python-data-science-perspective-1

June 16, 2024

[16]:
```
#1. Docsring: Tells about function
print(print.__doc__)
```

```
Prints the values to a stream, or to sys.stdout by default.

  sep
    string inserted between values, default a space.
  end
    string appended after the last value, default a newline.
  file
    a file-like object (stream); defaults to the current sys.stdout.
  flush
    whether to forcibly flush the stream.
```

[17]:
```
#We can add docstring to our function:
def f(x,y):
    '''Returns sum of x and y
    Arg(x) must be int or float
    Arg(y) must be int or float'''
    return x+y
print(f.__doc__)
```

```
Returns sum of x and y
    Arg(x) must be int or float
    Arg(y) must be int or float
```

[ ]:
```
# A function should Do one thing only i.e. we should not create a function that␣
 ↪do multiple task at once, instead we should nest or create seperate␣
 ↪functions for each task to reduce error probability and easy debugging.
```

[2]:
```
#2. Context Manager: In Python, a context manager is a way to allocate and␣
 ↪release resources precisely when you want to. The most common use of context␣
 ↪managers is with the with statement, which ensures that resources are␣
 ↪properly cleaned up when the code block inside the with statement is exited.
with open('example.txt', 'r') as file: #as file assigned the variable file to␣
 ↪opened data
    content = file.read()
# The file is automatically closed after the block.
```

[4]:

```
#open is a built-in context manager, we can create our own context manager as␣
 ↪below:
import contextlib
@contextlib.contextmanager
def my_context(name):
    print('Hey!') #Here we can add any start-up code
    yield name #This is must in context manager. When you write yeild word, it␣
 ↪means that you are going to return a value, but you expect to finish the␣
 ↪rest of the function at some point in the future
    print("See you Soon!") #Here we can add any ending code
with my_context('Moin') as foo:
    print(f'You {foo}')
#If your code follows patterns such as␣
 ↪start-stop,start-end,ope-close,connect-disconnect etc. you should use␣
 ↪context manager.
# We can use nested context managers and try except finally for error handling␣
 ↪in context managers.
```

```
Hey!
You Moin
See you Soon!
```

[4]:

```
#3. Function as Object: python treats functions as object, like we have other␣
 ↪objects in python. Hence, we can store funct. as variables, list and much␣
 ↪more.
def f(name):
    return 'Hello '+name
#Store func. as variable (see difference in storage types)
x=f('Moin')
y=f
print(x)
print(y)
print(y('Moin'))
```

```
Hello Moin
<function f at 0x00000225150D7A60>
Hello Moin
```

[9]:

```
#Store func. in  list:
l=[f,print,len]
print(l)
print(l[0]('Moin'))
print(l[2]('Moin'))
#Similarly we can store funcs. in dict.
```

```
[<function f at 0x00000225150D7A60>, <built-in function print>, <built-in
function len>]
```

```
Hello Moin
4
```

[1]:

```
#4. Functions with Multiple Parameters:
def shout_all(word1,word2):
    shout1=str(word1)+'!!!'
    shout2=str(word2)+'!!!'
    shout_words=(shout1,shout2)
    return shout_words
yell1,yell2=shout_all('congratulations','you')
print(yell1)
print(yell2)
#UYNDERSTAND IT
```

```
congratulations!!!
you!!!
```

[2]:

```
import pathlib
import pandas as pd
path=pathlib.Path.cwd()/'Ames_Housing_Data1.tsv'
df=pd.read_csv(path,delimiter='\t')
#Defining Function on Pandas DF:
def count_entries(df,col_name):
    """Return a dictionary with counts of
    occurrences as value for each key."""
    langs_count = {}
    col = df[col_name]
    for entry in col:
        if entry in langs_count.keys():
            langs_count[entry]=langs_count[entry]+1
        else:
            langs_count[entry]=1
    return langs_count
result=count_entries(df,'MS Zoning')
print(result)
```

```
{'RL': 2274, 'RH': 27, 'FV': 139, 'RM': 462, 'C (all)': 25, 'I (all)': 2, 'A
(agr)': 2}
```

[3]:

```
#5. Packing/Unpacking Tuple:
a,b,c=(1,3,5)
a
```

[3]:

```
1
```

[4]:

```
T=(1,5,5)
a,b,c=T
b
```

[4]: 5

[5]:
```
#6. Nested Functions
def echo(n):
    """Return the inner_echo function."""
    def inner_echo(word1):
        """Concatenate n copies of word1."""
        echo_word = word1 * n
        return echo_word
    return inner_echo
# Call twice() and thrice() then print
print(echo(2)('hello'), echo(3)('hello'))
```

```
hellohello hellohellohello
```

[ ]:
```
#7. Global/nonlocal xglobal x changes value of x in whole scope while nonlocal␣
 ↪x changes value of x in nested scope and is used in nested func.
y = 0
def h(x):
    y = 11
    z = x + y
    def g(z):
        nonlocal y
        y = 10
        l = x + y + z
        return l
    result = g(z)
    print(y)  # This will print 10 because `g` modifies `y` using `nonlocal`
    return result

print(h(1))  # Prints 22, because x=1, y=10 (modified by g), and z=12 (from x +␣
 ↪y before modification)
print(y)     # Prints 0, because global `y` was never changed
```

[7]:
```
#8. Default Argumentation:
def f(x,y=2):
    return x**y
print(f(3),f(3,3)) #if you specify y it wll use it else it will take default␣
 ↪value of y
```

```
9 27
```

[ ]:
```
#9. Variable Length Args. Funcs.
```

[8]:
```

```

```
#i. *args: In Python, *args is used in function definitions to allow the␣
 ↪function to accept a variable number of arguments. *args is used to pass a␣
 ↪non-keyworded, variable-length argument list. The arguments passed to the␣
 ↪function are stored in a tuple.
def greet(greeting, *args):
    for name in args:
        print(f"{greeting}, {name}!")
greet("Hello", "Alice", "Bob", "Charlie")
```

```
Hello, Alice!
Hello, Bob!
Hello, Charlie!
```

[9]:

```
#ii. **kwargs:In Python, **kwargs is used in function definitions to allow the␣
 ↪function to accept an arbitrary number of keyword arguments. kwargs stands␣
 ↪for keyword arguments and is stored in a dictionary. This allows you to pass␣
 ↪a variable number of arguments to a function, where the arguments are␣
 ↪specified by keywords.
def greet(greeting, **kwargs):
    print(greeting)
    for key, value in kwargs.items():
        print(f"{key}: {value}")
greet("Hello", name="Alice", age=30, city="New York")
```

```
Hello
name: Alice
age: 30
city: New York
```

[10]:

```
#10. Lambda Function:
echo_word = lambda word1,echo:word1*echo
result = echo_word('hey',5)
print(result)
```

```
heyheyheyheyhey
```

[ ]:

```
#11. Map/Filter
```

[11]:

```
#Map (alternate to For loop) & Lambda Function: The map function in Python␣
 ↪applies a given function to all the items in an input list (or any other␣
 ↪iterable) and returns a map object (which is an iterator). This is a␣
 ↪convenient way to apply a transformation to each element in a collection␣
 ↪without using a loop.
names = ["protego", "accio", "expecto patronum", "legilimens"]
# Use map() to apply a lambda function over names
shout_names = map(lambda item:item+'!!!' ,names)
print(shout_names)
print(list(shout_names))
```

```
<map object at 0x000002126E98CC70>
['protego!!!', 'accio!!!', 'expecto patronum!!!', 'legilimens!!!']
```

[12]:
```
#Filter (alternate to if) & Lambda Function: The filter function in Python is␣
 ↪used to construct an iterator from elements of an iterable for which a␣
 ↪function returns true. This function is useful for filtering elements out of␣
 ↪a sequence based on a condition.
fellowship = ['frodo', 'samwise', 'merry', 'pippin', 'aragorn', 'boromir',␣
 ↪'legolas', 'gimli', 'gandalf']
result = filter(lambda member:len(member)>6 , fellowship)
print(result)
print(list(result))
```

```
<filter object at 0x000002126E98CDF0>
['samwise', 'aragorn', 'boromir', 'legolas', 'gandalf']
```

[13]:
```
#12. Error Handling:
```

[14]:
```
def sqrt(x):
    try:
        return x**0.5
    except:
        print('x must be an integer or float')
print(sqrt(5))
print(sqrt('5'))
```

```
2.23606797749979
x must be an integer or float
None
```

[32]:
```
def f(x,y):
    try:
        return x/y
    except ZeroDivisionError: #We only excepted ZeroDivisionError
        print("Error: Cannot divide by zero.")
print(f(10,100))
print(f(10,0))
print(f(10/'0'))
```

```
0.1
Error: Cannot divide by zero.
None
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[32], line 8
```

```
      6 print(f(10,100))
      7 print(f(10,0))
----> 8 print(f(10/'0'))

TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

[33]:

```
#We can also define raising error for a specific condition:
def shout_echo(word1, echo=1):
    """Concatenate echo copies of word1 and three
    exclamation marks at the end of the string."""
    # Raise an error with raise
    if echo<0:
        raise ValueError('echo must be greater than or equal to 0')
    # Concatenate echo copies of word1 using *: echo_word
    echo_word = word1 * echo
    # Concatenate '!!!' to echo_word: shout_word
    shout_word = echo_word + '!!!'
    # Return shout_word
    return shout_word
# Call shout_echo
print(shout_echo("particle", echo=5))
shout_echo("particle", echo=-5)
```

```
particleparticleparticleparticleparticle!!!
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[33], line 16
     14 # Call shout_echo
     15 print(shout_echo("particle", echo=5))
---> 16 shout_echo("particle", echo=-5)

Cell In[33], line 7, in shout_echo(word1, echo)
      5 # Raise an error with raise
      6 if echo<0:
----> 7     raise ValueError('echo must be greater than or equal to 0')
      8 # Concatenate echo copies of word1 using *: echo_word
      9 echo_word = word1 * echo

ValueError: echo must be greater than or equal to 0
```

[ ]:

```
#13. Iteration:
```

[ ]:

```
#An iterable is an object that can return an iterator, while an iterator is an␣
↪object that keeps state and produces the next value when you call next() on␣
↪it
```

```python
flash = ['jay garrick', 'barry allen', 'wally west', 'bart allen']
superhero=iter(flash)
# Print each item from the iterator
print(superhero)
print(next(superhero))
print(next(superhero))
print(next(superhero))
print(next(superhero))
#Here flash is iterable and superhero is iterator.
```

[ ]:

```python
#or we can use:
for i in flash:
    print(i)
```

[ ]:

```python
#14. Enumerate & Zip: Enumerate generates an iterator of iterable that contains tuple based on natural indexing, while zip generates an iterator of iterable that contains tuples based on artificial (1st arg. of Zip) indexing
l1 = ['Moin', 'Rafi', 'Sid', 'Aneeq']
e=enumerate(l1,start=1) #start specifies start of indexing, default is 0
print(e)
for i in e:
    print(i)
```

[ ]:

```python
l1 = ['Moin', 'Rafi', 'Sid', 'Aneeq']
l2 = [24, 29, 26, 20]
z = zip(l1, l2)
print(z)
for i in z:
    print(i)
```

[ ]:

```python
l1 = ['Moin', 'Rafi', 'Sid', 'Aneeq']
l2 = [24, 29, 26, 20]
d = dict(zip(l1, l2))
l=list(zip(l1,l2))
t=tuple(zip(l1,l2))
print(d)
print(l)
print(t)
```

[ ]:

```python
l1 = ['Moin', 'Rafi', 'Sid', 'Aneeq']
l2 = [24, 29, 26, 20]
l3=['A','B','C','D']
z=zip(l1,l2,l3)
for i,j,k in z:
    print(i)
    print(j)
    print(k)
```

[ ]:

```
#Ziping & Unziping with *:
z1 = zip(l1,l2)
print(*z1) # Print the tuples in z1 by unpacking with *
z1 = zip(l1,l2) #Recreate z1
result1, result2 = zip(*z1) # 'Unzip' the tuples in z1 by unpacking with * and␣
 ↪zip(): result1, result2
print(result1)
print(result2)
```

[ ]:

```
#15. Iterating over Chunks of data in Pandas DF: In Pandas, a chunk is a subset␣
 ↪of a dataframe that is processed in smaller pieces, rather than loading the␣
 ↪entire dataframe into memory at once. This is useful when working with large␣
 ↪datasets that exceed available memory. Useful for Dealing BIG DATA
import pathlib
import pandas as pd
path=pathlib.Path.cwd()/'Ames_Housing_Data1.tsv'
ck=pd.read_csv(path,delimiter='\t',chunksize=100)
print(ck) #it is an iterable object
```

[ ]:

```
CkSP=[]
for chunk in ck:
    CkSP.append(sum(chunk['SalePrice']))
print(CkSP)
print(sum(CkSP))
#The above loop took sum of first 100 values, then next 100 and so on and then␣
 ↪sum of all give us total  sum at end
```

[ ]:

```
df=pd.read_csv(path,delimiter='\t')
df['SalePrice'].sum()
```

[ ]:

```
#We can automate many work on pandas df by using functions
# Define count_entries()
def count_entries(csv_file,  delim,c_size, colname):
    """Return a dictionary with counts of
    occurrences as value for each key."""
    # Initialize an empty dictionary: counts_dict
    counts_dict = {}
    # Iterate over the file chunk by chunk
    for chunk in pd.read_csv(csv_file,delimiter=delim,chunksize=c_size):
        # Iterate over the column in DataFrame
        for entry in chunk[colname]:
            if entry in counts_dict.keys():
                counts_dict[entry] += 1
            else:
                counts_dict[entry] = 1
    # Return counts_dict
    return counts_dict
```

```

# Call count_entries(): result_counts
result_counts = count_entries(path,'\t', 100, 'Sale Condition')

# Print result_counts
print(result_counts)
print(df['Sale Condition'].value_counts())
```

[ ]:

```
#16. List Comprehension: List comprehension is a concise way to create lists in␣
 ↪Python. It allows for the generation of lists by specifying an expression␣
 ↪followed by a for loop, and optionally, an if statement to filter items.
 ↪Basic Syntax: [expression for item in iterable if condition]
```

[ ]:

```
# Create a list of even numbers from 0 to 9
evens = [x for x in range(10) if x % 2 == 0]
print(evens)
```

[ ]:

```
# Create a list of tuples (x, y) where x is from 0 to 2 and y is from 0 to 2
tuples = [(x, y) for x in range(3) for y in range(3)]
print(tuples)
```

[ ]:

```
# Define a function to double a number
def double(x):
    return x * 2
doubled = [double(x) for x in range(5)]
print(doubled)
```

[ ]:

```
#If else
[num ** 2 if num % 2 == 0 else 0 for num in range(10)]
```

[ ]:

```
#17. Dictionary comprehension: {key_expression: value_expression for item in␣
 ↪iterable if condition}
```

[ ]:

```
# Create a dictionary of even numbers and their squares from 0 to 4
even_squares = {x: x**2 for x in range(5) if x % 2 == 0}
print(even_squares)
```

[ ]:

```
# Create a list of tuples
pairs = [('a', 1), ('b', 2), ('c', 3)]
dictionary = {key: value for key, value in pairs}
print(dictionary)
```

[ ]:

```
# Create an initial dictionary
original_dict = {'a': 1, 'b': 2, 'c': 3}
inverted_dict = {value: key for key, value in original_dict.items()}
print(inverted_dict)
```

[ ]:

```
#Practical Example:
words = ["hello", "world", "python", "dictionary", "comprehension"]
lengths_dict = {word: len(word) for word in words}
print(lengths_dict)
```

[ ]:

```
#18. Generator: In Python, generators provide a convenient way to implement␣
 ↪iterators. They allow you to iterate over a potentially large dataset␣
 ↪without loading the entire dataset into memory. This is achieved by using␣
 ↪the yield keyword, which allows the generator to produce a series of values␣
 ↪over time instead of computing and returning them all at once. Each call to␣
 ↪yield produces a value and pauses the function's execution, maintaining its␣
 ↪state for the next call. This allows you to iterate over large datasets or␣
 ↪infinite sequences without using excessive memory.
# l=[i for i in range(0,10**100)] if you run it, python will cash
g=(i for i in range(0,10**100))
print(g) #this is generator object which we can work on as required
for i in g:
    if i<=100:
        print(i)
```

[ ]:

```
def infinite_sequence(x):
    num1 = 0
    num2=0
    while True:
        yield num1**x,num2**x
        num1 += 1
        num2+=-1
# Create an infinite generator
gen = infinite_sequence(2)
# Print the first 10 numbers in the infinite sequence
for i in range(10):
    print(next(gen))
```

[ ]:

```
19. Closure/Decorators: Advance Python Topics but are crucial in Advanced Data␣
 ↪Science (Learn them afterward)
```