Python Data Science Toolbox

by datacamp

Built-in functions

str()

example

```
x = str(5)
print(x)
```

output 5

Defining a function

```
def square():      #function header
        new_value = 4 ** 2      #function body
        print(new_value)
```

Function parameters

** add parameter within the parentheses

```
def square(value):
        new_value = value ** 2
        print(new_value)
square(4)
```

output 16

```
square(5)
```

output(25)

** can now put in any value

** when you define a function, you write parameters in the function header

** when you call a function, you pass arguments into the function

To return and assign variable instead of printing it you can use the keyword return

```
def square(value):
        new_value = value ** 2
        return new_value
```

**we can assign the variable;

for this example we use num

```
num = square(4)
print(num)
```

***returning values is generally more desirable than printing because print() call assigned to a variable has data type NoneType

Docstrings

describe what your function does

serve as documentation for your function

placed in the immediate line after the function header
**must be placed between triple quotes “””

Multiple function parameters

example

```
def raise_to_power(value1, value2):
	“””Raise the value1 to the power of value2”””
	new_value = value1 ** value2
	return new_value
```

**to call function > #of arguments needs to be = to #of parameters

example with above function

```
result = raise_to_power(2,3)
print(result)
output 8
```

Can make function return multiple values in the form of tuples

Tuples are:

- like a list in that they can contain multiple values
- unlike lists, they are immutable (ie unchangeable, cannot be modified) once they have been constructed
- are constructed using parentheses () instead of brackets like lists

Can unpack tuples

What this means:

example

```
even_nums = (2,4,6) #our tuple
#now assign/unpack tuple to its own variables
a,b,c = even_nums
print(a) > output 2
print(b) > output 4
print(c) > output 6
```

Can access tuples like you access lists

Indexing

example

```
even_nuns = (2, 4, 6)
print(even_nums[1] > output 4
```

Returning multiple values

example

```
def raise_both(value1, value2):
	“””Raise value1 to the power of value2 and vice versa”””
```

```
        new_value1 = value1 ** value2
        new_value2 = value2 ** value1
        new_tuple = (new_value1, new_value2)
        return new_tuple
result = raise_both(2, 3)
print(result) > (8, 9)
```

Example

```
# Import pandas
import pandas as pd

# Import Twitter data as DataFrame: df
df = pd.read_csv('tweets.csv')

# Initialize an empty dictionary: langs_count
langs_count = {}

# Extract column from DataFrame: col
col = df['lang']

# Iterate over lang column in DataFrame
for entry in col:

    # If the language is in langs_count, add 1
    if entry in langs_count.keys():
        langs_count[entry] += 1
    # Else add the language to langs_count, set the value to 1
    else:
        langs_count[entry] = 1

# Print the populated dictionary
print(langs_count)
```

Crash course on scope in functions

- not all objects are accessible everywhere in a script
- scope is part of the program where an object or name may be accessible
- Names refer to the variables or more generally objects such as functions that are defined in your program
- example variable x has a name as does the function sum
- Three types of scope:
    - 1. Global scope - defined in the main body of a script
    - 2. Local scope - defined inside a function
    - **once the execution of a function is done, any name inside the local

    - scope ceases to exist, which means that you cannot access those names anymore outside of the function definition
    - 3. Built-in scope - names in the pre-defined built-in module
- **when we reference a name, first the local scope is searched, then the global, if the name is in neither, then the built-in scope is searched

To alter the value of a global name within a function call
example

```
new_val = 10
def square(value):
        global new_val
        new_val = new_val ** 2
        return new_val
```

we can do this with the keyword global
we use it within the body of the function to state that we wish to alter and the variable to alter

Nested functions
example structure

```
def outer(..):
        x = …
        def inner(…):
                y = x ** 2
        return …
```

Python searches the local scope of the function inner, if it doesn’t find x, it searches the scope of the function outer
**the outer function is also called an enclosing function
if Python can’t find x in the function outer then it searches global and then looks into built-in modules
why nested functions?:
example we want a function that takes 3 parameters and performs the same function on each of them
example

```
def mod2plus5(x1, x2, x3):
        def inner(x):
                return x % + 5
        return (inner(x1), inner(x2), inner(x3))
print(mod2plus5(1, 2, 3))
```

Another example
Returning functions

```
def raise_val(n):
        “””Return the inner function.”””
```

```
    def inner(x):
    “””Raise x to the power of n.”””
        raised = x ** n
        return raised
    return inner
square = raise_val(2)
cube = raise_val(3)
print(square(2), cube(4))
```

like the keyword global that we used to change the global scope
we can use the keyword nonlocal to make changes in the enclosing scope
example

```
def outer():
    ‘’’Prints the value of n.’’’
    n = 1
    def inner():
        nonlocal n
        n = 2
        print(n)
    inner()
    print(n)
outer() > output 2 2
```

**’Closure’ means that the nested or inner function remembers the state of its enclosing scope when called.
**meaning that what is defined locally in the enclosing scope is available to the inner function even when the outer function has finished execution.

Add a default argument

```
#pow is second argument; we put it to =1 which means if we don’t call a 2nd
argument it will automatically call 1
def power(number, pow=1):
    “””Raise number to the power of pow.”””
    new_value = number ** pow
    return new_value
power(9,2)
#in this case we put in a 2nd argument 2 so it will run with 2
#if we put power(9), the 2nd argument will run as the default 1
```

Flexible arguments: *args(1)
for when you want to write a function but aren’t sure how many arguments a user will want to pass
example a function that takes floats or ints and adds them all up, irrespective of

how many there are
below example is of a function that sums up all the arguments passed to it
*args turns all the arguments passed to a function call into a tuple called args in the function body
then use our desired function in the function body and loop over the tuple args and add each element of it successively

```
def add_all(*args):
    “””Sum all values in *args together.”””
    #Initialize sum
    sum_all = 0
    #accumulate the sum
    for num in args:
        sum_all += num
    return sum_all
add_all(5, 10, 15)
```

output 30
**can put any input into add_all

Flexible arguments: **kwargs
** allows you to pass an arbitrary number of keyword arguments
ie arguments preceded by identifiers
turns the identifier-keyword pairs into a dictionary within the function body
example

```
def print_all(**kwargs):
    “””Print out key-value pairs in **kwargs.”””
    #Print out the key-value pairs
    for key, value in kwargs.items():
        print(key + ‘: “ + value)
print_all(name=‘dumbledore’, job=‘headmaster’
```

output > job: headmaster name: dumbledore

Lambda functions

```
raise_to_power = lambda x, y: x ** y
raise_to_power(2,3)
```

output 8
example of a good time to use is with map functions
function map takes two arguments: map(func, seq)
map() applies the function to ALL elements in the sequence
map function is a type of anonymous function meaning we can pass lambda functions to map without even naming them
example

```
nums = [48, 6, 9 ,21, 1]
square_all = map(lambda num: num ** 2, nums
print(square_all)
```

**output will state that it is a map object
**to see our outputs we need to print it out as a list

```
print(list(square_all))
```

Errors and exceptions

```
def sqrt(x):
    try:
        return x ** 0.5
    except:
        print(‘x must be an int or float’)
```

** this makes clearer type error messages for users
sometimes you will actually want to raise an error
example for the squareroot of a negative number, we may prefer an error message over a complex number
example

```
def sqrt(x):
    if x < 0:
        raise ValueError(‘x must be non-negative’)
    try:
        return x ** 0.5
    except TypeError:
        print(‘x must be an int or float’)
```

Iterators vs. Iterables
Iterator example - for loop
iterate over a range object using a loop (ie characters in a string or a sequence of numbers)
produces next value with next()
example

```
for i in range(4):
    print(i)
```

Iterable examples - lists, strings, dictionaries, file connections
official definition - an object with an associated iter() method
**applying iter() to an utterable creates an iterator
what a for loop is doing:
it takes an iterable, creates the associated iterator object, and iterates over it

iter() will continue to call until there are no values left to return and then will throw a StopIteration error
Can call all values in one output by using a star (unpacks all elements)
example

```
word = ‘Data’
it = iter(word)
print(*it)
```

output D a t a instead of

```
D
a
t
a
```

without star

Iterating over dictionaries need to use items method

```
pythonistas = {‘hugo’: ‘bone-anderson’, ‘francis’: ‘castro’}
for key, value in pythonistas.items():
        print(key, value)
```

Iterating over file connections

```
file = open(‘file.txt’)
it = iter(file)
print(next(it))
```

Playing with iterators

enumerate()

allows us to add a counter to any iterable

a function that takes any iterable as argument and returns a special enumerate object, which consists of pairs containing the elements of the original iterable, along with their index within the iterable

we can then use the function lis to turn this enumerate object into a list of tuples and print it to see what it contains

example

```
avengers = [‘hawkeye’, ‘iron man’, ‘thor’, ‘quicksilver’]
e = enumerate(avengers)
print(type(e))
```

output class enumerate

```
e_list = list(e)
print(e_list)
```

output [(0, ‘hawkeye’), (1, ‘iron man’), (2, ‘thor’), (3, ‘quicksilver)]

**Can enumerate and unpack at same time

```
avengers = [see above]
for index, value in enumerate(avengers):
        print(index, value)
```

output

```
0 hawkeye
```

```
1 iron man
2 thor
3 quicksilver
```

**can use argument ‘start’ to start index at any point; default is 0

zip()

allows us to stitch together an arbitrary number of iterables and returns an iterator of tuples

example can zip two lists together

example

```
avengers = [see above]
names = [‘barton’, ‘stark’, ‘odinson’, ‘maxim off’]
z = zip(avengers, names)
print(type(z))
output class zip
z_list = list(z)
print(z_list)
output
[(‘hawkeye’, ‘barton’), (‘iron man’, ‘stark’), (‘thor’, ‘odin so’), (‘quicksilver’, ‘maxim off’)]
```

zip and unpack at same time

```
avengers = [see above]
names = [see above]
for z1, z2, in zip(avengers, names):
        print(z1, z2)
output
hawkeye barton
iron man stark
thor odinson
quicksilver maxim off
```

**can also use the ‘splat’ (ie star) operator to print all the elements

Loading data in chunks

when there is too much data to hold in memory

load data in chunks, run operation, get solution, store solution, discard chunk, and repeat

example - collecting sum

```
result = [ ]
for chunk in pd.read_csv(‘data.csv’, chunksize=1000):
        result.append(sum(chunk[‘x’]))
total = sum(result)
```

```
print(total)
```

another way of doing this

```
total = 0
for chunk in pd.read_csv(‘data.csv’, chunk size=1000):
        total += sum(chunk[‘x’])
print(total)
```

List comprehensions
an elegant way to define and create lists based on existing lists
example

```
nums = [12, 8, 21, 3, 16]
new_nums = [num + 1 for num in nums]
print(new_nums)
output [13, 9, 22, 4, 17]
```

**can write a list comprehension over any iterable
list comprehensions collapse ‘for’ loops for building lists into a single line
required components are:

- iterator
- iterator variable (represent members of iterable)
- output expression

Nested loop example

```
pairs_1 = [ ]
for num1 in range(0,2):
        for num2 in range(6,8):
                pairs_1.append((num1, num2))
print(pairs_1)
output [(0,6), (0,7), (1,6), 1,7)]
```

** how to do this with list comprehensions

```
pairs_2 =- [(num1, num2) for num1 in range(0,2) for num2 in range(6,8)]
```

example of conditionals on the iterable

```
[num **2 for num in range(10) if num % 2 == 0)
```

side note/reminder modulo (%) operator yields the remainder from the division of the first argument by the second

example of conditionals on the output expression

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

[output expression for iterator variable in iterable if predicate expression]

Dictionary or dict comprehensions

example

```
pos_neg = {num: -num for num in range(9)}
```

Generators

returns a generator object

same format as list comprehension except with () instead of []

main difference is that generators do not store the list in memory

it does not construct the list but instead keeps it as an object

this object can be iterated over to produce elements of the list as required

why use generators

good for large data sets, efficient iteration, infinite sequences, lazy evaluation (ie values are generated on-demand), and for pipelining and data processing (ie each step operates on one item at a time, improving code readability and reducing memory overhead)

**generators offer a memory-efficient and flexible way to generate sequences of values when dealing with large data sets, infinite sequences, or when you want to optimize performance by avoiding unnecessary memory allocations

** use yield in place of return

generator function example

```
def num_sequence(n):
        “””Generate values from 0 to n.”””
        i = 0
        while i < n
                yield i
                i += 1
```

Re-cap

List comprehension

Basic:

[output expression for iterator variable in iterable]

Advanced:

[output expression + conditional on output for iterator variable in iterable + conditional on iterable]

zip() accepts an arbitrary number of iterables and returns an iterator of tuples

defining a function

function header begins with def followed by function name with arguments inside parentheses followed by a colon

function body performs the computation that the function does and closes with the keyword return, followed by the value or values to return

Readline() method is used to read a single line from a file

Used in generators
Best used with text files
keeps track of current position in the file
returns line as a string
returns an empty string (‘’) when reaches the end