Programming for Data Science Notes

John Michael Epperson

Contents

Chapter 1

About

These are my notes for DS 5100: Programming for Data Science, Fall 2024 as taught by Professor Raf Alvarado.

Chapter 2

Introducing Python

Data structures in a nutshell:

Data Structure	Ordered	Mutable	Constructor	Example
List	Yes	Yes	`[]` or `list()`	`[5.7,4,'yes'5.7]`
Tuple	Yes	No	`()` or `tuple()`	`(5.7,4,'yes'5.7)`
Set	No	Yes	`{}` or `set()`	`{5.7, 4, 'yes'}`
Dictionary	No	No	`{ }` or `dict()`	`{'Jun':75,'Jul':89}`

	Ordered	Unordered
Mutable	List `[]`	Set `{}`
Immutable	Tuple `()`	Dict `{ }`

Chapter 3

Control Structures

Structures control the flow of data in a program.

These structures are made of code blocks that provide **looping and branching** capabilities to your code, based on **boolean conditions**.

3.1 Control Structures General

3.1.1 Conditions

I think you already know this, here's a simple example:

```
val = -2

if val >= 0:
    print(val)
else:
    print(-val)
```

```
## 2
```

3.1.2 Indentation

Is essential to a python program.

- Python gets rid of the curly brackets and `IF...END IF` statements of c-like languages and replaces them with a consistent indentation pattern. This created more readable, easier to type code.

3.1.2.1 `elif`

`elif` is reached when previous statements are not.

Unlike a series of `if` statements, an `elif` statement is **only** processed if none of the preceding 'if' statements are **not processed**. See example:

```
val = -2

if -10 < val < -5:
    print('bucket 1')
elif -5 <= val < -2:
    print('bucket 2')
elif val == -2:
    print('bucket 3')
```

```
## bucket 3
```

3.1.2.2 `else`

`else` can be used as a catchall for situations where no condition is met. Put it at the end.

It's good practice to include and `else` statement.

3.1.2.3 `if` and `else` statements as one-liners

Python provides a short-hand way of defining `if` statements:

```
x = 3

print('odd') if x % 2 == 1 else print('even')
```

```
## odd
```

The general form:

`ACTION1 if CONDITION else ACTION2`

both `if` and `else` are required. Otherwise, code machine broke…

```
print('odd') if x % 2 == 1
```

```
## expected 'else' after 'if' expression (<string>, line 1)
```

Basically it's either/or

3.1.2.4 Using mulitple conditions

I think you got this one chief. Make sure you use parentheses to keep order of operations in tact.

```
print(1) if (-2 < val < 2) or (val > 10) else print(2)
```

```
## 2
```

3.1.3 Loops

Loops are **fundamental** to constructs in programming.

The repitition of a process as the data changes is called **iteration**.

Loops exemplify a **premise** of this course that data structures imply algorithms. Using the term 'list' broadly, we may say that **lists imply loops**.

The two main kinds of loops in python are `while` and `for`.

3.1.3.1 `while` loops

While loops iterate 'while' some condition is true. they stop when the condition is false.

Example:

```
ix = 1
ix_list = []
while ix < 10:
    ix *= 2
    ix_list.append(ix)
print(ix)
```

```
## 16
```

```
print(ix_list)
```

```
## [2, 4, 8, 16]
```

```
print(ix_list[-1])
```

```
## 16
```

Note that it is possible for while loops to be unending.

For example:

`while 1: \  print("This is so annoying")`

This is why it's important to make sure your looping condition can be met. You may also break out of a loop by other means.

3.1.3.2 `break`

Sometimes you want to quit the loop early, if some condition is met.

To do this, put `break` in the `if` statement.

```
ix = 1
while ix < 10:
    ix = ix * 2
    if ix == 4:
        break
print(ix)
```

```
## 4
```

The `break` causes the loop to end early.

3.1.3.3 `continue`

Sometimes you want to introduce skipping behavior in loops.

In this case, use `continue` in `if` statement.

```
ix = 1
while ix < 10:
    ix = ix * 2
    if ix == 4:
        print('skipping 4...')
        continue
    print(ix)
```

```
## 2
## skipping 4...
## 8
## 16
```

3.1.3.4 `for`

In contrast to `while` loops, `for` loops iterate over an iterable data structure, such as a list.

They stop once the list is finished. See example below:

```
cities = ['Charlottesville','New York','SF','BOS','LA']

for city in cities:
    print(f'"{city.lower()}"', end=' ')
```

```
## "charlottesville" "new york" "sf" "bos" "la"
```

Conditions can be placed inside `if` statements to skip within or stop the loop.

e.g.: Quit early if `SF` reached, using `break`:

```
cities = ['Charlottesville','New York','SF','BOS','LA']

for city in cities:
    if city == 'SF':
        break
    print(f'"{city.lower()}"', end=' ')
```

```
## "charlottesville" "new york"
```

Skip over `SF` if reached, using `continue`:

```
cities = ['Charlottesville','New York','SF','BOS','LA']

for city in cities:
    if city == 'SF':
        continue
    print(f'"{city.lower()}"', end=' ')
```

```
## "charlottesville" "new york" "bos" "la"
```

3.1.4 `while` vs `for`

When choosing between a `for` and `while` loop, consider the following:

- `for` loops are used to loop through a list of values or an operation in which the number of iteration is **known** in advance.

- `while` loops are used when you **don't know** how many iterations it will take - you are depending on some condition to be met.

The former is often used when **processing** data, the latter whe performing algorithmic **modeling** tasks, such as optimizing and convergence.

3.2 Iterables and Iterators

Sequential data structures like lists and tuples have a **natural affinity** to loops.

Sequences imply loops, and loops expect sequences.

In python this relationship is captured by the resonance between the words **iteration** and **iterables**.

Iterable data structures that can be iterated over, meaning they can return their elements one at a time.

Examples of iterable objects include lists, tuples, sets, dicts, and strings.

Typically we iterate over iterables using `for` **loops**, as we saw when we reviewed control structures.

3.2.1 Lists

3.2.1.1 iterating using `for`

review of iteration by means of a `for` loop

```
tokens = ['living room', 'was', 'quite', 'large']
for tok in tokens:
    print(tok)
```

```
## living room
## was
## quite
## large
```

3.2.1.2 Iterators

An **iterator** is used to iterate over iterable objects by removing one element at a time from the iterables.

3.2.1.3 Iterating with Iterators

An iterator works by popping out and removing a value at each iteration.

This means that when iterating through an iterable object you empty as you go, leaving an empty data structure at the end.

This is useful in situations where you want to save memory.

Many functions in python return iterables so it's helpful to understand them even if you don't create any yourself.

3.2.1.4 Using `iter()` and `next()`

convert a sequence to an iterator object using `iter()`

Then use `next()` to get the next item from the iterator.

```
tokens = ['living room','was','quite','large']
myit = iter(tokens)
print(next(myit))
```

```
## living room
```

```
print(next(myit))
```

```
## was
```

```
print(next(myit))
```

```
## quite
```

```
print(next(myit))
```

```
## large
```

Calling `next()` when the iterator has reached the end of the list produces an exception:

```
next(myit)
```

```
## StopIteration
```

Note that when used witha n iterable created by `iter()`, `for` implicitly executes `next()` on each loop iteration.

```
myit = iter(tokens) # Reset the iterator
for next_it in myit:
    print(next_it)
```

```
## living room
## was
## quite
## large
```

3.2.2 Sequences and Collections

So far, we've iterated over a list; now we'll look at sets, strings, tuples, dicts, and ranges!

Lists, tuples, and strings are **sequences**. Sequences are designed so that elements come out of them the same way they were put in (they are ordered).

Sets and dictionaries are not sequences per se, since the order of their elements is not as important as their names. Sets and dicts are called **collections**.

Post Python 3.7, dicts preserve order, and sets are sorted; but it don't matter too much.

3.2.2.1 Sets

Iterating using `for`:

```
princesses = {'belle', 'cinderella', 'rapunzel'}
for princess in princesses:
    print(princess)
```

```
## cinderella
## rapunzel
## belle
```

Now with `iter()` and `next()`:

```
princesses_i = iter(princesses)
print(next(princesses_i))
```

```
## cinderella
```

```
print(next(princesses_i))
```

```
## rapunzel
```

```
print(next(princesses_i))
```

```
## belle
```

```
type(princesses_i)
```

```
## <class 'set_iterator'>
```

3.2.2.2 Strings

using `for`

```
str1 = 'data'
for my_char in str1:
    print(my_char)
```

```
## d
## a
## t
## a
```

3.2.2.3 Tuples

`for`

```
metrics = ('auc','recall','precision','support')
for met in metrics:
    print(met)
```

```
## auc
## recall
## precision
## support
```

3.2.2.4 Dicts

```
courses = {'fall': ['regression','python'],
         'spring': ['capstone','pyspark','nlp']}
```

```
for k in courses:
  print(k)
```

```
## fall
## spring
```

```
for k in courses.keys():
  print(k)
```

```
## fall
## spring
```

```
for v in courses.values():
  print(v)
```

```
## ['regression', 'python']
## ['capstone', 'pyspark', 'nlp']
```

```
for k, v in courses.items():
  print(f"{k.upper()}:\t{', '.join(v)}")
```

```
## FALL:    regression, python
## SPRING:  capstone, pyspark, nlp
```

```
for k in courses.keys():
  print(f"{k.upper()}:\t{', '.join(courses[k])}") #index into dict
```

```
## FALL:    regression, python
## SPRING:  capstone, pyspark, nlp
```

3.2.2.5 Ranges

using `for`

```
for i in range(5):
  print(str(i+1).zfill(2),(i+1)**2*'|')
```

```
## 01 |
## 02 ||||
## 03 |||||||||
## 04 ||||||||||||||||
## 05 |||||||||||||||||||||||||
```

3.2.3 Get iteration number with `enumerate()`

Very often you will want to know iteration number you are on in a loop.

can be used to name files or dict keys, for example.

`enumerate()` will return the index and key for each iteration.

```
courses
```

```
## {'fall': ['regression', 'python'], 'spring': ['capstone', 'pyspark', 'nlp']}
```

```
for i, semester in enumerate(courses):
    course_name = f"{str(i).zfill(2)}_{semester}:\t{'-'.join(courses[semester])}"
    print(course_name)
```

```
## 00_fall: regression-python
## 01_spring:   capstone-pyspark-nlp
```

3.2.4 Nested Loops

Iterations can be nested, which is very powerful

This works well with nested data structures, like dicts within dicts.

This is basically how `JSON` files are handles, by the way…

Be careful, though, these can get deep and complicated.

```
for i, semester in enumerate(courses):
    print(f"{i+1}. {semester.upper()}:")
    for j, course in enumerate(courses[semester]):
        print(f"\t{i+1}.{j+1}. {course}")
```

```
## 1. FALL:
##  1.1. regression
##  1.2. python
## 2. SPRING:
##  2.1. capstone
##  2.2. pyspark
##  2.3. nlp
```

Use nested loops to get the cartesian product.

```
die = range(1,7)
die_rolls = []
for face1 in die:
    for face2 in die:
        die_rolls.append((face1, face2))
print(die_rolls)
```

```
## [(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (2, 1), (2, 2), (2, 3), (2, 4), (2
```

Now get frequency of die roll sums.

```
die_roll_sums = {}
```

```
for my_die_roll in die_rolls:
    my_die_roll_sum = sum(my_die_roll)
    die_roll_sums[my_die_roll_sum] = die_roll_sums.get(my_die_roll_sum, 0) + 1
```

```
for k, v in die_roll_sums.items():
    print(str(k).zfill(2), v, '|' * v)
```

```
## 02 1 |
## 03 2 ||
## 04 3 |||
## 05 4 ||||
## 06 5 |||||
## 07 6 ||||||
## 08 5 |||||
## 09 4 ||||
## 10 3 |||
## 11 2 ||
## 12 1 |
```

3.3 Comprehensions

3.3.1 List Comprehensions

Consider the following task.

Check if each integer in a list is odd and save the results (true or false) in a list.

With a standard loop, you could do this

```
vals = [1,5,6,8,12,15]
is_odd = []
for val in vals:
    if val % 2:
        is_odd.append(True)
    else:
        is_odd.append(False)
```

```
is_odd
```

```
## [True, True, False, False, False, True]
```

Now do the same with a list comprehension:

```
is_odd_comp = [val % 2 == 1 for val in vals]
is_odd_comp
```

```
## [True, True, False, False, False, True]
```

Much shorter and if you understand the syntax, quicker to interpret.

Here's how you might save all the odd numbers in a list:

```
odd_vals = [val for val in vals if val % 2 == 1]
odd_vals
```

```
## [1, 5, 15]
```

3.3.2 Comprehensions in General

Comprehensions provide a concise method for iterating over any iterable object to a new iterable object.

There are comprehensions for each type of iterable:

- List comprehensions
- Dict comprehensions
- Set comprehensions

Note: there is no tuple comprehension.

Comprehensions are essentially concise `for` loops that address the use case of *transforming one iterable into another*.

They are also more efficient than loops.

All comprehensions have the form:

listlike_result = `[expression + context + condition]`

For example, in the comprehension above we can see these parts by breaking up the code into three lines:

This is syntactically legit.

The type of comprehension is indicated by the use of enclosing pairs, just like anonymous constructors:

- List comprehensions: `[expression + context + condition]`
- Dict comprehensions: `{expression + context + condition}`
- Set comprehensions: `{expression + context + condition}`

Parts:

- **Expression** defines what to do with each element in the list.
  - this can be a complex expression, or it may not include the iterated value at all
  - for dicts, the expression is actually complex, it must be a key/value pair
- **Context** defines which iterable elements to select
- **Condition** defiens a boolean condition on the iterated value that determines if it gets included in the expression

Note that you can include comprehensions within comprehensions

And you can include multiple context + condition statements

3.3.3 Examples

3.3.3.1 Removing Stopwords

Define a sentence and a list of stop words.

Filter out stop words (considered not important).

```
sentence = "I am not a fan of this film"
stop_words = ['a','am','an','i','the','of']
```

```
clean_words = [word for word in sentence.split() if word.lower() not in stop_words]
clean_words
```

```
## ['not', 'fan', 'this', 'film']
```

Side note: this task can be accomplished with sets, if not concerned with multiple instances of the same word.

```
s1 = set(stop_words)
s2 = set(sentence.lower().split())
s2-s1
```

```
## {'film', 'not', 'this', 'fan'}
```

3.3.3.2 Selecting Tokens Containing Units

Given list of measurements, retain elements containing *mmHg* (millimeters of mercury)

```
units = 'mmHg'
measures = ['20', '115mmHg', '5mg', '10 mg', '7.5dl', '120 mmHg']
measures_mmhg = [measure for measure in measures if units in measure]

measures_mmhg
```

```
## ['115mmHg', '120 mmHg']
```

Filtering on Two Conditions

```
units1 = 'mmHg'
units2 = 'dl'
meas_mmhg_dl = [meas for meas in measures if units1 in meas or units2 in meas]

meas_mmhg_dl
```

```
## ['115mmHg', '7.5dl', '120 mmHg']
```

This can be written differently for clarity:

```
[meas
 for meas in measures
 if units1 in meas
 or units2 in meas]
```

```
## ['115mmHg', '7.5dl', '120 mmHg']
```

3.3.3.3 Dict Comprehensions

Dict comprehensions provide a concise method for iterating over a dict to create a new dict

This is common when data is structured as key-value pairs, and we'd like to filter the dict.

Here we define various deep learning models and their depths (in layers).

```
model_arch = {'cnn_1':15,'cnn_2':20,'rnn':10}
```

we use a comprehension to create a new dict containing only key-value pairs where the key contains the string `cnn`.

```
cnns = {key:model_arch[key] for key in model_arch.keys() if 'cnn' in key}

cnns
```

```
## {'cnn_1': 15, 'cnn_2': 20}
```

We build the key-value pairs using `key:model_arch[key]`, where key indexes into the dict `model_arch`.

3.4 Nested Comprehensions

Recall that nested loops are the algorithmic complement to nested data structures.

Just as we can nest loops using `for` loops, we can do so with comprehensions

Here are some examples:

3.4.1 Example 1: Creating a Matrix

Here is how we can make a matrix - a two dimensional data structure where each element is of the same data type - using plain old `for` loops.

```
matrix1 = [] # Matrix created
for i in range(5):
    matrix1.append([]) # Row created
    for j in range(5):
        matrix1[i].append(j) # Cell populated

matrix1
```

```
## [[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]
```

Here's how we can do it with a nested list comprehension:

```
matrix2 = [[j for j in range(5)] for i in range (5)]

matrix2
```

```
## [[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]
```

Replacing five lines of code with just one.

3.4.2 Example 2: Filtering a Nested List:

Create a filter to extract some things we want from a matrix.

In this case we want to pull out all the odd numbers and save them as a list.

```
matrix3 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
odd_numbers1 = []
for row in matrix3:
    for element in row:
        if element % 2 != 0:
            odd_numbers1.append(element)

odd_numbers1
```

```
## [1, 3, 5, 7, 9]
```

Now do it with a comprehension

```
odd_numbers2 = [element for row in matrix3 for element in row if element % 2 != 0]

odd_numbers2
```

```
## [1, 3, 5, 7, 9]
```

Compare the traditional `for` loop way:

with the comprehension (inside `[]`):

```
element
    for row in matrix3
        for element in row
            if element % 2 != 0
```

See how they are the same, except that the kernel operation - appending filtered values to a result list - is hoisted to the top in the case of the comprehension.

3.4.3 Example 3: Flattening Nested Sub-Lists

Here we take a nested structure and flatten it out.

```
nested_list = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]
flat_list1 = []
for sub_list in nested_list:
    for val in sub_list:
        flat_list1.append(val)

flat_list1
```

```
## [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Comprehension:

```
flat_list2 = [val for sub_list in nested_list for val in sub_list]

flat_list2
```

```
## [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

3.4.4 Example 4: Manipulating Matrix Values

Finally, we demonstrate how to manipulate each value in a matrix of words.

In this case, we simply capitalize each string.

```
matrix4 = [["apple", "banana", "cherry"],
        ["date", "fig", "grape"],
        ["kiwi", "lemon", "mango"]]
modified_matrix1 = []
for row in matrix4:
    modified_row = []
```

```
    for fruit in row:
        modified_row.append(fruit.capitalize())
    modified_matrix1.append(modified_row)

modified_matrix1
```

```
## [['Apple', 'Banana', 'Cherry'], ['Date', 'Fig', 'Grape'], ['Kiwi', 'Lemon', 'Mango']]
```

Comprehension:

```
modified_matrix2 = [[fruit.capitalize() for fruit in row] for row in matrix4]

modified_matrix2
```

```
## [['Apple', 'Banana', 'Cherry'], ['Date', 'Fig', 'Grape'], ['Kiwi', 'Lemon', 'Mango']]
```