2c.Lambdas

November 2, 2024

1 Lambdas

- anonymous/inline functions
- "throw away" functions
- good for passing a function as an argument to another function
- quick and easy
- one line
- expression
- No statements - no assignments (x=3) , some logic - loops, etc.
- 'anonymous functions' - can and should be unnamed

[ ]:
```
import pandas as pd
from functools import reduce
import numpy as np
```

2 Syntax

`lambda argument(s) : expression` expression is the return value

You shouldn't use named lambda functions. Use def instead. This is just an example so you can see lambda on it's own

[ ]:
```
# Don't do this.  It works, but is considered "bad practice", e.g. no type␣
↪hints, no doc string
multiply_2_lambda = lambda x: x*2
type(multiply_2_lambda)
```

[ ]:
```
function
```

[ ]:
```
# ... but it does work.
multiply_2_lambda(9)
```

[ ]:
```
18
```

[ ]:
```
multiply_2_lambda?
```

[ ]:
```
# Do this instead. Notice type hints and doc string
def multiply_2_def(x: int) -> int:
```

```
  '''Doubles the value of an integer'''
  return x*2
type(multiply_2_def)
```

[ ]:
```
function
```

[ ]:
```
multiply_2_def(9)
```

[ ]:
```
18
```

[ ]:
```
multiply_2_def("hello ")
```

[ ]:
```
' hello hello'
```

[ ]:
```
multiply_2_def?
```

3 A few functions in Python that take lambdas as arguments

Generally, lambdas are used in the context of another function, such as the following: - map - reduce - sort

3.1 Map

[ ]:
```
# Map allows you to transform all the items in an iterable without
# using a for loop
# Input to function is a single element at a time
# It is useful when you need to apply a transformation function to each item in
# an iterable
# ... without having to create a named function.

my_list = [2,3,6,7,4,4,9]

list_3 = list(map(lambda x: x*3, my_list))
print(list_3)
```

```
[6, 9, 18, 21, 12, 12, 27]
```

[ ]:
```
# The same using a named function
def times_3( x: int) -> int:
  '''Triples the value of an integer'''
  return x*3

my_list = [2,3,6,7,4,4,9]

# notice the name of the function is passed without "()"
list_3 = list(map(times_3, my_list))
print(list_3)
```

```
[6, 9, 18, 21, 12, 12, 27]
```

[ ]:
```
times_3("hi ")
```

[ ]:
```
'hi hi hi '
```

3.2 Reduce

[ ]:
```
# Reduce will apply a function *cumulatively* to all elements in an interable.
# Input is initial pair folllowed by cumulative value and next element
my_list = [2, 3, 7, 3]
print(reduce(lambda x, y: x * y, my_list))
```

```
126
```

[ ]:
```
# The same using a named function
def times_xy( x:int, y:int) -> int:
  '''Multiplies X and Y'''
  return x * y

my_list = [2, 3, 7, 3]
print(reduce(times_xy, my_list))
```

```
126
```

[ ]:
```
print(reduce(lambda x, y: x + y, my_list))
```

```
15
```

[ ]:
```
print(reduce(lambda x, y: x + y, range(101)))
```

```
5050
```

3.3 Sort

[ ]:
```
# Key is a parameter used to specify a function used on each list element prior
# sorting, e.g. with nested lists

words = [['chrysanthemum', 9], ['foo',8], ['blue',-7], ['loo',9], ['barbaric',␣
  ↪5], ['barber',3]]
words.sort()     # Sorts by first element (the string)
print(words)
```

```
[['barbaric', 5], ['barber', 3], ['blue', -7], ['chrysanthemum', 9], ['foo', 8],
['loo', 9]]
```

[ ]:
```
words.sort(key = lambda x: x[1]) # Sorts by second element (the number)
print(words)
```

```
[['blue', -7], ['barber', 3], ['barbaric', 5], ['foo', 8], ['chrysanthemum', 9],
['loo', 9]]
```

[ ]:
```
words.sort(key = lambda x: abs(x[1])) # Sorts by second element (the number)
print(words)
```

```
[['barber', 3], ['barbaric', 5], ['blue', -7], ['foo', 8], ['chrysanthemum', 9],
['loo', 9]]
```

[ ]:
```
# Same using a named function
def get_second_item(x):
  return x[1]

words = [['chrysanthemum', 9], ['foo',8], ['blue',-7], ['loo',9], ['barbaric',␣
 ↪5], ['barber',3]]
words.sort(key=get_second_item)
print(words)
```

```
[['blue', -7], ['barber', 3], ['barbaric', 5], ['foo', 8], ['chrysanthemum', 9],
['loo', 9]]
```

[ ]:
```
get_second_item
```

[ ]:
```
<function __main__.get_second_item(x)>
```

4 Lambdas in DataFrames

[ ]:
```
df = pd.DataFrame([[1,2,3],[4,3,6],[7,6,5]])
df
```

[ ]:

	0	1	2
0	1	2	3
1	4	3	6
2	7	6	5

[ ]:
```
# Create a new column that is a function of another column
df[3] = df[2].apply(lambda x: x*2)
df
```

[ ]:

	0	1	2	3
0	1	2	3	6
1	4	3	6	12
2	7	6	5	10

[ ]:
```
# Same using a named function
df[4] = df[2].apply(multiply_2_def)
df
```

[ ]:

	0	1	2	3	4
0	1	2	3	6	6
1	4	3	6	12	12

```
2  7  6  5  10  10
```

[ ]:
```
df_foo = pd.DataFrame()
df_foo
```

[ ]:
```
Empty DataFrame
Columns: []
Index: []
```

[ ]:
```
# Same using a named function
df_foo["func"] = df[2].apply(multiply_2_def)
df_foo
```

[ ]:
```
   func
0     6
1    12
2    10
```

[ ]:
```
# Create a new column that is a function of another column
df_foo["lambda"] = df[2].apply(lambda x: x*2)
df_foo
```

[ ]:
```
   func  lambda
0     6       6
1    12      12
2    10      10
```

4.1 Your Turn

4.1.1 Part 1

A data frame, `df`, has been defined for you below. Use lambdas to do the following:

1. Create a fourth column that is the square of the first column.
2. Create a fifth column that is the square root of the second column.

```
df = pd.DataFrame([[1,2,3],[4,3,6],[7,6,5]])
df
```

[68]:
```
df = pd.DataFrame([[1,2,3],[4,3,6],[7,6,5]])
df
```

[68]:
```
   0  1  2
0  1  2  3
1  4  3  6
2  7  6  5
```

[ ]:
```
# Solution 1
df[3] = df[0].apply( lambda x: x**2)
df
```

[ ]:
```
   0  1  2   3
0  1  2  3   1
1  4  3  6  16
2  7  6  5  49
```

[ ]:
```
# Solution 2
df[4] = df[1].apply(np.sqrt)
df
```

[ ]:
```
   0  1  2   3         4
0  1  2  3   1  1.414214
1  4  3  6  16  1.732051
2  7  6  5  49  2.449490
```

[ ]:
```
# Solution 2 - variant 2
from math import sqrt
df[1].apply(lambda x: sqrt(x))
```

[ ]:
```
0    1.414214
1    1.732051
2    2.449490
Name: 1, dtype: float64
```

[ ]:
```
# Solution 2 - variant 3
np.sqrt(df[1])
```

[ ]:
```
0    1.414214
1    1.732051
2    2.449490
Name: 1, dtype: float64
```

[ ]:
```
df[1].apply(lambda x: x**0.5)
```

[ ]:
```
0    1.414214
1    1.732051
2    2.449490
Name: 1, dtype: float64
```

[ ]:
```
df[1].map(lambda x: x**0.5)
```

[ ]:
```
0    1.414214
1    1.732051
2    2.449490
Name: 1, dtype: float64
```

[73]:
```
( df
# .apply(lambda x: x**2, axis=1)
# .apply(lambda x: x.sum(), axis=1)
```

```
)
```

[73]:
```
0     14
1     61
2    110
dtype: int64
```

4.1.2 Part 2

A data frame, `df`, has been defined for you below. Use lambdas to do the following:

1. Create a column `fname_cap` that has the first letter of `fname` capitalized.
2. Create a column `lname_cap` that has the first letter of `lname` capitalized.
3. Create a column `lfname_cap` that is `lname_cap` and `fname_cap` separated by a comma and a space.

In the end, the data frame should look like this:

	fname	lname	fname_cap	lname_cap	lfname_cap
0	GEORGE	WASHINGTON	George	Washington	Washington, George
1	JOHN	ADAMS	John	Adams	Adams, John
2	THOMAS	JEFFERSON	Thomas	Jefferson	Jefferson, Thomas
3	JAMES	MADISON	James	Madison	Madison, James

[ ]:
```
names =  ['GEORGE WASHINGTON',
          'JOHN ADAMS',
          'THOMAS JEFFERSON',
          'JAMES MADISON'
]

(fname, lname) = list(zip(*[ i.split() for i in names ]))

df = pd.DataFrame( { "fname": fname,
                     "lname": lname } )
df
```

[ ]:
```
     fname       lname
0   GEORGE  WASHINGTON
1     JOHN       ADAMS
2   THOMAS   JEFFERSON
3    JAMES     MADISON
```

[ ]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
```

```
---  ------  --------------  -----
 0   fname   4 non-null      object
 1   lname   4 non-null      object
dtypes: object(2)
memory usage: 192.0+ bytes
```

[ ]:
```
# Solution 1
# Create a column fname_cap that has the first letter of fname capitalized.

df["fname_cap"] = df["fname"].apply(lambda x: x.capitalize() )
df
```

[ ]:
```
     fname       lname fname_cap
0   GEORGE  WASHINGTON    George
1     JOHN       ADAMS      John
2   THOMAS   JEFFERSON    Thomas
3    JAMES     MADISON     James
```

[ ]:
```
# Solution 2
df["lname_cap"] = df["lname"].apply(lambda x: x.capitalize() )
df
```

[ ]:
```
     fname       lname fname_cap   lname_cap
0   GEORGE  WASHINGTON    George  Washington
1     JOHN       ADAMS      John       Adams
2   THOMAS   JEFFERSON    Thomas   Jefferson
3    JAMES     MADISON     James     Madison
```

In pandas, without lambda:

```
df["lfname_cap"] = df["lname_cap"] + ", " + df["fname_cap"]
```

In SQL:

```
select lname_cap || ", " || fname_cap as "lfname"
```

[ ]:
```
# Solution 3
df[["lname_cap", "fname_cap"]].apply(lambda x:  x["lname_cap"] + ", " +␣
 ↪x["fname_cap"] , axis = 1 )
```

[ ]:
```
0    Washington, George
1           Adams, John
2     Jefferson, Thomas
3        Madison, James
dtype: object
```

[ ]:
```
# Solution 3
df["lfname_cap"] = df[["lname_cap", "fname_cap"]].apply(lambda x: f"{x.
 ↪iloc[0]}, {x.iloc[1]}", axis = 1 )
df
```

[ ]:

	fname	lname	fname_cap	lname_cap	lfname_cap
0	GEORGE	WASHINGTON	George	Washington	Washington, George
1	JOHN	ADAMS	John	Adams	Adams, John
2	THOMAS	JEFFERSON	Thomas	Jefferson	Jefferson, Thomas
3	JAMES	MADISON	James	Madison	Madison, James

[ ]:
```
df[["lname_cap", "fname_cap"]].iloc[0]
```

[ ]:
```
lname_cap    Washington
fname_cap        George
Name: 0, dtype: object
```

[ ]:
```
# Solution 3
df[["lname_cap", "fname_cap"]].apply(lambda x: f"{x['lname_cap']},␣
 ↪{x['fname_cap']}", axis = 1 )
```

[ ]:
```
0    Washington, George
1           Adams, John
2     Jefferson, Thomas
3        Madison, James
dtype: object
```

[ ]:
```
# Solution 3
df[["lname_cap", "fname_cap"]].apply(lambda x: x.index, axis = 1 )
```

[ ]:
```
0    Index(['lname_cap', 'fname_cap'], dtype='object')
1    Index(['lname_cap', 'fname_cap'], dtype='object')
2    Index(['lname_cap', 'fname_cap'], dtype='object')
3    Index(['lname_cap', 'fname_cap'], dtype='object')
dtype: object
```

[ ]:
```
# Solution 3
df[["lname_cap", "fname_cap"]].apply(lambda x: print(x), axis = 0 )
```

```
0    Washington
1         Adams
2     Jefferson
3       Madison
Name: lname_cap, dtype: object
0    George
1      John
2    Thomas
3     James
Name: fname_cap, dtype: object
```

[ ]:
```
lname_cap    None
fname_cap    None
dtype: object
```