Cheatsheets / **Python for Data Science: Working with Data**

Handling Complex Datasets

`if`, `elif`, and `else` statements

Python has `if`, `elif`, and `else` statements to control the flow of data by running code only under certain conditions.
Conditions are tested top-to-bottom in the following syntax:

```
if condition1:
   <indented code block>
elif condition2:
   <indented code block>
else:
   <indented code block>
```

As soon as one condition is `True`, Python runs the corresponding indented code block and skips any remaining statements. If no conditions are `True`, the `else` code block is run.

```
# Assign a letter grade given a test score
test_score = 87
if test_score >= 90:
    grade = "A"
elif test_score >= 80:
    grade = "B"
elif test_score >= 70:
    grade = "C"
else:
    grade = "F"
print("Letter Grade:", grade)
# Output: B
```

Python Iterables

Python **iterables** are variables that can be fed one piece at a time into iteration processes like loops. Examples of iterables include:

- Lists
- Strings (one letter at a time)
- Dictionaries
- pandas objects like Series and DataFrames

On the other hand, variables like Booleans aren't iterables: a Boolean is `True` or `False`, and has no smaller component pieces.

codecademy

Python `for` Loops

Python `for` loops run the same code on each item of a list (or other iterable) in order.
The generic syntax for a `for` loops is

```
for temporary_variable in iter
    <indented code block>
```

```
# For loop that prints numbers times 2
numbers = [1,2,3]
for num in numbers:
  print(num*2)
# Output: 2 4 6
```

Python `while` Loops

Python `while` loops are used to run a code block repetitively until a certain condition is met.
The generic syntax is

```
while condition:
    <indented code block>
```

```
# print even numbers from 0 to 10
number = 0
while number < 12:
    print(number)
    number += 2
# Output: 0 2 4 6 8 10
```

As long as the `condition` is `True` , the `while` loop will keep running the indented code block. If the condition will never become `False` , a `while` loop will repeat infinitely (or until the computer crashes!)

codecademy

Break and Continue

The `break` and `continue` keywords are used to control the execution of code in loops:

- `break` ends the loop early
- `continue` skips the current iteration of the loop and moves on to the next item

For example, in the code snippet we can search for `3` in `numbers` by using

- `continue` to skip iterations where `num != 3`
- `break` to end the loop if `num == 3`

```
# Using continue and break in a for loop
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num != 3:
        continue
    elif num == 3:
        print("Found number 3!")
        break
# Output: Found number 3!
```

range()

The Python `range()` function generates sequences of numbers like `1,2,3,4,5`. The syntax is `range(start,stop,step)` where

- `start` is the first value in the sequence
- `stop` is one number *beyond* the last number
- `step` is the number of values to skip at regular intervals or *step size*

The `start` and `step` are optional. The syntax `range(stop)` generates the sequence of all numbers starting at `0` and stopping before `stop`. The function `list()` converts a range to a standard Python list.

```
# Generate the sequence of numbers 0,1,2
range(3)

# Generate the sequence of numbers
1,2,3,4,5
range(1,6)

# Generate the even numbers 0,2,4,6
range(0,8,2)
```

List Comprehension

A **list comprehension** generates a new list from an iterable, using `for` loop syntax.
The general syntax for a list comprehension is

```
[<calculation with temporary_v
```

```
# Convert a list of penny costs to
dollars
pennies = [1000, 250, 55, 175, 804]
dollars = [amount/100 for amount in
pennies]
print(dollars)
# Output: [10.0, 2.5, 0.55, 1.75, 8.04]
```

codecademy

Vertical Concatenation

Multiple pandas DataFrames can be combined or **stacked** along the vertical index axis through a process called **vertical concatenation** using the syntax

```
pd.concat(list_of_dataframes)
```

If the DataFrames have different columns, pandas will insert NaN values to compensate.
Here are the two input DataFrames from the code snippet, and the final output of the concatention:

X

A	B
100	A
200	B
300	A

Y

A	B	C
400	A	True
500	B	False

df

A	B	C
100	A	NaN
200	B	NaN
300	A	NaN
400	A	True
500	B	False

```
X = pd.DataFrame({'A': [100,200,300],
                  'B': ["A", "B", "A"]})

Y = pd.DataFrame({'A': [400, 500],
                  'B': ["A","B"],
                  'C': [True, False]})

df = pd.concat([X, Y])
```

Print Share