [4, 5]
numbers[1:4]   # => [1, 2, 3]
numbers[1:1]   # => []
numbers[-99:99]   # => [0, 1, 2, 3, 4, 5]
numbers[::-1]   # => [5, 4, 3, 2, 1, 0]
numbers[::-2]   # => [5, 3, 1]
```

List Slicing

Access collection of elements by specifying **`[start:stop:step]`**
Gives a list, even when number of elements is not bigger than 1.

```
numbers = [0, 1, 2, 3, 4, 5]
"""
ASCII art analogy :)
 +---+---+---+---+---+---+
 | 0 | 1 | 2 | 3 | 4 | 5 | => Indices
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6 => Borders
-6  -5  -4  -3  -2  -1
"""
```

```
numbers[0::2]   # => [0, 2, 4]
numbers[:]  # => [0, 1, 2, 3, 4, 5]
numbers[1:]  # => [1, 2, 3, 4, 5]
numbers[-2:]  # => [4, 5]
numbers[1:4]  # => [1, 2, 3]
numbers[1:1]  # => []
numbers[-99:99]  # => [0, 1, 2, 3, 4, 5]
numbers[::-1]  # => [5, 4, 3, 2, 1, 0]
numbers[::-2]  # => [5, 3, 1]
```

Slices with `step` = 1 are called **Basic Slice**.
Slices with `step != 1` are called **Extended Slice**.

List Mutation

List Mutation

`list.append(x)`: Append x to end of the sequence

List Mutation

`list.append(x)`: Append x to end of the sequence
`list.insert(i, x)`: Insert x to index i

List Mutation

`list.append(x)`: Append x to end of the sequence
`list.insert(i, x)`: Insert x to index i
`list.pop(i=-1)`: Remove and return element at index i

List Mutation

`list.append(x)`: Append x to end of the sequence
`list.insert(i, x)`: Insert x to index i
`list.pop(i=-1)`: Remove and return element at index i
`list.remove(x)`: Remove first occurrence of x

List Mutation

`list.append(x)`: Append x to end of the sequence
`list.insert(i, x)`: Insert x to index i
`list.pop(i=-1)`: Remove and return element at index i
`list.remove(x)`: Remove first occurrence of x
`list.extend(iterable)`: Add all elements in iterable to end of list

List Mutation

`list.append(x)`: Append x to end of the sequence
`list.insert(i, x)`: Insert x to index i
`list.pop(i=-1)`: Remove and return element at index i
`list.remove(x)`: Remove first occurrence of x
`list.extend(iterable)`: Add all elements in iterable to end of list
`list[i] = new_value`: Update value of index i with new value

List Mutation

`list.append(x)`: Append x to end of the sequence
`list.insert(i, x)`: Insert x to index i
`list.pop(i=-1)`: Remove and return element at index i
`list.remove(x)`: Remove first occurrence of x
`list.extend(iterable)`: Add all elements in iterable to end of list
`list[i] = new_value`: Update value of index i with new value
`list[basic_slice] = iterable`: Change elements in basic slice with elements in iterable, sizes can be different: `numbers[:] = []`

List Mutation

`list.append(x)`: Append x to end of the sequence
`list.insert(i, x)`: Insert x to index i
`list.pop(i=-1)`: Remove and return element at index i
`list.remove(x)`: Remove first occurrence of x
`list.extend(iterable)`: Add all elements in iterable to end of list
`list[i] = new_value`: Update value of index i with new value
`list[basic_slice] = iterable`: Change elements in basic slice with elements in iterable, sizes can be different: `numbers[:] = []`
`list[extended_slice] = iterable`: Change elements in extended slice with elements in iterable 1-1, sizes must be equal.

Some Other List Operations

`in` operator: Check whether an element is in list. `3 in numbers` ⇒ `True`
`len(list)`: Returns the length of list(and other collections).
`list.index(value, start=0, stop=len(list))`: Return first index of value.
`list.count(value)`: Count number of occurrences of value in list.
`list.reverse()`: Reverse the list (in-place)
`list.sort()`: Sort list elements (in-place)
For more, type `help(list)` in your interactive interpreter.

Strings

Special kind of **lists**!

Strings

Special kind of **lists**! `name = 'Ahmet'`

Strings

Special kind of **lists**! `name = 'Ahmet'`
You can do:

- Indexing: `name[2]` $\Rightarrow$ 'm'

Strings

Special kind of **lists**! `name = 'Ahmet'`
You can do:

- Indexing: `name[2]` $\Rightarrow$ 'm'
- Slicing: `name[::-1]` $\Rightarrow$ 'temhA'

Strings

Special kind of **lists**! `name = 'Ahmet'`
You can do:

- Indexing: `name[2]` ⇒ 'm'
- Slicing: `name[::-1]` ⇒ 'temhA'
- Search by `in` operator: `'hm' in name` ⇒ `True`

Strings

Special kind of **lists**! `name = 'Ahmet'`
You can do:

- Indexing: `name[2]` ⇒ 'm'
- Slicing: `name[::-1]` ⇒ 'temhA'
- Search by `in` operator: `'hm'in name` ⇒ `True`

You can not do:

- String mutation: name[2]='H' ⇒ **`TypeError`**

Strings

Special kind of **lists**! `name = 'Ahmet'`
You can do:

- Indexing: `name[2]` ⇒ 'm'
- Slicing: `name[::-1]` ⇒ 'temhA'
- Search by `in` operator: `'hm'in name` ⇒ `True`

You can not do:

- String mutation: name[2]='H' ⇒ **`TypeError`**

Special functions about strings: `str.isnumeric()`, `str.capitalize()`, `str.format(...)`, `str.find()` ...

Loops

Do something for many elements or based on a condition.

```
while <condition>:
    <expression>
    <expression>
    ...
```

Similar to simple if blocks, but runs again and again until condition check fails.

```
for <item> in <iterable>:
    <expression>
    <expression>
    ...
```

Iterable: collection of **ordered** elements.
What is next after this item?

For Loops

What is next after this item?
numbers[1] is after numbers[0]

For Loops

What is next after this item?
numbers[1] is after numbers[0] $\neq$ **numbers[1]** $>$ **numbers[0]**

For Loops

What is next after this item?
numbers[1] is after numbers[0] $\neq$ **numbers[1]** $>$ **numbers[0]**
Examples of iterables:

For Loops

What is next after this item?
numbers[1] is after numbers[0] $\neq$ **numbers[1]** $>$ **numbers[0]**
Examples of iterables: lists,

For Loops

What is next after this item?
numbers[1] is after numbers[0] $\neq$ **numbers[1]** $>$ **numbers[0]**
Examples of iterables: lists, strings,

For Loops

What is next after this item?
numbers[1] is after numbers[0] $\neq$ **numbers[1]** $>$ **numbers[0]**
Examples of iterables: lists, strings, ranges

For Loops

What is next after this item?
numbers[1] is after numbers[0] $\neq$ **numbers[1]** $>$ **numbers[0]**
Examples of iterables: lists, strings, ranges

Ranges

`range(start, stop, step)`: creates a sequence of integers from start (inclusive) to stop (exclusive) by step.
Can be **indexed** and **sliced**
`len()` and `in` operator can be used

For Loops

```
names = [Mario, Peter, Anna ,Paul ,Anna]

for number in range(2, 5):
    # In every iteration, we a have a different value from iterable
    # We can access the value with the name we specified
    print(number)
    # range is collection of integers, we can use ints in indexing
    print('Hello {}'.format(names[number]))

    # Nested loops
    for name in names:
        # In every iteration name changes, in the order of names
        if name != names[number]:
            print('{} says hello to {}'.format(name, names[number]))
```

Break, Continue & Pass

Break terminates the closest for or while loop

```
for i in range(0, 5):
    if i % 2 == 1:
        break
    print(i)
```

```
x = 1
while x < 100:
    x *= 2
    if (x+1) % 3 == 0:
        break
    print(x)
```

Continue continues with the next iteration of the loop

```
for i in range(0, 5):
    if i % 2 == 1:
        continue
    print(i)
```

```
x = 1
while x < 100:
    x *= 2
    if (x+1) % 3 == 0:
        continue
    print(x)
```

Break, Continue & Pass

Pass does not have an effect

```
for letter in 'Python':
   if letter == 'y':
      pass
      print ('In pass case')
   print(letter)
```

- Loops, conditional statements, functions etc. cannot be empty
- Use when you have to create one

For Else, While Else??

Questions?