Week 7: Accessing data in Pandas: Indexing and I/O

(Lecture 6 in ED)
NEW VERSION

Breno Schmidt

UNSW

Today

- Announcements
  - The new Dropbox folder
  - Issues with `yfinance`
- Class format going forward
- Quick review of assignment statements and objects
- Pandas Indexing
  - `loc`, `iloc`, and `[]`
- Intro to Pandas I/O:
  - `read_csv`: CSV -> data frame
  - `to_csv`: data frame -> CSV

Announcements

- Issues with `yfinance`
  - Only relevant for Lec 9, 10, and 11
  - I will implement a new API (will discuss in Week 8)
- The new Dropbox shared folder:
  - You will find the link in ED
    - Under "`Resources > Codes`"
    - It includes the codes Yiping already posted
  - Let's take a look

Class format going forward

Starting in Week 7:

- Extensive use of scaffolds during class
  - Make sure you have both the scaffold and PyCharm ready
- The scaffolds are located under `webinars`
  - The "solutions" are also included
- Let's take a look at the codes for today

Relevant files for today

All relevant files for today are in Dropbox

```
<DROPBOX_SHARED_FOLDER>/
|
|__ data/
|   |__ qan_prc_2020.csv      <- Created by yf_exaple3_solution.py
|
|__ webinars/                 <- Codes discussed in class
|   |__ __init__.py
|   |
|   |__ week7/                      <- Codes we will discuss today
|   |   |__ __init__.py
|   |   |__ week7_slides_p0.py
|   |   |__ week7_slides_p1.py
|   |   |__ week7_slides_p2.py
|
|__ lectures/                 <- Companion codes (discussed in the lectures in ED)
|   |__ lec_pd_indexing.py
|   |__ lec_pd_csv.py
```

Companion codes vs codes discussed in class

- Companion codes (under `lectures`):
  - Should be completed as you progress through the lectures in ED
- Codes discussed during class (under `webinars`):
  - Will be completed during class
  - Similar (not identical) to companion codes

Completed codes ("solutions")

Completed codes will be uploaded to the corresponding "solutions" folder

```
<DROPBOX_SHARED_FOLDER>/
|
|__ webinars/
|   |__ week7/
|   |   |__ solutions/                  <- Completed codes
|   |   |   |__ week7_slides_p0.py
|   |   |   |__ week7_slides_p1.py
|   |   |   |__ week7_slides_p2.py
|
|__ lectures/
|   |__ solutions/                    <- Completed codes
|   |   |__ lec_pd_indexing.py
|   |   |__ lec_pd_csv.py
```

Review of assignment statements, objects, and variables

Assignment statements: Overview

Assignment statements have the following format

```
<target> = <expression>
```

Python will "read" these statements from left to right:

1. Evaluate `<target>`
   - If not valid, raise an exception
2. Evaluate `<expression>`
   - Result is an object
3. Assign the object produced by `<expression>` to `<target>`

Example: <target> is a name (I)

- Assignment statements have the form:

  ```
  <target> = <expression>
  ```

- The simplest case is when <target> is a (variable) name.
  - In this case, how does Python evaluate these statements?
  1. Check if name is valid
     - If not, raise exception
  2. Evaluate <expression>
     - Produces an object (instance stored in the computer memory)
  3. Assign the object produced by <expression> to the variable
  - The variable is now *bound* to that object
- Intuitively:
  - Assignment statements of this type will bind names to objects
  - Bound names evaluate to the object they are currently bound to

Example: <target> is a name (II)

Consider the following statement:

```
lst = [1, 2]
```

Python will evaluate this statement from left to right:

1. The name "lst" is a valid variable name
2. Evaluate the expression `[1, 2]`
   - Expression is a valid *list literal*
     - Instruction to create a list instance with elements 1 and 2
     - This instance is created and stored in the computer memory
   - Result is an object (i.e., this list instance)
3. Assign this object to the variable `lst`
   - The name `lst` is now *bound* to *that* list instance
   - In the current namespace, unless rebound, the name "lst" will evaluate to that instance

Example: <target> is a name (III)

Keep the following in mind:

- Variable names can be *rebound*

```
lst = [1, 2]    # -> Name "lst" bound to the object [1, 2]
lst = -99       # -> Name "lst" is rebound to the object -99
```

- This means that the same name cannot be bound to two objects
  - Trying will simply rebound the variable to the other object
- However, different names can be bound to the *same* object
  - In the example below, both “x” and “y” are bound to the same object:

```
x = -99         # -> Name "x" bound to the object -99
y = x           # -> Name "y" bound to the same object x is bound to
```

Back to assignment statements

- Assignment statements have the form:

  `<target> = <expression>`

- In the examples so far, `<target>` was a variable name
- This is not the only possibility
  - More complex expressions can serve as the `<target>`
- Let's look at an example

Example: <target> as a "reference" (I)

- Consider the following statements:

```
lst = [1, 2] # -> lst bound to this list object

# The next statement also has the form:
#   <target> = <expression>

lst[0] = -99
```

- In the second statement, <target> is `lst[0]`
  - Not a single variable name
  - This means it has to be evaluated first
- How does Python evaluate `lst[0]`?

Example: <target> as a "reference" (II)

```
<target> = <expression>
  |              |
  V              V
lst[0]     =    -99
```

Evaluating <target> (intuition):

- Python will "read" the statement `lst[0] = -99` from left to right
  - <target> consists of the name "lst" followed by `[0]`
  - The name "lst" is bound to a list instance `[1, 2]`
    - Imagine Python replaces "lst" with *that* instance `[1, 2]`
  - Now we have a list object followed by `[0]`
    - This is a reference to the first element of *that* instance `[1, 2]`

In this case:

- <target> is a reference to the first element of *that* instance `[1, 2]`

Example: <target> is a "reference" (III)

Back to the example:

```
lst = [1, 2]  # -> Name lst bound to [1, 2]
lst[0] = -99
```

How is the statement `lst[0] = -99` evaluated?

1. `<target>`: `lst[0]` is a reference to *that* list `[1, 2]`
2. `<expression>`: `-99` -> create an `int` instance with value -99
3. Assign this instance `-99` to the first element of `[1, 2]`
   - Lists are mutable so Python changes it in place
   - `lst` is bound to the same object, but the object changed to `[-99, 2]`

- So, the list object is the same but its elements are different
  - How can we tell the object hasn't changed?

Object IDs in Python

Intuition:

- Think of the computer memory as a street
- Each object is allocated an "address" on that street
  - Different objects need different addresses
- Python assigns each object it creates a unique ID
  - Similar to memory addresses
  - Objects will keep their ID as long as the program is running
  - Different objects cannot have the same ID (at the same time)
- We can get the ID of an object using the built-in function `id`

```
lst = [1, 2]  # lst is bound to a list object
print( id(lst) ) # -> Displays the ID of that object
```

Why is this important?

- Motivating example: Consider the following statements:

```
# Create some lists and assign them to variables
lst0 = [1, 2]
lst1 = [1, 2]
lst2 = lst0

# Make some changes
lst2[0] = -99
```

- At the end of the program above:
  - How many **list** objects were created?
  - What is the value of `lst2[0]`?
  - What is the value of `lst0[0]`?

Please open PyCharm

The example above is discussed in the following code:

```
<DROPBOX_SHARED_FOLDER>/
|
|__ webinars/
|   |__ week7/
|   |   |__ week7_slides_p0.py
```

Part 1: Pandas indexing

Pandas indexing: Overview (I)

- Selecting elements from data structures:
  - Dicts -> Selection by keys ("labels")

```
dic = {'a': 1, 2: 3}
dic['a'] # -> 1 (selection by key/label)
dic[2]   # -> 3 (selection by key/label)
```

  - Tuples/lists -> Selection by index (position)

```
lst = [1, 2, 3]
lst[0] # -> 1 (selection by index/position)
```

- Note that there is no ambiguity when using `[]`:
  - `dict[indexer]`: indexer is the key
  - `list[indexer]`: indexer is an integer (position)
- What about Pandas?

Pandas indexing: Overview (II)

- Intuitively, a Pandas series combine:
  - Array of values
  - Index (similar to dict)
- We would like to select elements based on:
  - Their position in the array
  - Their label in the index
- But if we use `ser[indexer]` how does Pandas know?
- In principle, using `ser[indexer]` can be very confusing!
  - Is the indexer a label or a position?
- Solution: Implement two properties, `.loc` and `.iloc`
  - `ser.loc`: Selection by labels (in the index)
  - `ser.iloc`: Selection by position (in the array)
- Today: Using `.loc` and `.iloc` to select elements from series and data frames

Part 2: Intro to Pandas I/O

Pandas I/O: Overview

- Pandas allows you to both "import" and "export" data
- Typically, `pandas.read_<format>` is used to import data stored as `<format>`
  - E.g., `pandas.read_csv: CSV -> pandas.DataFrame`
- You can use data frame methods to "export" the data to `<format>`:
  - `df.to_csv(<path>)`: Saves the content of the `df` to the CSV file `<path>`
- We will focus on `read_csv` and `to_csv` today.

Please open PyCharm

For the remainder of this lecture, please refer to the following codes:

```
<DROPBOX_SHARED_FOLDER>/
|
|__ webinars/
|   |__ week7/
|   |   |__ week7_slides_p1.py
|   |   |__ week7_slides_p2.py
```