Python training notes

By **Vinod Kumar Kayartaya**. You may also be interested in this video tutorial on YouTube.

Table of contents

[Back to TOC]

Overview

Python is a general purpose, interpreted language created by `Guido van Rossum`, and was released in the year 1991. Python encourages code readability and write fewer lines to express the concepts.

Python supports both procedural and object orinented programming. Many language features support functional programming and aspect-oriented programming.

The language was named after the popular TV show, `Monty Python's Flying Circus` (1969-74).

Features of Python include:

- Free and open source
- Interpreted, high level
- Portability
- Extensible
- Embeddable
- Vast library of packages
- Supports procedure oriented, object oriented, functional, and aspect oriented

Python being a very simple and easy to learn language, can be used for teaching programming to kids and non-programmers. However, the language offers so many powerful features, it can be used for variety of applications such as Web applications, Scientific and Numeric applications, and creating software prototypes.

[Back to TOC]

Hello, Python

When you install Python, you get a command `python`. When you run the command with out supplying any script file, it will open a shell known as REPL (Read Eval Print and Loop). The REPL has a command prompt that allows us to execute Python commands directly.

```
$ python
Python 3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> message = "Hello, Python!"
>>> print(message)
Hello, Python!
>>> exit()
$
```

Each command you enter will be read, evaluated, and the result is printed. Again it will repeat the same by printing the prompt and waiting for user inputs. To close the REPL, just call the

`exit()` function. REPL is a very quick way to test some of the commands. So, many developers keep it handy.

Another and most common way to work with Python is to write a script file and pass it to the `python` command. Let's write our first script and save the file as `hello.py`.

```
# This is a python script
message = "Hello, Python!"
print(message)
```

Any line that starts with the pound (#) symbol is considered as a `code comment`, and the interpretor will simply ignore the same. To execute the code, open a terminal (command prompt) and run the command:

```
$ python hello.py
```

As expected, it will simply print out the message "Hello, Python!".

On Unix based operating system (Linux, Ubuntu, MacOS etc), we may also add a special comment (usually known as hashbang), pointing to the python executable:

```
#! /usr/bin/python
```

In order to execute the file, we have to add `x` permission to the script file, and then we can simply run the program using the filename itself.

```
$ chmod u+x hello.py
$ ./hello.py
$ Hello, Python!
```

[Back to TOC]

Variables in Python

A variable represents a value in the memory. Unlike many other popular languages, in Python there is no need to declare a variable in advance. It is dynamically typed. The datatype of the variable changes based on the value you assign to it. The naming convention for variables (and identifiers in general) is defined in `PEP 8`.

PEP stands for Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. The PEP should provide a concise technical specification of the feature and a rationale for the feature.

Some good variable convention:

- lowercase
- underscore_between_words
- don't start with numbers

```
# Working with variables

>>> var_a = 1
>>> type(var_a)
<class 'int'>
>>>
>>> var_a =
9873459863486962394686823498734598634869623946868234987345986348696239468682349873459863486962394686823498734598634869623946868234634869623946868234987345986348696239468682349873459863486962394686823498734598634869623946868234987345986348696239468682349873459863486962394686823498734598634869623946868234
>>> type(var_a)
<class 'int'>
>>>
>>> var_a = 1.2
>>> type(var_a)
<class 'float'>
>>>
>>> var_a = "Vinod"
>>> type(var_a)
<class 'str'>
>>>
>>> var_a = 'Vinod'
>>> type(var_a)
<class 'str'>
>>>
>>> var_a = True
>>> type(var_a)
<class 'bool'>
>>>
>>> var_a = None
>>> type(var_a)
<class 'NoneType'>
```

Couple of things to observe:

- Numbers can be integers or float.
- Python allow extremely large values for numericals with out losing any precision.
- Single and double quotes mean the same thing - they are just strings.
- Boolean literals are `True` and `False`.
- `None` represents absense of value in a variable. In a boolean context, it will evaluate to False.
- Unlike C/C++/Java/C# the statements do not end with semicolon.

A value in Python is an object in the memory and has a unique id associated with it. Although it is not very useful practically, we can always find the id using the `id(var)` function.

```
>>> var_a = "Vinod"
>>> id(var_a)
4331162848
>>> var_a = 123
>>> id(var_a)
4297627840
>>> var_a = var_a + 10
>>> id(var_a)
4297628160
>>>
```

With numeric variables, we can do the following maths:

Operator	Used for	Example	Result
+	Addition	a = 7+2	9
-	Subtraction	a = 7-2	5
*	Multiplication	a = 7*2	14
/	Division	a = 7/2	3.5
//	Integer division	a = 7//2	3
**	Exponent	a = 7**2	49

[Back to TOC]

Strings

A String (class `str`) is a value with in single or double quotes. Strings are immutable (they can't be changed in place). By enclosing a text in 3 single or double quotes, we can even have multiline strings. These multiline strings are usually used for documentation purposes.

```
my_name = "Vinod"
my_city = 'Bangalore'
doc = '''This is a multiline string
usually used for code documentation
inside a module, class, function etc.'''
```

Strings can also contain escape characters that would produce different output:

Escape sequence	Output
\n	New line

Escape sequence	Output
\t	Tab
\'	Single quoute
\"	Double quoute
\\	Back slash
\b	ASCII backspace
\u0C95	A unicode character

We can format a string in two ways:

C style - using % format specifiers

```
>>> my_name = "Vinod"
>>> my_city = "Bangalore"
>>> info = "%s lives in %s" % (my_name, my_city)
>>> info
'Vinod lives in Bangalore'
>>>
```

PEP 3101 style - using { }

```
>>> my_name = "Vinod"
>>> my_city = "Bangalore"
>>> info = "{0} lives in {1}".format(my_name, my_city)
>>> info
'Vinod lives in Bangalore'
>>>
```

Most useful `string` functions

- capitalize
  - Return a capitalized version of the string, i.e. make the first character have upper case and the rest lower case.
  - For example, "VINOD KUMAR".capitalize() returns "Vinod kumar"
- center
  - Return the string, centered in a string of length width. Padding is done using the specified fill character (default is a space)
  - For example, "VINOD".center(25, "*") returns "******VINOD******"
- count
  - Return the number of non-overlapping occurrences of substring sub in the string
  - For example, "ANANDA RAMA".count("AN") returns 2
- endswith
  - Return True if the string ends with the specified suffix, False otherwise.

    - For example, "VINOD".endswith("D") returns True
- find
    - Return the lowest index in string where substring sub is found, such that sub is contained within the string
    - For example, "VINOD KUMAR".find("MA") returns 8, and "VINOD KUMAR".find("xy") returns -1
- index
    - Like the **find** function, but raise ValueError when the substring is not found
- isalnum
    - Return True if all characters in string are alphanumeric and there is at least one character in string False otherwise
- isalpha
    - Return True if all characters in string are alphabetic and there is at least one character in S, False otherwise
- isdecimal
    - Return True if there are only decimal characters in string False otherwise
- isdigit
    - Return True if all characters in string are digits and there is at least one character in string False otherwise
- islower
    - Return True if all cased characters in string are lowercase and there is at least one cased character in string False otherwise
- isnumeric
    - Return True if there are only numeric characters in string False otherwise
- isspace
    - Return True if all characters in string are whitespace and there is at least one character in string False otherwise
- istitle
    - Return True if string is a titlecased string and there is at least one character in string i.e. upper- and titlecase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise
- isupper
    - Return True if all cased characters in string are uppercase and there is at least one cased character in string False otherwise
- join
    - Return a string which is the concatenation of the strings in the iterable
    - For example, "-".join(["Vinod", "Kumar", "John", "Jane"]) returns "Vinod-Kumar-John-Jane"
- lower
    - Return a copy of the string string converted to lowercase
- replace
    - Return a copy of string with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced
    - For example, "black white orange yellow".replace("orange", "red") returns "black white red yellow"
- split

    - Return a list of the words in string using sep as the delimiter string.
    - For example, "black,white,orange,yellow".split(",") returns a list ["black", "white", "orange", "yellow"]
- startswith
    - Return True if string starts with the specified prefix, False otherwise
    - For example, "VINOD".endswith("V") returns True
- strip
    - Return a copy of the string string with leading and trailing whitespace removed
- swapcase
    - Return a copy of string with uppercase characters converted to lowercase and vice versa
    - For example, "Vinod Kumar Kayartaya".swapcase() returns "vINOD kUMAR kAYARTAYA"
- title
    - Return a titlecased version of string
    - For example, "VINOD KUMAR KAYARTAYA".title() returns "Vinod Kumar Kayartaya"
- upper
    - Return a copy of string converted to uppercase

Python does not allow addition / concatenation of values of different types.

```
>>>
>>> n1 = 100
>>> n2 = 12
>>> n1 + n2
112
>>> s1 = "Vinod"
>>> s2 = "Kumar"
>>> s1 + s2
'VinodKumar'
>>> n1 + s1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

But one strange operation is that you can multiply a string by a number!

```
>>>
>>> "-" * 10
'----------'
>>> "vinod" * 3
'vinodvinodvinod'
>>> 10 * "-"
'----------'
>>>
```

Selection construct

For conditional execution, we can use the `if` statements. The syntax for the same is:

```
Syntax:

    if <condition> :
        <statements>
```

Two things to observe from the above syntax:

1. After the boolean criteria, we have a colon symbol
2. The statements to be executed if the boolean criteria evaluates to true should be indented. This is one of the Python's phylosophy - the code should be readable. In most other languages (C/C++/Java/C#), we would use a pair of curly braces group the statements to be executed when the condition is true. In some other languages (VB, Oracle and MySQL stored procedures, Unix shell scripts), we would use a statement like `end-if` to denote the end of the statemetns to be executed when the condition is true.

Here is an example. It shows how to read a value from the keyboard and print if the number is even or odd.

```
# Python script to check if the input number is even or odd

# In Python 3, the input function returns a 'str' object.
num = input("Enter a number: ")

# converting the num from 'str' to 'int' type.
num = int(num)

message = "The input number is an odd number."

if num % 2 == 0:
    message = "The input number is an even number."

print(message)
```

When executed, here is the output:

```
$ python3 if1.py
Enter a number: 23
The input number is an odd number.
$
$ python3 if1.py
Enter a number: 34
```

```
The input number is an even number.
$
```

You might have observed that I have assigned a default value to the variable `message`. I am only chaning it's value, if the condition evaluates to true. Alternately we may use the `else` block to assign the alternate message to the variable.

```
Syntax:

    if <condition> :
        <statements>
    else:
        <statements>
```

Let's have a look at another example. This time, I want to read the age of the user and print if he/she can vote or not!

```
# Check if the user can vote or not.

age = int(input("Enter your age: "))

if age>=18:
    print("Yes! you can and should vote.")
else:
    print("Hey, just wait for another %d years to vote." % (18-age))
```

And the output of the same is:

```
$ python3 ifelse1.py
Enter your age: 43
Yes! you can and should vote.
$
$ python3 ifelse1.py
Enter your age: 12
Hey, just wait for another 6 years to vote.
$
```

In some cases, when the condition evaluates to false, we would like to check one more condition. And when the second one fails, we would like to check for another, and so on. We can do this with the combination of `if-elif-else`.

```
Syntax:

    if <condition> :
        <statements>
    elif <condition> :
        <statements>
```

```
    ...
        ...
    else:
        <statements>
```

This example shows how to find the number of days in a given month.

```
# Find the number of days in the input month.

mon = int(input("Enter a month (1-12): "))

if mon == 2:
    print("Maximum number of days in the input month is 28 or 29.")
elif mon in (4, 6, 9, 11):
    print("Maximum number of days in the input month is 30.")
elif mon in (1, 3, 5, 7, 8, 10, 12):
    print("Maximum number of days in the input month is 31.")
else:
    print("You have entered an invalid month number.")
```

And here are few sample execution of the same.

```
$ python3 ifelifelse1.py
Enter a month (1-12): 3
Maximum number of days in the input month is 31.
$
$ python3 ifelifelse1.py
Enter a month (1-12): 4
Maximum number of days in the input month is 30.
$
$ python3 ifelifelse1.py
Enter a month (1-12): 2
Maximum number of days in the input month is 28 or 29.
$
$ python3 ifelifelse1.py
Enter a month (1-12): 17
You have entered an invalid month number.
$
```

Python does not have ternary operator (like in C/C++/Java/C#), which would have been handy. But, if-else combination can be used for the same too:

```
condition = year%400==0 or (year%4==0 and year%100!=0)
days = 29 if condition else 28
```

[Back to TOC]

Using loops

A loop (iteration) is a programming construct that executes one or more statements repeatedly until the loop condition fails. There are two types of loops - `while` and `for`.

The "while" loop

```
while <condition> :
    <statements>
```

Here is an example that prints the multiplication table for the input number:

```
# Accept a number and print the multiplication table for the same.

num = int(input("Enter a number: "))

n = 1

while n<=10:
    p = n * num
    print("%d X %d = %d" % (num, n, p))
    n = n + 1
```

The result of script execution:

```
$ python3 table.py
Enter a number: 23
23 X 1 = 23
23 X 2 = 46
23 X 3 = 69
23 X 4 = 92
23 X 5 = 115
23 X 6 = 138
23 X 7 = 161
23 X 8 = 184
23 X 9 = 207
23 X 10 = 230
$
```

The "for" loop

The `for` statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object:

```
for <var> in <sequence>:
    <statements>
```

From the given sequence, values are assigned one by one to the variable, until all of them were exhausted. For example, the following example reverses the input name:

```
# reverse the input name.

name = input("Enter your name: ")
rev = ""

for c in name:
    rev = c + rev

print("reverse of '%s' is '%s'" % (name, rev))
```

which produces the following output:

```
$ python3 print1.py
Enter your name: Vinod
reverse of 'Vinod' is 'doniV'
$
```

We can also use the `range` object to loop from a starting to an ending number. For example, if we want to iterate from 1 to 10, we may very well use the `for loop` with the `range` object.

```
# Print the factorial of the input number.

num = int(input("Enter a number: "))
fact = 1

for n in range(1, num+1):
    fact = fact * n

print("Factorial of %d is %d" % (num, fact))
```

You may have noticed that I used `num+1` as the second parameter to the range function. This is because, the range function will only represent a sequence from the first parameter to 1 less than the second parameter. So if we have to loop from 1 to 10, we have to supply `1, 11` to the range function.

Here is the output from the above script:

```
$ python3 factorial.py
Enter a number: 5
Factorial of 5 is 120
$
```

We may also use a step value to alter the sequence of numbers generated by the range function. For example, the following code generates a number from 1 to 25, skipping every

alternate numbers:

```
# print alternate numbers.

for i in range(1, 26, 2):
    print(i)
```

We can use a negative number to loop in reverse. Of course, the first two parameters to be reversed too.

```
# print numbers in reverse

for i in range(10, 0, -1):
    print(i)
```

[Back to TOC]

Lists

A `list` is a data structure that represents a sequence of objects. An empty list can be created using a pair of square brackets `[]` or using the constructor.

```
>>>
>>> names = []
>>> type(names)
<class 'list'>
>>>
>>>
>>> names = list()
>>> type(names)
<class 'list'>
>>>
```

A list object may contain values of different types, although a list generally contains objects of similar types. The values are separated using commas and enclosed with in the square brackets. An element in a list can be accessed using `index`. The values can be changed in a list, and hence the lists are `mutable` objects.

```
>>>
>>> names = ["vinod", "shyam", "john", "jane"]
>>> names
['vinod', 'shyam', 'john', 'jane']
>>> names[1]
'shyam'
>>> names[3] = "jane doe"
```

```
>>> names
['vinod', 'shyam', 'john', 'jane doe']
>>>
```

You can append one ore members using the `+=` operator or the `append` function.

```
>>> names
['vinod', 'shyam', 'john', 'jane doe']
>>> names += ["vishal"]
>>> names
['vinod', 'shyam', 'john', 'jane doe', 'vishal']
>>> names.append("vijay")
>>> names
['vinod', 'shyam', 'john', 'jane doe', 'vishal', 'vijay']
```

The slice `:` operator allows us to get or set part of a list. The `len` function returns the number of elements in the list.

```
>>> names =['vinod', 'shyam', 'john', 'jane doe', 'vishal', 'vijay']
>>> names[3:5]
['jane doe', 'vishal']
>>> len(names)
6
>>> names[3:5] = []
>>> len(names)
4
>>> names
['vinod', 'shyam', 'john', 'vijay']
>>>
```

The list can be iterated using the `for` loop.

```
>>> names = ['vinod', 'shyam', 'john', 'vijay']
>>> for name in names:
...     print("Hello, %s" % name)
...
Hello, vinod
Hello, shyam
Hello, john
Hello, vijay
>>>
```

Methods of "list" objects

- list.append(x)
    - Add an item to the end of the list.
    - Equivalent to `a[len(a):] = [x]` or `a += [x]`.
- list.extend(iterable)

    - Extend the list by appending all the items from the iterable.
    - Equivalent to `a[len(a):] = iterable` or `a += iterable`

- list.insert(i, x)
    - Insert an item at a given position.
    - The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list
    - and a.insert(len(a), x) is equivalent to a.append(x).
- list.remove(x)
    - Remove the first item from the list whose value is x.
    - It is an error if there is no such item.
- list.pop([i])
    - Remove the item at the given position in the list, and return it.
    - If no index is specified, a.pop() removes and returns the last item in the list. (The square brackets around the i in the method signature denote that the parameter is optional, not that you should type square brackets at that position.
- list.clear()
    - Remove all items from the list. Equivalent to `del a[:]`.
- list.index(x[, start[, end]])
    - Return zero-based index in the list of the first item whose value is x.
    - Raises a ValueError if there is no such item.
    - The optional arguments start and end are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list.
    - The returned index is computed relative to the beginning of the full sequence rather than the start argument.
- list.count(x)
    - Return the number of times x appears in the list.
- list.sort(key=None, reverse=False)
    - Sort the items of the list in place (the arguments can be used for sort customization, see sorted() for their explanation).
- list.reverse()
    - Reverse the elements of the list in place.
- list.copy()
    - Return a shallow copy of the list. Equivalent to a[:].

Using Lists as Stacks

```
>>> stack = ["vinod", "shyam", "john"]
>>> stack.append("jane")
>>> stack.append("smith")
```

```
>>> stack
["vinod", "shyam", "john", "jane", "smith"]
>>> stack.pop()
"smith"
>>> stack
["vinod", "shyam", "john", "jane"]
>>> stack.pop()
"jane"
>>> stack.pop()
"john"
>>> stack
["vinod", "shyam"]
```

[Back to TOC]

Tuples

A `tuple` is a read only data structure. Once the variable is assigned with the values, it can not be changed. Hance they are known as immutable collections. A tuple can be created by simply assigning one or more comma separated values. There are only two methods - `count` and `index`.

Following are different ways to create a tuple:

```
>>> names = ()
>>> type(names)
<class 'tuple'>
>>> names = "vinod", "shyam", "john", "jane"
>>> type(names)
<class 'tuple'>
>>> names = "vinod",  # notice the comma at the end
>>> type(names)
<class 'tuple'>
>>> names = ("vinod",)  # notice the comma after "vinod"
>>> type(names)
<class 'tuple'>
>>> names = tuple(["vinod", "shyam", "john", "jane"])
>>> type(names)
<class 'tuple'>
>>> names
('vinod', 'shyam', 'john', 'jane')
>>> names.index("shyam")
1
>>> names.count("john")
1
>>>
```

The statement `names = "vinod", "shyam"` is an example of tuple `packing`. The reverse operation is also possible, which is called `unpacking the tuple`.

```
>>> names = "vinod", "shyam"
>>> type(names)
<class 'tuple'>
>>> names
('vinod', 'shyam')
>>> n1, n2 = names
>>> n1
'vinod'
>>> n2
'shyam'
>>>
```

[Back to TOC]

Sets

A `set` is an `unordered collection` with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference. Curly braces or the set() constructor can be used to create sets. To create an empty set you have to use set() constructor, and not {}; the latter creates an empty `dictionary`.

Here are some examples to create a set:

```
>>> names = {"vinod", "shyam", "john", "jane", "vinod", "vinod"}
>>> names
{'jane', shyam', vinod', john'}
>>> type(names)
<class 'set'>
>>> names.add("kishore")
>>> names
{'jane', kishore', 'john', 'shyam', 'vinod'}
>>> names = set(["vinod", "shyam", "vinod", "vinod"])
>>> names
{'shyam', 'vinod'}
>>>
>>> x = set("anantharama")
>>> x
{'m', 'h', 'n', 'r', 'a', 't'}
```

Notice that when a set constructor is passed a string, a new `set` consisting of unique letters in the string will be created.

The set class includes these methods:

- add
    - Add an element to a set.

        - This has no effect if the element is already present.
- clear
    - Remove all elements from this set.
- copy
    - Return a shallow copy of a set.
- difference
    - Return the difference of two or more sets as a new set.
    - i.e. all elements that are in this set but not the others.
- difference_update
    - Remove all elements of another set from this set.
- discard
    - Remove an element from a set if it is a member.
        - If the element is not a member, do nothing.
- intersection
    - Return the intersection of two sets as a new set.
        - i.e. all elements that are in both sets.
- intersection_update
    - Update a set with the intersection of itself and another.
- isdisjoint
    - Return True if two sets have a null intersection.
- issubset
    - Report whether another set contains this set.
- issuperset
    - Report whether this set contains another set.
- pop
    - Remove and return an arbitrary set element.
        - Raises KeyError if the set is empty.
- remove
    - Remove an element from a set; it must be a member.
        - If the element is not a member, raise a KeyError.
- symmetric_difference
    - Return the symmetric difference of two sets as a new set.
        - i.e. all elements that are in exactly one of the sets.
- symmetric_difference_update
    - Update a set with the symmetric difference of itself and another.
- union
    - Return the union of sets as a new set.
        - i.e. all elements that are in either set.
- update
    - Update a set with the union of itself and others.

[Back to TOC]

Dictionaries

Python provides a mapping object collection in the form of `class dict`. The dict contains key-value pairs, which are mutable.

The keys in a dictionary can be of any arbitrary types. However, the key must be hashable, i.e, values containing lists, dicts, or othermutable types may not be used as keys. If you are planning to use numbers as keys, prefer not to use floating point numbers, since computers store these floating point numbers as approximations.

A dict can be created in different ways:

- a comma-separated list of `key:value` pairs with in curly braces
  - my_info = {"name": "Vinod", "email": "vinod@vinod.co", "city": "Bangalore"}
- using the dict() constructor
  - my_info = dict(name="Vinod", email="vinod@vinod.co", city="Bangalore")

We can also use a list of keys and list of values zipped together in a dict constructor.

```
>>> keys = ["name", "email", "city"]
>>> values = ["Vinod", "vinod@vinod.co", "Bangalore"]
>>> my_info = dict(zip(keys, values))
>>> type(my_info)
<class 'dict'>
>>> my_info
{'name': 'Vinod', 'email': 'vinod@vinod.co', 'city': 'Bangalore'}
>>>
```

Here are the list of functions in dict object:

- clear
  - Remove all items from the dictionary.
- copy
  - Return a shallow copy of the dictionary.
- fromkeys
  - Returns a new dict with keys from iterable and values equal to value.
- get
  - Return the value for key if key is in the dictionary, else default.
  - If default is not given, it defaults to None, so that this method never raises a KeyError.
- items
  - Return a new view of the dictionary's items ((key, value) pairs).
- keys
  - Return a new view of the dictionary's keys.
- pop
  - If key is in the dictionary, remove it and return its value, else return default.
  - If default is not given and key is not in the dictionary, a KeyError is raised.
- popitem
  - Remove and return an arbitrary (key, value) pair from the dictionary.

    - popitem() is useful to destructively iterate over a dictionary, as often used in set algorithms.
    - If the dictionary is empty, calling popitem() raises a KeyError.
- update
    - Update the dictionary with the key/value pairs from other, overwriting existing keys. Return None.
    - update() accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two).
    - If keyword arguments are specified, the dictionary is then updated with those key/value pairs: d.update(red=1, blue=2).
- values
    - Return a new view of the dictionary's values.

[Back to TOC]

Functions

A function is a subroutine, a piece of code that can be reused by supplying different parameters at different times. For example, if we want to get the factorial of a number, the logic for calculating the factorial can be generalized and placed in a function, and when needed, we can supply a number to that function as parameter, and obtain the factorial of the same.

A function is created by using the `def` keyword. The `def` keyword must be followed by the function name and the parenthesized list of formal parameters (optional). The statements that form the body of the function start at the next line, and must be indented.

```
def <function-name> :
    <function-body>
    [return <value>]
```

A return value from a function is optional, and if not used, the return value will be a `None`.

```
>>> def greet():
...     print("Hello, world!")
...
>>>
```

Calling the function `greet()` will simply print a `Hello, world!` message. However, if we want to customize the output, we may have to declare/create one or more parameters. In the following code, `name` is a parameter (or argument) to the function `greet`.

```
>>> def greet(name):
...     print("Hello, %s", name)
...
>>>
```

The parameters and the variables used in the function's body are stored in a local scope. and will not be accessible outside the function's body.

To call the above greet function, we supply a parameter like this:

```
>>> greet("Vinod")
>>> Hello, Vinod
```

And if you don't supply any parameter or more than the defined number of parameters to the function, Python raises a `TypeError`.

```
>>> greet()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: test() missing 1 required positional argument: 'name'
>>> greet("Vinod", "Bangalore")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: test() takes 1 positional argument but 2 were given
>>>
```

The `greet` function does not return a value, so if you assign the function call to a variable, it will be a `None`.

```
>>> message = greet("Vinod")
Hello, Vinod
>>> print(message)
None
>>> type(message)
<class 'NoneType'>
>>>
```

Here is an example of a function that returns a value. Also, the parameter has a default value, so if it's not supplied, will not raise `TypeError`.

```
>>> def factorial(num=5):
...     f = 1
...     for i in range(num, 1, -1):
...         f *= i
...     return f
...
>>> factorial()
120
>>> factorial(12)
479001600
>>>
```

A function is also an object, of `class function`. It can be assigned to another variable, just like any other objects.

```
>>> type(factorial)
<class 'function'>
>>> fact = factorial
>>> fact()
120
>>> fact(22)
1124000727777607680000
>>>
```

A function call itself inside it's body. This is generally called `recursion`.

```
>>> def factorial(n=1):
...     if n>1:
...         return n * factorial(n-1)
...     return 1
...
>>> factorial(5)
120
>>> factorial(22)
1124000727777607680000
>>> factorial(123)
121463043670253296757662432418812958554542170884833823153289181618292
358923621676688311569606126402021707358352212940477825910915704116514
721860295199062616467307339074198149529600000000000000000000000000000
>>>
```

We can optionally use a text (referred as `docstring`) enclosed with in 3 single or double quoutes to document the function. Like the body of the function, this too must be indented.

```
>>> def factorial(n=1):
...     """The factorial function
...        Accepts a parameter and returns the factorial of the same.
...        This function uses the logic n! = n * (n-1)!
...        If there is no parameter supplied, then 1 is assumed"""
...     if n>1:
...         return n * factorial(n-1)
...     return 1
...
>>>
```

And when the command `help(factorial)` is executed, we get the documentation in this format:

```
Help on function factorial in module __main__:

factorial(n=1)
```

```
    The factorial function
    Accepts a parameter and returns the factorial of the same.
    This function uses the logic n! = n * (n-1)!
    If there is no parameter supplied, then 1 is assumed
(END)
```

[Back to TOC]

Function arguments

In Python it is possible to define functions with variable number of arguments. There are three forms, and all of them can be used independent of each other or together.

- Default arguments
- Arbitrary argument list
- Keyword arguments

Default arguments

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. An important point to keep in mind is that non-default arguments should be defined first and then the default arguments.

For example,

```
>>> def add_contact(name, email, city="Bangalore"):
...     # code to add the details to a data store
...     pass
...
>>>
```

can be invoked in the following ways:

```
add_contact("Vinod", "vinod@vinod.co")
add_contact(email="vinod@vinod.co", name="Vinod", city="Bangalore")
add_contact(name="Vinod", city="Bangalore", email="vinod@vinod.co")
```

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```
>>> def add_names(name, names = []):
...     names.append(name)
```

```
...     return names
...
>>>
```

Each time we call the `add_names` function by supplying different name, the list `names` grows. However, unlike mutable objects, the list `names` is assigned with `[]` only once.

```
>>> lst = add_names("Vinod")
>>> add_names("Shyam")
['Vinod', 'Shyam']
>>> add_names("John Doe")
['Vinod', 'Shyam', 'John Doe']
>>> add_names("Jane Doe")
['Vinod', 'Shyam', 'John Doe', 'Jane Doe']
>>>
```

Arbitrary argument list

The least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple. Before the variable number of arguments, zero or more normal arguments may occur.

```
>>> def add_nums(*args):
...     s = 0
...     for n in args:
...         s += n
...     return s
...
>>> add_nums(10, 20, 30, 40)
100
>>> add_nums(10, 20, 30, 40, 50)
150
>>> add_nums()
0
```

In the above function `add_nums`, the parameter `args` has a prefix of `*`, indicating that `args` is a tuple.

```
>>> def add_nums(*args):
...    print("args is of type", type(args))
...
>>> add_nums()
args is of type <class 'tuple'>
>>>
```

We may combine the arbitrary arguments with default arguments. Here is an example:

```
>>> def concat_ws(*args, delimiter=","):
...     return delimiter.join(args)
...
>>> concat_ws("vinod", "shyam", "john", "jane")
'vinod,shyam,john,jane'
>>> concat_ws("vinod", "shyam", "john", "jane", delimiter="-")
'vinod-shyam-john-jane'
>>>
```

In the second usage, we are assigning a `delimiter` and the values before that are considered as the tuple to be assigned to the variable `args`.

Keyword arguments

A keyword argument is simply a `dict`, received as `**kwargs`. Of course, the variable name itself can be anything.

```
>>> def print(**kwargs):
...     print("typeof kwargs is", type(kwargs))
...     print("kwargs is", kwargs)
...
>>>
>>> print(name="Vinod", email="vinod@vinod.co", city="Bangalore")
typeof kwargs is <class 'dict'>
kwargs is {'name': 'Vinod', 'email': 'vinod@vinod.co', 'city':
'Bangalore'}
>>>
```

A common approach to these variety of arguments is to use them in these sequence:

- mandatory arguments
- default arguments
- arbitrary arguments (not much used in practice)
- keyword arguments

An example,

```
>>> def save_data(name, email, city="Bangalore", **additional_info):
...     print("Saving the following data...")
...     print("name : ", name)
...     print("email : ", email)
...     print("city : ", city)
...     for key in additional_info.keys():
...         print(key, ":", additional_info[key])
...     # logic for saving the data here
...
>>> save_data("Vinod", "vinod@vinod.co", website="http://vinod.co",
country="India")
Saving the following data...
name :  Vinod
email :  vinod@vinod.co
```

```
city :  Bangalore
website : http://vinod.co
country : India
>>>
```

[Back to TOC]

Class and objects

An object is an instance of a class. A class defines a data type. For example, the data type `int` is a class and the variable that holds a value of int is an object. All objects have unique ids.

In order to create a data type of our own, we can use the keyword `class` to define one.

```
class Person:
    '''A class to represent information and operation about a
Person'''

    def __init__(self, name="", email="", city=""):
        self.name = name
        self.email = email
        self.city = city
```

As in all constructs of Python, the code that belongs to the class are indented. The class typically contain variables and functions. One special function is `__init__`, which is called the initializer or constructor. This is automatically invoked when an instance of the class is created. The first parameter of any member function of a class represents the reference to the instance itself. This reference can be used to assign properties to the object.

Check this code below:

```
class Test:
    def __init__(self):
        print("Inside the __init__ function, id of self is",
id(self))

t1 = Test()
print("Inside the __main__ function, id of t1 is", id(t1))
```

In the output, you may observe that both the ids are same.

```
$ python3 oop2.py
Inside the __init__ function, id of self is 4331161416
Inside the __main__ function, id of t1 is 4331161416
$
```

The name of the variable `self` can be anything else as well. However, it is customary and convesion to use the same name. If the **__init__** does not have at least one parameter, we will get a *TypeError*.

```
class Test:
    def __init__():
        pass

t1 = Test()
```

results in this error:

```
$ python3 oop2.py
Traceback (most recent call last):
  File "oop2.py", line 6, in <module>
    t1 = Test()
TypeError: __init__() takes 0 positional arguments but 1 was given
$
```

Coming back to the **class Person**, let's examine a part of it.

```
class Person:
    def __init__(self, name="", email="", city=""):
        self.name = name
        self.email = email
        self.city = city

p1 = Person("Vinod", "vinod@vinod.co", "Bangalore")
```

You may observe that the **__init__** function takes 4 parameres but the call **Person("Vinod", "vinod@vinod.co", "Bangalore")** supplies only 3. As mentioned earlier, the first parameter `self` automatically receives the reference of the object, which then will be returned and assigned to the variable `p1`. So, any property assignment to `self` is same as being assigned to `p1`.

So if we were to print the details using p1, we have no problem with it.

```
print("Name", p1.name)
print("Email", p1.email)
print("City", p1.city)
```

No problem output:

```
$ python3 oop1.py
Name Vinod
```

```
Email vinod@vinod.co
City Bangalore
$
```

In general the data members represent crucial information that should not be assigned directly. We hide them behind functions. This concept is known as **Encapsulation**. In other Object Oriented Languages, we usually have a keyword `private` to define such data members, and the language itself disallows the use of private members outside of the class. In Python we use **double underscore (dunder)** members, and Python does not allow to access them using a reference variable.

If we have to change the data members to **dunder** members,

```
class Person:
    '''A class to represent information and operation about a
Person'''

    def __init__(self, name="", email="", city=""):
        self.__name = name
        self.__email = email
        self.__city = city
```

and using an instance access those members,

```
p1 = Person("Vinod", "vinod@vinod.co", "Bangalore")

print("Name", p1.__name)
print("Email", p1.__email)
print("City", p1.__city)
```

would result in this error:

```
$ python3 oop1.py
Traceback (most recent call last):
  File "oop1.py", line 18, in <module>
    print("Name", p1.__name)
AttributeError: 'Person' object has no attribute '__name'
$
```

The proper method to handle this situation is to provide a non-private function.

```
class Person:

    ...

    def print(self):
        print("Name :", self.__name)
        print("Email :", self.__email)
```

```
        print("City :", self.__city)

p1 = Person("Vinod", "vinod@vinod.co", "Bangalore")
p1.print()
```

And we don't have any error!

```
$ python3 oop1.py
Name : Vinod
Email : vinod@vinod.co
City : Bangalore
$
```

[Back to TOC]

Inheritance

Inheritance is a means of `code reusability`. A derived class inherits members from the base class. Syntactically, the derived class declaration takes the base classname as a parameter.

```
class DerivedClassName(BaseClassName):
    # members of the class
```

As an example, consider the `class Employee` has an additional data `salary`, apart from the `name` and `city`. We can reuse the `class Person` as-is and define the new features in the `class Employee`.

```
class Employee(Person):
    def __init__(self, name="", email="", salary=0):
        Person.__init__(self, name, email)
        self.__salary = salary
```

One thing to check out is, in the `__init__` of `class Employee` we are calling the `__init__` of the `class Person`. While this is not necessary, we do it to pass on any of the parameters received throuh the Employee constructor to it's base class constructor. This can also be achieved using the call to the `__init__` using the `super()` method.

```
super().__init__(name, email)
```

Also, the `class Employee` redfines the `print` function. This is often termed as method-overriding. In the absense of this method, call to the print function using an object of Employee class would invoke the one defined in the Person class. Since overriding allows us to provide a

new logic, and if this logic involves invoking the base class's method, then we can do so by invoking it using the `classname`.

```
class Employee(Person):
    ...
    def print(self):
        Person.print(self)
        print("Salary :", self.__salary)
```

This is how we can create an object of `class Employee` and call the `print` function:

```
e1 = Employee("John Doe", "johndoe@mail.com", 45000)
e1.print()
```

And the output:

```
$ python3 oop3.py
Name : John Doe
City : johndoe@mail.com
Salary : 45000
$
```

When the derived class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class. The attribute must be a non-private one.

For example, if `e1.info()` is invoked, then the method (attribute) `info` will be searched in the `clsas Employee`. Since we do not have such function in the class, it will now be looked in the base class - `class Person`. If such a function is not there even in the `class Person`, then Python will raise an exception:

```
AttributeError: 'Employee' object has no attribute 'info'
```

Python supports `multiple inheritance` also. This can be done in the way defined below:

```
class SmartPhone(Phone, Camera):
    ...
    ...
```

In case of multiple inheritance, if we use `super().__init__()` to invoke the base class constructor, Python will call only the first base class's constructor. In the above code sample, it is `Phone.__init__()`. So, it is best to use the explicit constructor call, as listed below:

```
class SmartPhone(Phone, Camera):
    def __init__(self):
        Camera.__init__(self)
        Phone.__init__(self)

    ...
    ...
```

[Back to TOC]

Exception handling

Often when a script is run, we may encounter some errors. Errors are of two types.

1. *Syntax errors* also known as *parse errors*
2. *Exceptions*

Syntax errors are the most common kind of errors that you may get, obviously due to the errors in the grammer of your code.

```
>>>
>>> for i in range(1, 11)
  File "<stdin>", line 1
    for i in range(1, 11)
                        ^
SyntaxError: invalid syntax
>>>
>>>
>>> for i in range(1, 11):
... print(i)
  File "<stdin>", line 2
    print(i)
        ^
IndentationError: expected an indented block
>>>
```

The parser repeats the offending line and displays a ^ pointing at the earliest point in the line where the error was detected. The error is mostly caused by the token preceding ^. File name and line number are printed so you know where to look in case the input came from a script.

Exceptions are the errors that occur during the execution of the code, inspite of the proper syntax. For example, in the following code everything works just fine, until you enter a zero as the second input:

```
n1 = int(input("Enter a number: "))
n2 = int(input("Enter a number: "))
n3 = n1/n2
```

```
print("%d / %d is %d" % (n1, n2, n3))
```

Here are couple of sample outputs

```
$ python3 exceptions2.py
Enter a number: 12
Enter a number: 3
12 / 3 is 4
$
$ python3 exceptions2.py
Enter a number: 17
Enter a number: 3
17 / 3 is 5
$
$ python3 exceptions2.py
Enter a number: 12
Enter a number: 0
Traceback (most recent call last):
  File "exceptions2.py", line 3, in <module>
    n3 = n1/n2
ZeroDivisionError: division by zero
$
```

As you would see, the first two attempts had no problem. However the third attempt when `0` was supplied as the second argument, we do get `ZeroDivisionError`. This is because, Python interpreter cannot divide any number by zero, and whenever it encounters such scenario, it raises an exception by creating an object of suitable class (ZeroDivisionError in this case), and throws it. Since there is no mechanism to catch the thrown object, the code execution will break.

Handling exceptions

The idea of handling exceptions is to ensure that the script execution does not break. This is achieved by enclosing the possible erroneous code in a `try-except` block.

```
try:
  <potential-erroneous-code>
except <Exception-Type>:
  <code-to-handle-the-exception>
```

Here is an example of handling an exception.

Suppose we want to read a number from the user using the `input` function. The return value from this function is of type `str`. So, we must use a conversion function (usually the type constructor, like int or float) to get an inteeger. However, if the user enters a non-numeric value, we do get a **ValueError**.

```
num = int(input("Enter a number: "))
print("cube of %d is %d" %d (num, num*num*num))
```

I tried executing the above script using `12` as an input - No problem. However, when I entered `asdf`, here is what I got:

```
$ python3 exceptions3.py
Enter a number: 12
cube of 12 is 1728
$
$ python3 exceptions3.py
Enter a number: asdf
Traceback (most recent call last):
  File "exceptions3.py", line 1, in <module>
    num = int(input("Enter a number: "))
ValueError: invalid literal for int() with base 10: 'asdf'
$
```

You may guess, the script execution has broken. To avoid the same and take an alternate step, we sorround the code by `try-except` block.

```
try:
    num = int(input("Enter a number: "))
    print("cube of %d is %d" % (num, num*num*num))
except ValueError:
    print("OOPS! you were supposed to enter a number!")
```

Here is the output this time:

```
$ python3 exceptions3.py
Enter a number: 12
cube of 12 is 1728
$
$ python3 exceptions3.py
Enter a number: asdf
OOPS! you were supposed to enter a number!
$
```

And we can nest it with other constructs like loops too.

```
while True:
    try:
        num = int(input("Enter a number: "))
        break
    except ValueError:
        print("OOPS! Not a number! Try again.")
```

```
print("cube of %d is %d" % (num, num*num*num))
```

.. and the output!

```
$ python3 exceptions3.py
Enter a number: asd
OOPS! Not a number! Try again.
Enter a number: qwe
OOPS! Not a number! Try again.
Enter a number: 33
cube of 33 is 35937
$
```

There are handful excetion classes are defined in Python. Here is the complete hierarchy of those classes:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
           +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
```

```
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

[Back to TOC]

Raising exceptions

We can use the `raise` keyword to report an erroneous situation to Python. This way a function can communicate to the caller of the function that there was a runtime error.

For example, given this function:

```
def factorial(num=1):
    f = 1
    for n in range(num, 1, -1):
        f *= n
    return f
```

When we call the function with a positive number,

```
num = 5
fact = factorial(num)
```

```
print("factorial of %d is %d" % (num, fact))
```

we do get the output properly:

```
factorial of 5 is 120
```

However, when we supply a negative number, the function returns 1.

```
factorial of -2 is 1
```

If we want to report to the caller of the function that negative numbers are not allowed, `raise` comes to our rescue.

Here is the modified version of the function factorial:

```
def factorial(num=1):
    if num<0:
        raise ValueError("Negative numbers are not allowed!")
    f = 1
    for n in range(num, 1, -1):
        f *= n
    return f
```

and the output when we supply `-2`:

```
$ python exceptions4.py
Traceback (most recent call last):
  File "exceptions4.py", line 10, in <module>
    fact = factorial(num)
  File "exceptions4.py", line 3, in factorial
    raise ValueError("Negative numbers are not allowed!")
ValueError: Negative numbers are not allowed!
$
```

At times, we may not be able to use the built-in error classes. For example, if a function receives a variable representing `age`, it make sense to raise `InvalidAgeError` instead of `ValueError`. However, there is no such class called `InvalidAgeError`. To create one such class, we can create a derived class from one of the built-in error classes.

```
class InvalidAgeError(Exception):
    def __init__(self, message="Invalid age. Must be >0 and <120"):
        self.message = message

    def __str__(self):
        return self.message
```

Notice that we are overriding dunder method `__str__` which simply represents the textual version of the exception object. This is automatically called when the exception is raised.

```
class Person(object):
    def __init__(self, name="", age=20):
        if(age<0 or age>120):
            raise InvalidAgeError()

        self.__name = name
        self.__age = age

    def info(self):
        print("Name = %s" % self.__name)
        print("Age = %d" % self.__age)
```

In the constructor function of the `class Person`, we check if the user passes a proper value for the `age` or not. If the user supplies a wrong value, we create an object of `InvalidAgeException` and raise the same. The exception is received at the code that invoked the constructor.

```
p1 = Person("Jane Doe", -12)
p1.info()
```

```
Traceback (most recent call last):
  File "exceptions5.py", line 24, in <module>
    p1 = Person("Jane Doe", -12)
  File "exceptions5.py", line 11, in __init__
    raise InvalidAgeError()
__main__.InvalidAgeError: Invalid age. Must be >0 and <120
```

The calling code may be enclosed in a `try-except` block to handle the erroneous situation:

```
try:
    p1 = Person("Jane Doe", -12)
    p1.info()
except Exception as ex:
    print("There was an error while creating the Person object: ")
    print(ex)
```

```
$ python exceptions5.py
There was an error while creating the Person object:
Invalid age. Must be >0 and <120
$
```

[Back to TOC]

Regular expressions

Regular expression is a powerful language feature in Python for matching text patterns and has become a standard for of searching, replacing, and parsing text with complex patterns of characters.

Using a regular expression, many complex lines of coding can be reduced to one liners. Because of its power, often regular expressions are used in text editors, compilers and interpreters.

General uses of regular expressions include:

- Search a part of the string in a bigger string (search and match)
- Replace parts of the string (sub)
- Break a string into small pieces (split)

The `re` is a built in module in Python, and it has to be imported, before it can be used:

```
import re
```

Following are special characters and notations used with regular expressions.

- Alternative `|`
- Grouping `[]`
- Quantification `?*+{n,m}`
- Anchors `^$`
- Meta characters `.[][-][^]`
- Escape character classes `\d\s\w`

Alternative

```
"cat|dog" # "cat" or "dog"
"vinod|kumar" # "vinod" or "kumar"
```

Grouping

```
"(G|M)ary" # "Gary" or "Mary"
"Vi(nod|j(ay|et))" # "Vinod" or "Vijay" or "Vijet"
```

Following special characters represent multiplicity of the preceding element:

- ? `zero or one of the preceding element`
- * `zero or more of the preceding element`
- + `one or more of the preceding element`
- {m,n} `m to n times of the preceding element`

```
"vinodh?" # "vinod" or "vinodh"
"colou?r" # "color" or "colour"
```

```
"fo*ot" # "fot" or "foot" or "fooot" or "fooooooooot" etc
"120*5" # "1205" or "12005" or "1200005" etc

"ama+n" # "aman" or "amaan" or "amaaan" or "amaaaan" etc
"12+34" # "1234" or "12234" or "12222222234" etc

"go{2,4}gle" # google or "gooogle" or "goooogle"
"9{3}" # "999"
"z{2,}" # "zz" or "zzz" or "zzzz" or "zzzzz" ....
```

Anchors

- ^ matches the starting position with in the string
- $ matches the ending position with in the string

```
"^obje" # "object" or "objective" or "object oriented" etc
"^2017" # "2017" or "2017/01/20"

"gram$" # "program" or "telegram" or "kilogram" etc
"2017$" # "2017" or "2010-2017" or "20/01/2017" etc
```

Meta-characters

- . matches any single character
- [] matches a single character that is contained with in the brackets
- [-] matches a single character that is contained within the brackets and the specified range
- [^] matches a single character that is not contained within the brackets

```
"cat." # "cat" or "cats" or "caty" or "cate" or any 4 letter words
that start with "cat"
"12.4" # "124" or "1234" or "12a3" etc

"[abc]" # "a" or "b" or "c"
"[aeiou]" # any vowel
"[1234567890]" # any digit

"[a-d]" # "a" or "b" or "c" or "d"
"[a-zA-Z]" # all alphabets lowercase or uppercase
"[0-9]" # all digits

"[^aeiouAEIOU]" # any non-vowel characters
"[^0-9]" # any non-digit
"[^abc]" # any character, but not "a", "b", or "c"
```

Escape character classes

The Escape character classes specify a group of characters to match in a string

- \d - Matches a decimal digit [0-9]
- \D - Matches non digits
- \s - Matches a single white space character [\t-tab, \n-newline, \r-return, \v-space, \f-form]
- \S - Matches any non-white space character
- \w - Matches alphanumeric character class ([a-zA-Z0-9_])
- \W - Matches non-alphanumeric character class ([^a-zA-Z0-9_])
- \w+ - Matches one or more words / characters
- \b - Matches word boundaries when outside brackets. atches backspace when inside brackets
- \B - Matches nonword boundaries
- \A - Matches beginning of string
- \z - Matches end of string

The `re` module

provides the following useful functions:

- compile
  - Compile a regular expression pattern into a regular expression object, which can be used for matching using its match() and search() methods. The expression's behaviour can be modified by specifying a flag value.
- findall
  - Return all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order found. If one or more groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group. Empty matches are included in the result unless they touch the beginning of another match.
- finditer
  - Return an iterator yielding MatchObject instances over all non-overlapping matches for the RE pattern in string. The string is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result unless they touch the beginning of another match.
- match
  - If zero or more characters at the beginning of string match the regular expression pattern, return a corresponding MatchObject instance. Return None if the string does not match the pattern; note that this is different from a zero-length match. Note that even in MULTILINE mode, re.match() will only match at the beginning of the string and not at the beginning of each line. If you want to locate a match anywhere in string, use search() instead.
- purge
  - Clear the regular expression cache.
- search
  - Scan through string looking for the first location where the regular expression pattern produces a match, and return a corresponding MatchObject instance. Return None if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

- split Split string by the occurrences of pattern. If capturing parentheses are used in pattern, then the text of all groups in the pattern are also returned as part of the resulting list.
- sub
  - Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement string. If the pattern isn't found, string is returned unchanged. The replacement string can be an actual string or a function that returns string; if it is a string, any backslash escapes in it are processed. That is, \n is converted to a single newline character, \r is converted to a carriage return, and so forth. Unknown escapes such as \j are left alone.

Try these examples:

Search the given tokens in a sentence.

```
import re
sentence = "Here is the First Regular Expression Program"
tokens = [ "first", "hello", "world", "program"]

for token in tokens:
    print("Searching for '%s' in '%s' ->" % (token, sentence),
end="")
    if re.search(token, sentence, re.I):
        print("Yes! found it.")
    else:
        print("Not found :-(")
```

Split the words from a sentence.

```
>>> import re
>>> sentence = "Python is an awesome programming language"
>>> pattern = r"\W+"
>>> search = re.compile(pattern)
>>> search.split(sentence)
['Python', 'is', 'an', 'awesome', 'programming', 'language']
>>>
```

String substitution examples

```
import re

DOB = "20-01-1991 # This is Date of Birth"

# Delete Python-style comments
Birth = re.sub (r'#.*$', "", DOB)
print("Date of Birth : ", Birth)

# Remove anything other than digits
Birth1 = re.sub (r'\D', "", Birth)
```

```
print("Before substituting DOB : ", Birth1)

# Substituting the '-' with '.(dot)'
New=re.sub (r'\W',".",Birth)
print("After substituting DOB: ", New)
```

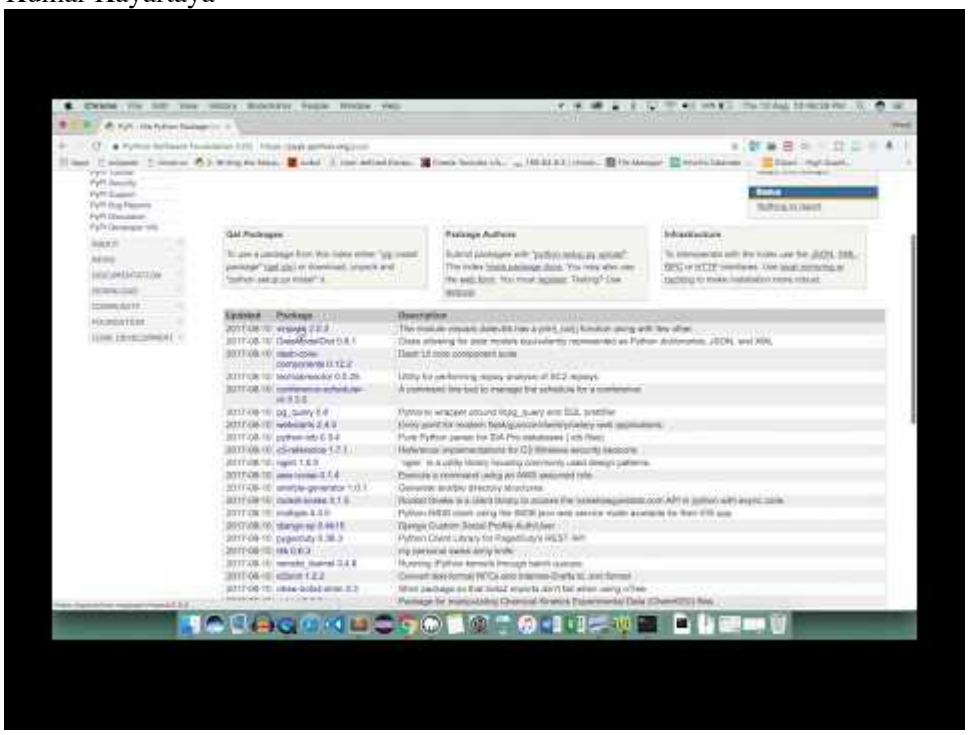
Examples for re.findall() function

```
import re

line = "Python is a widely used high-level scripting language for
general purpose programming, created by Guido van Rossum and first
released in 1991"

print("Find all words starts with p")
print(re.findall(r"\bp[\w]*",line, re.I))
print()

print("Find all five characters long words")
print(re.findall(r"\b\w{5}\b", line))
print()

# Find all four, six characters long words
print("find all 4, 6 char long words")
print(re.findall(r"\b\w{4,6}\b", line))
print()

# Find all words which are at least 13 characters long
print("Find all words with 13 char")
print(re.findall(r"\b\w{13,}\b", line))
print()
```

[Back to TOC]

Modules and packages

Watch the video : Building and deploying a Python package



While uploading your package to the Pypi repository, the pip3 (or pip) command also looks for a file called `.pypirc` in your home directory. It contains the login credentials required to publish new content.

filename: **/Users/vinodkumar/.pypirc**

```
[server-login]
username:kayartaya
password:BIGSECRET
```

Files used in the demo:

filename: **setup.py**

```
import setuptools

setuptools.setup(
    name = "vinpack",
    version = "2.0.3",
    packages = ["vinpack"],
    package_dir = {"vinpack": "./vinpack"},
    author="Vinod",
    author_email="vinod@vinod.co",
    maintainer="Vinod",
    maintainer_email="vinod@vinod.co",
    url="http://vinod.co",
    description="The module vinpack.dateutils has a print_cal()
function along with few other.",
    long_description="""The package vinpack has only one module
'dateutils' with the following members:
```

```
is_leap(year)
    -> returns True/False for a leap year/ non-leap year

max_days(month, year)
    -> returns the maximum number of days in the given month of the
year

print_cal(month=None, year=None)
    -> prints the calendar for the given month and year
    -> prints the calendar for current month/year if not
specified""",
    license="MIT"
    )
```

filename: **dateutils.py**

```
import datetime, calendar

def line():
    print("-"*20)

def is_leap(year):
    return year%400==0 or (year%4==0 and year%100!=0)

def max_days(month, year):
    if month==2:
        return 29 if is_leap(year) else 28
    elif month in (4, 6, 9, 11):
        return 30
    else:
        return 31

def calendar(month=None, year=None):
    '''This function prints the calendar for the input parameters -
Month and the Year

    The month and year are not validated.
    Make sure to input 1-12 for Month and a positive number for year

    If the month/year is not supplied, then it will default to the
current month and year

    Author: Vinod <vinod@vinod.co>
    '''

    today = datetime.date.today()
    if month == None:
        month = today.month

    if year == None:
        year = today.year

    dt = datetime.date(year, month, 1)
```

```
    line()
    header = "{0} - {1}".format(calendar.month_name[month], year)
    print(header.center(20))
    print("Su Mo Tu We Th Fr Sa")
    line()

    days = max_days(month, year)
    weekday = dt.weekday() + 1

    if weekday!=7: print(" " * (weekday * 3), end="")

    for d in range(1, days+1):
        print("%2d " % d, end="")
        if (weekday+d)%7==0: print()

    print()
    line()


if __name__ == '__main__':
    calendar()
```

Command to install the package on our local `Python`

```
pip3 install .
```

Command to install the package to the `http://pypi.python.org` repository

```
python3 setup.py sdist upload
```

Before executing the above command, make sure you have created an account with `http://pypi.python.org` and created a `.pypirc` file in the home directory.

[Back to TOC]

List tricks

The list:

```
names = ["anu", "vinod kumar kayartaya", "khushi", "karishma k"]
```

Sorting a list with out mutating:

```
sorted(names)
```

Sorting the list by length:

```
sorted(names, key=len)
```

Sorting the list by length in reverse order:

```
sorted(names, key=len, reverse=True)
```

reverse a list

```
lst = [12, 23, 34, 45, 56, 67, 78, 89, 90]
rev_lst = lst[::-1]
```

Same can be applied on the strings as well:

```
name = 'Vinod Kumar'
rev_name = name[::-1]
```

List of dicts:

```
data = [{"name": "Vinod", "age": 44},
{"name": "Anu", "age": 42},
{"name": "Khushi", "age": 15},
{"name": "Karishma", "age": 7},
]
```

Sorting the list of dicts by name:

```
sorted(data, key=lambda d:d["name"])
sorted(data, key=lambda d:d["age"])
```

... in reverse order:

```
sorted(data, key=lambda d:d["name"], reverse=True)
sorted(data, key=lambda d:d["age"], reverse=True)
```

List Comprehensions:

```
# convert list elements to their squares
nums = [12, 33, 40, 20]
squares = [ n*n for n in nums]
```

```
# convert list elements into uppercase
names = ["Vinod", "Anu", "Khushi", "Karishma"]
names_upr = [ s.upper() for s in names ]

# remove non-numeric values from a list
lst = [10, 20, 'asdf', 'xyz', 33, 45, 20, 'abc']
lst_nums = [n for n in lst if type(n) in (int, float)]

# get even numbers from a list
lst = [12, 23, 34, 45, 56, 67, 78, 89, 90]
lst_evens = [n for n in lst if n%2==0]
```

Using dict in a formatted string:

```
for d in data:
   print("%(name)s is %(age)d years old" % d)
```