11_python_tutorial

July 25, 2024

0.1 Introduction

Python is a great general-purpose programming language on its own, but with the help of a few popular libraries (numpy, scipy, matplotlib) it becomes a powerful environment for scientific computing.

I expect that you have some experience with Python; and if you don't, this section will serve as a quick crash course

In this tutorial, we will cover:

- Basic Python: Basic data types (Containers, Lists, Dictionaries, Sets, Tuples), Functions, Classes

0.2 A Brief Note on Python Versions

As of Janurary 1, 2024, Python has officially dropped support for `python2`. We'll be using Python 3.10 for this iteration of the course. You can check your Python version at the command line by running `python --version`.

[ ]:
```
# checking python version
!python --version
```

```
Python 3.10.12
```

Don't Miss Any Updates!

Before we continue, we have a humble request, to be among the first to hear about future updates of the course materials, simply enter your email below, follow us on (formally Twitter), or subscribe to our YouTube channel.

0.3 Basics of Python

Python is a high-level, dynamically typed multiparadigm programming language. Python code is often said to be almost like pseudocode, since it allows you to express very powerful ideas in very few lines of code while being very readable. As an example, here is an implementation of the classic quicksort algorithm in Python:

[ ]:
```
def quicksort(array):
    if len(array) <= 1:
        return array
    pivot = array[len(array) // 2]
    left = [number for number in array if number < pivot]
    middle = [number for number in array if number == pivot]
    right = [number for number in array if number > pivot]
    return quicksort(left) + middle + quicksort(right)

quicksort([3,6,8,10,1,2,1])
```

[ ]:
```
[1, 1, 2, 3, 6, 8, 10]
```

[ ]:
```
sorted('listen', reverse=True)
```

[ ]:
```
['t', 's', 'n', 'l', 'i', 'e']
```

0.3.1 Variables

Variables are stores of value

[ ]:
```
name = 'Juma'
age = 19
id_number = 190045
```

Rules to consider

- Variable names should be meaningful eg `number` instead of `x`
- Variable names should contain only alpha-numberic characters, and maybe under_scores
- Variable names can only start with letters or an underscore
- Variable name cannot contain special characters
- Variables names are case sensitive

0.4 Examples of variables

For this course, we'll use snake case for *quick* variables

[ ]:
```
name = 'Eva'
list_of_names = ['Eva', 'Shafara', 'Bob'] # snake case
```

we'll use camel case for function variables

[ ]:
```python
def calculateBMI(weight_kg, height_m): # camel case
    bmi = weight_kg / height_m ** 2
    rounded_bmi = round(bmi, 3)
    return rounded_bmi
```

finally, we'll use pascal case for class variables

[ ]:
```python
class MathFunction:    # Pascal Case
    def __init__(self, number):
        self.number = number

    def square(self):
        return self.number ** 2

    def cube(self):
        return self.number ** 3
```

0.4.1 Basic data types

Numbers Integers and floats work as you would expect from other languages:

[ ]:
```python
number = 3
print('Number: ', number)
print('Type: ', type(number))
```

```
Number:  3
Type:  <class 'int'>
```

[ ]:
```python
# Quick number arithmetics

print(number + 1)   # Addition
print(number - 1)   # Subtraction
print(number * 2)   # Multiplication
print(number ** 2)  # Enumberponentiation
```

```
4
2
6
9
```

[ ]:
```python
# Some compound assingment operators

number += 1 # number = number + 1
print(number)
number *= 2
print(number)
number /= 1 # number = number / 1
print(number)
```

```
number -= 2
print(number)
```

```
4
8
8.0
6.0
```

[ ]:
```
number = 2.5
print(type(number))
print(number, number + 1, number * 2, number ** 2)
```

```
<class 'float'>
2.5 3.5 5.0 6.25
```

[ ]:
```
# complex numbers
vector = 2 + 6j
type(vector)
```

[ ]:
```
complex
```

Booleans Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols:

[ ]:
```
t, f = True, False
type(t)
```

[ ]:
```
bool
```

Now we let's look at the operations:

[ ]:
```
# Logical Operators

print(t and f) # Logical AND;
print(t or f)  # Logical OR;
print(not t)   # Logical NOT;
print(t != f)  # Logical XOR;
```

```
False
True
False
True
```

Strings A string is a sequence of characters under some quotes. Eg.

[ ]:
```
hello = 'hello'   # String literals can use single quotes
world = "world"   # or double quotes; it does not matter
print(hello, len(hello))
```

```
hello 5
```

[ ]:
```
# We can string in python
full = hello + ' ' + world  # String concatenation
print(full)
```

```
hello world
```

[ ]:
```
hw12 = '{} {} {}'.format(hello, world, 12)  # string formatting
print(hw12)
```

```
hello world 12
```

[ ]:
```
statement = 'I love to code in {}'
modified = statement.format('JavaScript')
print(modified)
```

```
I love to code in JavaScript
```

[ ]:
```
# formatting by indexing
statement = '{0} loves to code in {2} and {1}'
statement.format('Juma', 'Python', 'JavaScript')
```

[ ]:
```
'Juma loves to code in JavaScript and Python'
```

[ ]:
```
# formatting by name
statement = '{name} loves to code in {language1} and {language2}'
statement.format(language2='Python', name='Juma', language1='JavaScript')
```

[ ]:
```
'Juma loves to code in JavaScript and Python'
```

[ ]:
```
# String Literal Interpolation
name = 'Juma'
language1 = 'JavaScript'
language2 = 'Python'

statement = f'{name} loves to code in {language1} and {language2}'

print(statement)
```

```
Juma loves to code in JavaScript and Python
```

String objects have a bunch of useful methods; for example:

[ ]:
```
string_ = "hello"
print(string_.capitalize())  # Capitalize a string
print(string_.upper())       # Convert a string to uppercase; prints "HELLO"
print(string_.rjust(7))      # Right-justify a string, padding with spaces
print(string_.center(7))     # Center a string, padding with spaces
```

```
print(string_.replace('l', '(ell)'))  # Replace all instances of one substring␣
 ↪with another
print('  world '.strip())  # Strip leading and trailing whitespace
```

```
Hello
HELLO
  hello
 hello
he(ell)(ell)o
world
```

[ ]:
```
statement = 'i love to code in Python '

capitalized = statement.capitalize()
upped = statement.upper()
replaced = statement.replace('Python', 'javascript')
statement.strip()
```

[ ]:
```
'i love to code in Python'
```

You can find a list of all string methods in the documentation.

0.4.2 Containers

- Python containers (collections) are objects that we use to group other objects
- Python includes several built-in container types: lists, dictionaries, sets, and tuples.

Lists A list is an ordered collection of python objects or elements. A list can contain objects of different data types

[ ]:
```
list_of_numbers = [3, 1, 2]   # Create a list
print(list_of_numbers)
print(list_of_numbers[2])
print(list_of_numbers[-1])     # Negative indices count from the end of the␣
 ↪list; prints "2"
```

```
[3, 1, 2]
2
2
```

[ ]:
```
list_of_numbers[2] = 'foo'    # replacing a specific value in a list
print(list_of_numbers)
```

```
[3, 1, 'foo']
```

[ ]:
```
list_of_numbers.append('bar') # Add a new element to the end of the list
print(list_of_numbers)
```

```
[3, 1, 'foo', 'bar']
```

```
[ ]: last_item = list_of_numbers.pop()     # Remove and return the last element of␣
      ↪the list
     print(last_item)    # returns the last item
     print(list_of_numbers) # Modifies the original list
```

```
bar
[3, 1, 'foo']
```

Research on: - `del` - `remove()`

As usual, you can find all the gory details about lists in the documentation.

Slicing In addition to accessing list elements one at a time, Python provides concise syntax to access a range of values in a list; this is known as slicing:

```
[ ]: list_of_numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
     print(list_of_numbers)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
[ ]: print(list_of_numbers)          # Prints "[0, 1, 2, 3, 4]"
     print(list_of_numbers[2:4])    # Get a slice from index 2 to 4 (exclusive);␣
      ↪prints "[2, 3]"
     print(list_of_numbers[2:])     # Get a slice from index 2 to the end; prints␣
      ↪"[2, 3, 4]"
     print(list_of_numbers[:2])     # Get a slice from the start to index 2␣
      ↪(exclusive); prints "[0, 1]"
     print(list_of_numbers[:])      # Get a slice of the whole list; prints ["0, 1,␣
      ↪2, 3, 4]"
     print(list_of_numbers[:-1])    # Slice indices can be negative; prints ["0, 1,␣
      ↪2, 3]"
     list_of_numbers[2:4] = [8, 9] # Assign a new sublist to a slice
     print(list_of_numbers)         # Prints "[0, 1, 8, 9, 4]"
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 3]
[2, 3, 4, 5, 6, 7, 8, 9]
[0, 1]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8]
[0, 1, 8, 9, 4, 5, 6, 7, 8, 9]
```

Loops A `for loop` is used to loop through (or iterate) over a sequence of objects (iterable objects). Iterable objects in python include strings, lists, sets etc

You can loop over the elements of a list like this:

[ ]:
```
list_of_animals = ['cat', 'dog', 'monkey']

for animal in list_of_animals:
    print(animal)
```

```
cat
dog
monkey
```

[ ]:
```
list_of_numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
list_of_squared_numbers = []

for number in list_of_numbers:
    list_of_squared_numbers.append(pow(number, 2))

list_of_squared_numbers
```

[ ]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 0]
```

If you want access to the index of each element within the body of a loop, use the built-in `enumerate` function:

[ ]:
```
animals = ['cat', 'dog', 'monkey']

for index, animal in enumerate(animals):
    print(f'{index}: {animal}')
```

```
0: cat
1: dog
2: monkey
```

List comprehensions:

[ ]:
```
numbers = [0, 1, 2, 3, 4]
squares = []

for number in numbers:
    squares.append(pow(number, 2))

print(squares)
```

```
[0, 1, 4, 9, 16]
```

You can make this code simpler using a list comprehension:

[ ]:
```
list_of_numbers = [0, 1, 2, 3, 4]

squares = [pow(number, 2) for number in list_of_numbers]
```

```
print(squares)
```

```
[0, 1, 4, 9, 16]
```

List comprehensions can also contain conditions:

[ ]:
```
numbers = [0, 1, 2, 3, 4]

even_squares = [pow(number, 2) for number in numbers if number % 2 == 0]

print(even_squares)
```

```
[0, 4, 16]
```

Research: - How to combine lists

Dictionaries

- A dictionary is an unordered and mutable collection of items
- A dictionary is created using curly brackets
- Each item in a dictionary contains a key/value pair

[ ]:
```
# creating a dictionary
person = {
    'first_name': 'Juma',
    'last_name': 'Shafara',
    'age': 51,
    'married': True
}
person
```

[ ]:
```
{'first_name': 'Juma', 'last_name': 'Shafara', 'age': 51, 'married': True}
```

[ ]:
```
# accessing items in a dictionary
first_name = person['first_name']
last_name = person['last_name']
full_name = first_name + ' ' + last_name

# display
full_name
```

[ ]:
```
'Juma Shafara'
```

[ ]:
```
# add items to a dictionary
person['hobby'] = 'Coding'
person
```

[ ]:
```
{'first_name': 'Juma',
 'last_name': 'Shafara',
```

```
 'age': 51,
 'married': True,
 'hobby': 'Coding'}
```

[ ]:
```
email = person.get('email', 'email not available')
print(email)
```

```
email not available
```

[ ]:
```
# modifying a value in a dictionay
person['married'] = False
person
```

[ ]:
```
{'first_name': 'Juma',
 'last_name': 'Shafara',
 'age': 51,
 'married': False,
 'hobby': 'Coding'}
```

[ ]:
```
# remove an item from a dictionary
person.pop('age')
person
```

[ ]:
```
{'first_name': 'Juma',
 'last_name': 'Shafara',
 'married': False,
 'hobby': 'Coding'}
```

Research: - How to remove an item using the `del` method - How to iterate over objects in a dictionary - Imitate list comprehension with dictionaries

You can find all you need to know about dictionaries in the documentation.

Sets

- A set is an unordered, immutable collection of distinct elements.
- A set is created using curly braces
- The objects are placed inside the brackets and are separated by commas
- As a simple example, consider the following:

[ ]:
```
animals = {'cat', 'dog'}

print('cat' in animals)   # Check if an element is in a set; prints "True"
print('fish' not in animals)  # prints "True"
```

```
True
True
```

[ ]:
```
animals.add('fish')      # Add an element to a set

print('fish' in animals) # Returns "True"

print(len(animals))       # Number of elements in a set;
```

```
True
3
```

[ ]:
```
animals.add('cat')       # Adding an element that is already in the set does␣
↪nothing
print(len(animals))

animals.remove('cat')    # Remove an element from a set
print(len(animals))
```

```
3
2
```

Research: - How to remove with `discard()` - How to remove with `pop()` - How to combine sets - How to get the difference between 2 sets - What happens when we have repeated elements in a set

Loops: Iterating over a set has the same syntax as iterating over a list; however since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set:

[ ]:
```
animals = {'cat', 'dog', 'fish'}

for index, animal in enumerate(animals):
    print(f'{index}: {animal}')
```

```
0: fish
1: cat
2: dog
```

Set comprehensions: Like lists and dictionaries, we can easily construct sets using set comprehensions:

[ ]:
```
from math import sqrt

print({int(sqrt(x)) for x in range(30)})
```

```
{0, 1, 2, 3, 4, 5}
```

Tuples

- A tuple is an (immutable) ordered list of values.
- A tuple is in many ways similar to a list; one of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot. Here is a trivial example:

[ ]:
```
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple keys
t = (5, 6)        # Create a tuple
print(type(t))
print(d[t])
print(d[(1, 2)])
```

```
<class 'tuple'>
5
1
```

[ ]:
```
# t[0] = 1
```

Research: - Creating a tuple - Access items in a tuple - Negative indexing tuples - Using range of indexes - Getting the length of items in a tuple - Looping through a tuple - Checking if an item exists in a tuple - How to combine tuples - Prove that tuples are immutable

0.4.3 Functions

- A function is a group of statements that performs a particular task
- Python functions are defined using the `def` keyword. For example:

[ ]:
```
def overWeightOrUnderweightOrNormal(weight_kg:float, height_m:float) -> str:
    '''
    Tells whether someone is overweight or underweight or normal
    '''
    height_m2 = pow(height_m, 2)
    bmi = weight_kg / height_m2
    rounded_bmi = round(bmi, 3)
    if bmi > 24:
        return 'Overweight'
    elif bmi > 18:
        return 'Normal'
    else:
        return 'Underweight'

overWeightOrUnderweightOrNormal(67, 1.7)
```

[ ]:
```
'Normal'
```

We will often use functions with optional keyword arguments, like this:

[ ]:
```
bmi = calculateBMI(height_m=1.7, weight_kg=67)

print(bmi)
```

```
23.183
```

[ ]:
```python
def greet(name:str='You')->str:
    """
    This function greets people by name
    Example1:
    >>> greet(name='John Doe')
    >>> 'Hello John Doe'
    Example2:
    >>> greet()
    >>> 'Hello You'
    """
    return f'Hello {name}'

# greet('Eva')
?greet
```

```
Signature: greet(name: str = 'You') -> str
Docstring:
This function greets people by name
Example1:
>>> greet(name='John Doe')
>>> 'Hello John Doe'
Example2:
>>> greet()
>>> 'Hello You'
File:      /tmp/ipykernel_66386/2049930273.py
Type:      function
```

0.4.4 Classes

- In python, everything is an object
- We use classes to help us create new object
- The syntax for defining classes in Python is straightforward:

[ ]:
```python
class Person:
    first_name = 'John'
    last_name = 'Tong'
    age = 20
```

[ ]:
```python
# Instantiating a class
object1 = Person()

print(object1.first_name)
print(object1.last_name)
print(object1.age)

print(f'object1 type: {type(object1)}')
```

```
John
```

```
Tong
20
object1 type: <class '__main__.Person'>
```

[ ]:
```python
# Instantiating a class
object2 = Person()

print(object2.first_name)
print(object2.last_name)
print(object2.age)
```

```
John
Tong
20
```

[ ]:
```python
class Person:
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def greet(self, name):
        return f'Hello {name}'
```

[ ]:
```python
object1 = Person('Juma', 'Shafara', 24)
print(object1.first_name)
print(object1.last_name)
print(object1.age)
print(type(object1))
```

```
Juma
Shafara
24
<class '__main__.Person'>
```

[ ]:
```python
object2 = Person('Eva', 'Ssozi', 24)
print(object2.first_name)
print(object2.last_name)
print(object2.age)
print(object2.greet('Shafara'))
print(type(object2))
```

```
Eva
Ssozi
24
Hello Shafara
<class '__main__.Person'>
```

[ ]:
```
class Student(Person):
    def __init__(self, first_name, last_name, age, id_number, subjects=[]):
        super().__init__(first_name, last_name, age)
        self.id_number = id_number
        self.subjects = subjects

    def addSubject(self, subject):
        self.subjects.append(subject)
```

[ ]:
```
student1 = Student('Calvin', 'Masaba', 34, '200045', ['math', 'science'])
```

[ ]:
```
student1.addSubject('english')
```

[ ]:
```
student1.subjects
```

[ ]:
```
['math', 'science', 'english']
```

Research: - Inheritance: This allows to create classes that inherit the attributes and methods of another class

What's on your mind? Put it in the comments!