Python syntax cheat sheet

April 22, 2025

1 Python syntax cheat sheet

This notebook demonstrates some basic syntax rules of the Python programming language.

1.0.1 Basic data types

Just like Excel and other data processing software, Python recognizes a variety of data types, including three we'll look at here: - Strings (text) - Numbers (integers, numbers with decimals and more) - Booleans (`True` and `False`).

You can use the built-in `type()` function to check the data type of a value.

Strings A string is a group of characters – letters, numbers, whatever – enclosed within single or double quotes (doesn't matter as long as they match). The code in these notebooks uses single quotes. (The Python style guide doesn't recommend one over the other: "Pick a rule and stick to it.")

If your string *contains* apostrophes or quotes, you have two options: *Escape* the offending character with a forward slash \:

```
'Isn\'t it nice here?'
```

… or change the surrounding punctuation:

```
"Isn't it nice here?"
```

The style guide recommends the second approach.

When you call the `type()` function on a string, Python will return `str`.

Calling the `str()` function on a value will return the string version of that value.

```
[ ]: 'Investigative Reporters and Editors'
```

```
[ ]: type('hello!')
```

```
[ ]: 45
```

```
[ ]: type(45)
```

```
[ ]: str(45)
```

```
[ ]: type(str(45))
```

```
[ ]: str(True)
```

If you "add" strings together with a plus sign `+`, it will concatenate them:

```
[ ]: 'IRE' + '/' + 'NICAR'
```

Numbers and math Python recognizes a variety of numeric data types. Two of the most common are integers (whole numbers) and floats (numbers with decimals).

Calling `int()` on a piece of numeric data (even if it's being stored as a string) will attempt to coerce it to an integer; calling `float()` will try to convert it to a float.

```
[ ]: 12
```

```
[ ]: 12.4
```

```
[ ]: type(12)
```

```
[ ]: type(12.4)
```

```
[ ]: int(35.6)
```

```
[ ]: int('45')
```

```
[ ]: float(46)
```

```
[ ]: float('45')
```

You can do basic math in Python. You can also do more advanced math.

```
[ ]: 4+2
```

[ ]:
```
10-9
```

[ ]:
```
5*10
```

[ ]:
```
1000/10
```

[ ]:
```
# ** raises a number to the power of another number
5**2
```

Booleans Just like in Excel, which has `TRUE` and `FALSE` data types, Python has boolean data types. They are `True` and `False` – note that only the first letter is capitalized, and they are not sandwiched between quotes.

Boolean values are typically returned when you're evaluating some sort of conditional statement – comparing values, checking to see if a string is inside another string or if a value is in a list, etc.

Python's comparison operators include:

- `>` greater than
- `<` less than
- `>=` greater than or equal to
- `<=` less than or equal to
- `==` equal to
- `!=` not equal to

[ ]:
```
True
```

[ ]:
```
False
```

[ ]:
```
4 > 6
```

[ ]:
```
10 == 10
```

[ ]:
```
'crapulence' == 'Crapulence'
```

[ ]:
```
type(True)
```

1.0.2 Variable assignment

The `=` sign assigns a value to a variable name that you choose. Later, you can retrieve that value by referencing its variable name. Variable names can be pretty much anything you want (as long as you follow some basic rules).

This can be a tricky concept at first! For more detail, here's a pretty good explainer from Digital Ocean.

[ ]:
```
my_name = 'Frank'
```

[ ]:
```
my_name
```

You can also *reassign* a different value to a variable name, though it's usually better practice to create a new variable.

[ ]:
```
my_name = 'Susan'
```

[ ]:
```
my_name
```

A common thing to do is to "save" the results of an expression by assigning the result to a variable.

[ ]:
```
my_fav_number = 10 + 3
```

[ ]:
```
my_fav_number
```

It's also common to refer to previously defined variables in an expression:

[ ]:
```
nfl_teams = 32
mlb_teams = 30
nba_teams = 30
nhl_teams = 31

number_of_pro_sports_teams = nfl_teams + mlb_teams + nba_teams + nhl_teams
```

[ ]:
```
number_of_pro_sports_teams
```

1.0.3 String methods

Let's go back to strings for a second. String objects have a number of useful methods – let's use an example string to demonstrate a few common ones.

[ ]:
```
my_cool_string = '    Hello, friends!'
```

`upper()` converts the string to uppercase:

[ ]:
```
my_cool_string.upper()
```

`lower()` converts to lowercase:

[ ]:
```
my_cool_string.lower()
```

`replace()` will replace a piece of text with other text that you specify:

[ ]:
```
my_cool_string.replace('friends', 'enemies')
```

`count()` will count the number of occurrences of a character or group of characters:

[ ]:
```
my_cool_string.count('H')
```

Note that `count()` is case-sensitive. If your task is "count all the e's," convert your original string to upper or lowercase first:

[ ]:
```
my_cool_string.upper().count('E')
```

split() will split the string into a list (more on these in a second) on a given delimiter (if you don't specify a delimiter, it'll default to splitting on a space):

[ ]:
```
my_cool_string.split()
```

[ ]:
```
my_cool_string.split(',')
```

strip() removes whitespace from either side of your string (but not internal whitespace):

[ ]:
```
my_cool_string.strip()
```

You can use a cool thing called "method chaining" to combine methods – just tack 'em onto the end. Let's say we wanted to strip whitespace from our string *and* make it uppercase:

[ ]:
```
my_cool_string.strip().upper()
```

Notice, however, that our original string is unchanged:

[ ]:
```
my_cool_string
```

Why? Because we haven't assigned the results of anything we've done to a variable. A common thing to do, especially when you're cleaning data, would be to assign the results to a new variable:

[ ]:
```
my_cool_string_clean = my_cool_string.strip().upper()
```

[ ]:
```
my_cool_string_clean
```

1.0.4 Comments

A line with a comment – a note that you don't want Python to interpret – starts with a # sign. These are notes to collaborators and to your future self about what's happening at this point in your script, and why.

Typically you'd put this on the line right above the line of code you're commenting on:

[ ]:
```
avg_settlement = 40827348.34328237

# coercing this to an int because we don't need any decimal precision
int(avg_settlement)
```

Multi-line comments are sandwiched between triple quotes (or triple apostrophes):

`''' this is a long comment '''`

or

`""" this is a long comment """`

1.0.5 The `print()` function

So far, we've just been running the notebook cells to get the last value returned by the code we write. Using the `print()` function is a way to print specific things in your script to the screen. This function is handy for debugging.

To print multiple things on the same line, separate them with a comma.

```
[ ]: print('Hello!')
```

```
[ ]: print(my_name)
```

```
[ ]: print('Hello,', my_name)
```

1.1 Collections of data

Now we're going to talk about two ways you can use Python to group data into a collection: lists and dictionaries.

1.1.1 Lists

A *list* is a comma-separated list of items inside square brackets: `[]`.

Here's a list of ingredients, each one a string, that together makes up a salsa recipe.

```
[ ]: salsa_ingredients = ['tomato', 'onion', 'jalapeño', 'lime', 'cilantro']
```

To get an item out of a list, you'd refer to its numerical position in the list – its *index* (1, 2, 3, etc.) – inside square brackets immediately following your reference to that list. In Python, as in many other programming languages, counting starts at 0. That means the first item in a list is item `0`.

```
[ ]: salsa_ingredients[0]
```

```
[ ]: salsa_ingredients[1]
```

You can use *negative indexing* to grab things from the right-hand side of the list – and in fact, `[-1]` is a common idiom for getting "the last item in a list" when it's not clear how many items are in your list.

```
[ ]: salsa_ingredients[-1]
```

If you wanted to get a slice of multiple items out of your list, you'd use colons (just like in Excel, kind of!).

If you wanted to get the first three items, you'd do this:

```
[ ]: salsa_ingredients[0:3]
```

You could also have left off the initial 0 – when you leave out the first number, Python defaults to "the first item in the list." In the same way, if you leave off the last number, Python defaults to "the last item in the list."

[ ]:
```
salsa_ingredients[:3]
```

Note, too, that this slice is giving us items 0, 1 and 2. The `3` in our slice is the first item we *don't* want. That can be kind of confusing at first. Let's try a few more:

[ ]:
```
# everything in the list except the first item
salsa_ingredients[1:]
```

[ ]:
```
# the second, third and fourth items
salsa_ingredients[1:4]
```

[ ]:
```
# the last two items
salsa_ingredients[-2:]
```

To see how many items are in a list, use the `len()` function:

[ ]:
```
len(salsa_ingredients)
```

To add an item to a list, use the `append()` method:

[ ]:
```
salsa_ingredients
```

[ ]:
```
salsa_ingredients.append('mayonnaise')
```

[ ]:
```
salsa_ingredients
```

Haha *gross*. To remove an item from a list, use the `pop()` method. If you don't specify the index number of the item you want to pop out, it will default to "the last item."

[ ]:
```
salsa_ingredients.pop()
```

[ ]:
```
salsa_ingredients
```

You can use the `in` and `not in` expressions to test membership in a list (will return a boolean):

[ ]:
```
'lime' in salsa_ingredients
```

[ ]:
```
'cilantro' not in salsa_ingredients
```

1.1.2 Dictionaries

A *dictionary* is a comma-separated list of key/value pairs inside curly brackets: `{}`. Let's make an entire salsa recipe:

[ ]:
```
salsa = {
    'ingredients': salsa_ingredients,
    'instructions': 'Chop up all the ingredients and cook them for awhile.',
    'oz_made': 12
}
```

To retrieve a value from a dictionary, you'd refer to the name of its key inside square brackets `[]` immediately after your reference to the dictionary:

```
[ ]: salsa['oz_made']
```

```
[ ]: salsa['ingredients']
```

To add a new key/value pair to a dictionary, assign a new key to the dictionary inside square brackets and set the value of that key with `=`:

```
[ ]: salsa['tastes_great'] = True
```

```
[ ]: salsa
```

To delete a key/value pair out of a dictionary, use the `del` command and reference the key:

```
[ ]: del salsa['tastes_great']
```

```
[ ]: salsa
```

1.1.3 Indentation

Whitespace matters in Python. Sometimes you'll need to indent bits of code to make things work. This can be confusing! `IndentationError`s are common even for experienced programmers. (FWIW, Jupyter will try to be helpful and insert the correct amount of "significant whitespace" for you.)

You can use tabs or spaces, just don't mix them. The Python style guide recommends indenting your code in groups of four spaces, so that's what we'll use.

1.1.4 `for` loops

You would use a `for` loop to iterate over a collection of things. The statement begins with the keyword `for` (lowercase), then a temporary `variable_name` of your choice to represent each item as you loop through the collection, then the Python keyword `in`, then the collection you're looping over (or its variable name), then a colon, then the indented block of code with instructions about what to do with each item in the collection.

Let's say we have a list of numbers that we assign to the variable `list_of_numbers`.

```
[ ]: list_of_numbers = [1, 2, 3, 4, 5, 6]
```

We could loop over the list and print out each number:

```
[ ]: for number in list_of_numbers:
         print(number)
```

We could print out each number *times 6*:

```
[ ]: for number in list_of_numbers:
         print(number*6)
```

… whatever you need to do in you loop. Note that the variable name `number` in our loop is totally arbitrary. This also would work:

```
[ ]: for banana in list_of_numbers:
         print(banana)
```

It can be hard, at first, to figure out what's a "Python word" and what's a variable name that you get to define.

Strings are iterable, too. Let's loop over the letters in a sentence:

```
[ ]: sentence = 'Hello, IRE/NICAR!'

     for letter in sentence:
         print(letter)
```

Because strings are iterable, like lists, you can use the same kinds of methods:

```
[ ]: # get the first five characters
     sentence[:5]
```

```
[ ]: # get the length of the sentence
     len(sentence)
```

```
[ ]: 'Hello' in sentence
```

You can iterate over dictionaries, too – just remember that dictionaries *don't keep track of the order that items were added to it.* (For that, you'd want an `OrderedDict`.

When you're looping over a dictionary, the variable name in your `for` loop will refer to the keys. Loop over the `salsa` dictionary:

```
[ ]: for key in salsa:
         print(key)
```

To get the *value* of a dictionary item in a for loop, use the key to retrieve it from the dictionary:

```
[ ]: for key in salsa:
         print(salsa[key])
```

1.1.5 `if` statements

Just like in Excel, you can use the "if" keyword to handle conditional logic.

These statements begin with the keyword `if` (lowercase), then the condition to evaluate, then a colon, then a new line with a block of indented code to execute if the condition resolves to `True`.

[ ]:
```python
if 4 < 6:
    print('4 is less than 6')
```

You can also add an `else` statement (and a colon) with an indented block of code you want to run if the condition resolves to `False`.

[ ]:
```python
if 4 > 6:
    print('4 is greater than 6?!')
else:
    print('4 is not greater than 6.')
```

If you need to, you can add multiple conditions with `elif`.

[ ]:
```python
HOME_SCORE = 6
AWAY_SCORE = 8

if HOME_SCORE > AWAY_SCORE:
    print('we won!')
elif HOME_SCORE == AWAY_SCORE:
    print('we tied!')
else:
    print('we lost!')
```