Comparisons and Control Flow

Chris Cornwell

Jan 2, 2025

Outline

Outline

Comparisons

We often need to compare variables, or compare a variable to some value – a “literal.”

Comparisons

We often need to compare variables, or compare a variable to some value – a “literal.”
The comparison operations are in the table below.

`==`	`!=`	`<`	`<=`	`>`	`>=`	`is`	`is not`

== and `!=`, for *equals* and *is not equal*.

<=, for *less than or equal to* (and similarly with >= and greater than).

Comparisons

We often need to compare variables, or compare a variable to some value – a "literal."
The comparison operations are in the table below.

`==`	`!=`	`<`	`<=`	`>`	`>=`	`is`	`is not`

== and `!=`, for *equals* and *is not equal*.

<=, for *less than or equal to* (and similarly with >= and greater than).

- Error occurs if a comparison doesn't make sense for that type.
- Don't use `is` or `is not` with literals – meaning, they compare values of two variables but not, for example, if variable `x` equals `5`. Use either == or `!=` for that.

Comparisons, examples

Below, an example that returns True, but produces a syntax warning.

```
v, w = 10, 5
x = w+5
# A warning from next line; read warning message
print(x is 10)
```

Comparisons, examples

Below, an example that returns True, but produces a syntax warning.

```python
v, w = 10, 5
x = w+5
# A warning from next line; read warning message
print(x is 10)
```

With the same variable assignments as above, the following prints `True` twice.

```python
print(v is x)
x == 10
```

Outline

Control Statements

So far, all code has had simple structure – a few lines of code each get executed once in order, top to bottom. Need a less simple structure to do interesting things.

Achieve this with **control flow** statements.

Control Statements

So far, all code has had simple structure – a few lines of code each get executed once in order, top to bottom. Need a less simple structure to do interesting things.

Achieve this with **control flow** statements.

Right now, we'll look at the following.

- `if-else` statements.

Control Statements

So far, all code has had simple structure – a few lines of code each get executed once in order, top to bottom. Need a less simple structure to do interesting things.

Achieve this with **control flow** statements.

Right now, we'll look at the following.

- `if-else` statements.
- Loops: `for` loops.
- Loops: `while` loops.

`if-else` statements

When you want part of code to execute only when a condition is satisfied, use an `if-else` statement.

`if-else` statements

When you want part of code to execute only when a condition is satisfied, use an `if-else` statement.
A basic example below (v=10 and w=5 as in slide from Comparisons).

```python
# basic if - else statement structure
Class = "MATH 371"
if len(Class) > v:
    y = w-5
    print("This sure is a long Class!")
else:
    y = w+10
    print("This Class just flies by!")
```

Notice the indentation of the lines between `if` and `else`, and those after `else`, which determines the code that runs on the condition.

`if-else` statements

When you want part of code to execute only when a condition is satisfied, use an `if-else` statement.
A basic example below (v=10 and w=5 as in slide from Comparisons).

```
1  # basic if - else statement structure
2  Class = "MATH 371"
3  if len(Class) > v:
4      y = w-5
5      print("This sure is a long Class!")
6  else:
7      y = w+10
8      print("This Class just flies by!")
```

Notice the indentation of the lines between `if` and `else`, and those after `else`, which determines the code that runs on the condition.

> Lines 7 and 8 will only be executed if `len(Class)` is less than 11. If a line after line 8 is *not* indented, it will be executed no matter what.

`elif` and `pass`

What if we have more than just two cases? That is, when the `if` condition fails we want to check another condition before deciding what to do.

`elif` and `pass`

What if we have more than just two cases? That is, when the `if` condition fails we want to check another condition before deciding what to do.

Put an `elif` block between the `if` and `else`. (This is short for “else, if…”.)

```
if y < w:
    print(f'y is less than w: {y} < {w}.')
elif y < 2*v:
    print(f'y ≥ w and less than 2v: {w} ≤ {y} < {2*v}.')
else:
    print(f'y ≥ 2v: {y} ≥ {2*v}.')
```

elif and pass

What if we have more than just two cases? That is, when the `if` condition fails we want to check another condition before deciding what to do.

Put an `elif` block between the `if` and `else`. (This is short for “else, if...”.)

```
if y < w:
    print(f'y is less than w: {y} < {w}.')
elif y < 2*v:
    print(f'y ≥ w and less than 2v: {w} ≤ {y} < {2*v}.')
else:
    print(f'y ≥ 2v: {y} ≥ {2*v}.')
```

If we only care to do something in one of the cases, put `pass` in the block for the other case.

```
if condition_to_be_checked:
    # some code here that does something
else:
    pass
```

Loops: `for` loop construction

Use a *loop* to construct, or compute, something through various iterations – the lines in a block of code execute over and over again until, at some point, it finishes.
The block of code that repeats should be indented, as with `if-else` statements. The next line not indented is outside that loop.

Loops: `for` loop construction

Use a *loop* to construct, or compute, something through various iterations – the lines in a block of code execute over and over again until, at some point, it finishes.

The block of code that repeats should be indented, as with `if-else` statements. The next line not indented is outside that loop.

Example:

```python
# for loop that adds integers 1 through 5
the_sum = 0
for i in [1,2,3,4,5]:
    the_sum += i
print(the_sum)
```

Loops: `for` loop construction

Use a *loop* to construct, or compute, something through various iterations – the lines in a block of code execute over and over again until, at some point, it finishes.

The block of code that repeats should be indented, as with `if-else` statements. The next line not indented is outside that loop.

Example:

```python
# for loop that adds integers 1 through 5
the_sum = 0
for i in [1,2,3,4,5]:
    the_sum += i
print(the_sum)
```

To avoid writing out the list of values for `i`, the following does the same as previous.

```python
the_sum = 0
for i in range(1,6):
    the_sum += i
print(the_sum)
```

The next slide is about the `range()` function.

More about `range()` constructor

`range` is a type; it is similar to a list of ints. The function `range()` creates a range object (an instance of it).
This function uses three arguments – `start`, `stop`, and `step`. As a list, `range(start, stop, step)` will contain integers between `start` and `stop-1`, with a gap of `step` between consecutive items.

[1]The behavior that has `start=0` and `step=1` here, we say these arguments have *default* values of 0 and 1.

More about `range()` constructor

`range` is a type; it is similar to a list of ints. The function `range()` creates a range object (an instance of it).
This function uses three arguments – `start`, `stop`, and `step`. As a list, `range(start, stop, step)` will contain integers between `start` and `stop-1`, with a gap of `step` between consecutive items.
For example, if `step = 1`, the range is

$$\left[\texttt{start}, \texttt{start+1}, \texttt{start+2}, \ldots, \texttt{stop-1}\right].$$

[1]The behavior that has $\texttt{start=0}$ and $\texttt{step=1}$ here, we say these arguments have *default* values of 0 and 1.

More about `range()` constructor

`range` is a type; it is similar to a list of ints. The function `range()` creates a range object (an instance of it).
This function uses three arguments – `start`, `stop`, and `step`. As a list, `range(start, stop, step)` will contain integers between `start` and `stop-1`, with a gap of `step` between consecutive items.
For example, if `step = 1`, the range is

$$[\texttt{start}, \texttt{start+1}, \texttt{start+2}, \ldots, \texttt{stop-1}].$$

More generally, its largest item equals `start + k*step`, with `k` as large as possible to remain less than `stop`. For example, `range(2, 30, 4)` is [2,6,10,14,18,22,26] as a list.

[1]The behavior that has `start=0` and `step=1` here, we say these arguments have *default* values of 0 and 1.

More about `range()` constructor

`range` is a type; it is similar to a list of ints. The function `range()` creates a range object (an instance of it).

This function uses three arguments – `start`, `stop`, and `step`. As a list, `range(start, stop, step)` will contain integers between `start` and `stop-1`, with a gap of `step` between consecutive items.

For example, if `step = 1`, the range is

$$[\mathtt{start}, \mathtt{start+1}, \mathtt{start+2}, \ldots, \mathtt{stop-1}].$$

More generally, its largest item equals `start + k*step`, with `k` as large as possible to remain less than `stop`. For example, `range(2, 30, 4)` is [2,6,10,14,18,22,26] as a list.

Given only two inputs, `range()` uses them as `start` and `stop`, setting `step=1`. A single input is interpreted as `stop`, setting `start=0`, `step=1`.[1]

So, `range(6)` is [0,1,2,3,4,5].

[1]The behavior that has `start=0` and `step=1` here, we say these arguments have *default* values of 0 and 1.

Other sequence types and for loops

A great thing: you are not *required* to use some range object in a `for` loop. You can replace it with anything of sequential type.

- When it makes sense, can make code better for reading!

Other sequence types and for loops

A great thing: you are not *required* to use some range object in a `for` loop. You can replace it with anything of sequential type.

- When it makes sense, can make code better for reading!

Example: say that in the variable `Class`, assigned to be `"MATH 371"` before, we're inserting a period `"."` after each letter (but, not after a space). A for loop to do it is below.

```python
new_Class_string = ""
for c in Class:
    if c == " ":
        new_Class_string += c
    else:
        new_Class_string += c + "."
print(new_Class_string)
```

Loops: `while` loop construction

Rather than repeating a block of code for items in a specified sequence, we can simply repeat it until a condition is satisfied – a `while` loop.

Loops: `while` loop construction

Rather than repeating a block of code for items in a specified sequence, we can simply repeat it until a condition is satisfied – a `while` loop.

Above the code block to be repeated, put: `while <condition>`, where `<condition>` should evaluate to either `True` or `False`.
Example:

```python
# Fibonacci numbers less than 10000
fibo_list = [1,1]
while fibo_list[-2] + fibo_list[-1] < 10000:
    fibo_list += [ fibo_list[-2] + fibo_list[-1] ]
print(fibo_list)
```

Outline

Ways of constructing lists

To generate or construct a list with a loop, a common way involves two parts.

1. Make an empty list.
2. Go through a loop, adding items to the list in each (or some) of the iterations.

Ways of constructing lists

To generate or construct a list with a loop, a common way involves two parts.

1. Make an empty list.
2. Go through a loop, adding items to the list in each (or some) of the iterations.

Example: have a function, `item()`, that determines the $\mathtt{n}^{th}$ item in the list. You want list to contain the first 50 items. The code could look like below.

```python
some_list = []
for n in range(50):
    # figure out the item with function item(n)
    some_list += [item(n)]
```

Ways of constructing lists

To generate or construct a list with a loop, a common way involves two parts.

1. Make an empty list.
2. Go through a loop, adding items to the list in each (or some) of the iterations.

Example: have a function, `item()`, that determines the `n`*th* item in the list. You want list to contain the first 50 items. The code could look like below.

```python
some_list = []
for n in range(50):
    # figure out the item with function item(n)
    some_list += [item(n)]
```

In Python, the same list can be created in one line of code. You do so by typing the following.

```python
some_list = [item(n) for n in range(50)]
```

Ways of constructing lists, continued

In the previous slide, the variable `some_list` is assigned by a *list comprehension*.

```python
some_list = [item(n) for n in range(50)]
```

Ways of constructing lists, continued

In the previous slide, the variable `some_list` is assigned by a *list comprehension*.

```
some_list = [item(n) for n in range(50)]
```

In list comprehensions: can add a condition to check. For example, if you only want to include `item(n)` if it is even, then you can write the following.

```
some_list = [item(n) for n in range(50) if item(n) % 2 == 0]
```

Ways of constructing lists, continued

In the previous slide, the variable `some_list` is assigned by a *list comprehension*.

```
some_list = [item(n) for n in range(50)]
```

In list comprehensions: can add a condition to check. For example, if you only want to include `item(n)` if it is even, then you can write the following.

```
some_list = [item(n) for n in range(50) if item(n) % 2 == 0]
```

A more complicated example: get those x between 1 and 100 at which the function $1000\frac{x}{(x^3+1)}$ has value between 0.5 and 2.

```
[x for x in range(1,101) if 0.5 < 1000*x/(x**3+1) < 2]
```