PYTHON DATA TYPES - BASICS

Gives an introduction to the basic types in Python

DECEMBER 20, 2017

Table of Contents

1. Python Conceptual Hierarchy

Python programs can be decomposed into modules, statements, expressions and objects as follows:

1. Programs are composed of modules

2. Modules contain statements

3. Statements contain expressions

4. Expressions _create_ and _process_ objects.

1.1. Need of Built-in Types

1. Built-in objects make programs easy to write

2. Built-in objects are components of extensions

3. Built-in objects are often more efficient than custom data structures

4. Built-in objects are a standard part of the language

2. Python's Core Data Types

Object Type	Example literal / creation
Numbers	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
Strings	'spam', "Bob's", b'a\x01c', u'sp\xc4m'
Lists	\[1, \[2, 'three'], 4.5], list(range(10))
Dictionaries	{ 'food' : 'spam', 'taste' : 'yum' }, dict(hours=10)
Tuples	(1, 'spam', 4, 'U'), tuple('spam'), namedtuple
Files	open('eggs.txt'), open(r'C:\ham.bin', 'wb')
Sets	set('abc'), {'a', 'b', 'c'}
Other core types	Booleans, types, None
Program Unit types	Functions, modules, classes
Implementation related types	Compiled code, stack tracebacks

Table 1: Built-in objects' preview

Once an object is created, its operation set are bound for all time.

i.e. Python is *dynamically typed*, a model that keeps track of types automatically instead of requiring declaration code, but it is also a *strongly typed*, a constraint that means you can perform on an object only operations that are valid for its type.

Python variables never need to be declared ahead of time. A variable is created when a value is assigned to it, may be assigned any type of object, and is replaced with its value when it shows up in an expression.

2.1. Numbers

Includes:

- *integers* that have no fractional part
- *floating-point* numbers that has fractional part
- *complex* numbers with imaginary parts
- *decimals* with fixed precision
- *rational* with numerator and denominator
- full-featured *sets*

Numbers in Python support the normal mathematical operations like:

- + performs addition
- * performs multiplication
- ** performs exponentiation

Floating point numbers behaves differently in version before Python 2.7 and 3.1 and the after those versions (3.X): -

In Pythons < 2.7 and 3.1

```
>>>
>>> 3.1415 * 2          # repr: as code. i.e result is shown with full
precision.
6.2830000000000004
>>> print(3.1415 * 2)   # str: user friendly. Result is shown in a user
friendly way
6.283
>>>
```

Code Snippet 1: Behaviour of floating point numbers in < 2.7 & 3.X and later versions

In Pythons >=2.7 and 3.1

```
>>>
>>> 3.1415 * 2            # repr: as code
6.283
>>>
```

Code Snippet 2: Behaviour of floating point numbers in >= 2.7 and 3.1

2.2. Strings

Strings are used to record both ***textual information*** aa well as ***arbitrary collections of bytes*** (such as image file's contents).

Strings are an example of *sequence* in Python - a positionally ordered collection of other objects. Sequences maintain a left-to-right order among the items they contain: their items are stored and fetched by their relative positions.

Strings are sequences of one-character strings.

Other sequence types include lists, tuples etc.

Strings are ***immutable***.

2.2.1. Sequence Operations:

1. Strings supports operations that assume a positional ordering among items.
2. In Python, *backward indexing* is also possible - positive indexes count from the left and negative indexes count back from right.
   Formally, negative index is simply added to the length of the string sequence.
   Sample code:

```
>>>
>>> S = 'Spam'  # Make a 4-character string and assign it to a name
>>> len(S)
4
>>> S[0] # First item in S, indexed by zero based position
'S'
>>> S[1] # Second item from left
'p'
>>> S[-1]  # Last item in S
'm'
>>> S[-2]    # Second-to-last item from the end.
'a'
>>> # Negative index == Add negative index to the string length.
>>> S[len(S)-2]  #== S[-2]
'a'
>>>
```

Code Snippet 3: Indexing in Sequence operations

3. Sequences support a more general form of indexing known as slicing, which is a way to extract an entire-section (slice) in a single step.
   Slicing can be considered to be a way of extracting an entire column from a string in a single step.
   General form ***X[I:J]***, means give me everything in X from offset I up to, but not including, offset J. The result is returned in a new object.

In a slice, the left bound defaults to zero, and the right bound defaults to the length of the sequence to be sliced.

Sample Code:

```
>>>
>>> S
'Spam'
>>> S[1:3]
'pa'
>>> S[1:]   # Everything past the first i.e. 1:len(S)
'pam'
>>> S       # S itself hasn't changed
'Spam'
>>> S[0:3]  # Everything but the last
'Spa'
>>> S[:3]   # Same as S[0:3]
'Spa'
>>> S[:-1]  # Everything but the last, S[0:-1]
'Spa'
>>> S[:]    # All of S as a top-level copy S[0:len(S)]
'Spam'
>>>
```

Code Snippet 4: Slicing of a string

4. As sequences, Strings also support *concatenation* with plus sign (joining two strings to form a new string) and *repetition* (making a new string by repeating another)

   Sample Code:

```
>>>
>>> S
'Spam'
>>> S + 'xyz'     # Concatenation
'Spamxyz'
>>> S             # Unchanged
'Spam'
>>> S * 8         # Repetition
'SpamSpamSpamSpamSpamSpamSpamSpam'
>>> S
'Spam'
>>>
```

Code Snippet 5: Concatenation in Strings

2.2.2. Immutability:

Strings are *immutable* in Python - every string operation is defined to produce a new string as its result, that cannot be changed in place after they are created.

i.e. Valued of immutable objects can never be overwritten. But you can always build a new one and assign it to the same name. This is because, Python cleans up old objects on the go.

```
>>> S = 'Spam'
>>> S
'Spam'
>>> S[0]
'S'
>>> S[0] = 'z'      # Immutable objects cannot be changed.
    ....error text...
    S[0] = 'z'      # Immutable objects cannot be changed.
TypeError: 'str' object does not support item assignment
>>> S = 'z' + S[1:] # Can make new objects
>>> S
'zpam'
>>>
```

Code Snippet 6: Immutability of Strings

Every object in Python is classified as Immutable (unchangeable) or not.

The core types *numbers*, *strings* and *tuples* are immutable, while *lists*, *dictionaries* and *sets* are not.

Immutability can be used to guarantee that an object remains constant throughout the program; mutable objects' values can be changed at any time and place.

The text based data can be changed *in-place*, in either of two ways:

1. Expand it into a *list* of individual characters and join it back together with nothing in between

```
>>>
>>> S = 'shrubberry'    # Expand to a list: [...]
>>> L = list(S)
>>> L
['s', 'h', 'r', 'u', 'b', 'b', 'e', 'r', 'r', 'y']
>>> L[1] = 'c'      # Change it in place
>>> ''.join(L)      # Join with empty delimiter - joining all the items in
the list.
'scrubberry'
>>> L
['s', 'c', 'r', 'u', 'b', 'b', 'e', 'r', 'r', 'y']
>>> '@'.join(L)    # Try joining with another delimiter e.g: @
's@c@r@u@b@b@e@r@r@y'
>>>
```

Code Snippet 7: Expanding string to a list for in-place modification

2. Use *bytearray* type available in Pythons 2.6, 3.0 and later

```
>>>
>>> B = bytearray(b'spam')   # A bytes/list hybrid (ahead)
>>> B.extend(b'eggs')        # b needed in 3.X, not in 2.X
>>> B
bytearray(b'spameggs')
>>> B.decode()             # Translate to normal string
'spameggs'
>>>
```

Code Snippet 8: Using bytearray for in-place modification of Strings

The *bytearray* supports in-place changes for text, but only for text whose characters are all at most 8-bits wide (e.g.: ASCII).

All other strings are still immutable - *bytearray* is a distinct hybrid of immutable *bytes* strings (whose b'...' syntax is required depending on Python version) and mutable *lists* (coded and displayed in []).

2.2.3. Type-Specific Methods:

In addition to generic sequence operations, strings also have operations all their own, available as methods.

Sample code to show some string methods and its usage:

```
>>>
>>> S = 'Spam'
>>> # To find the offset of a substring is S
>>> S.find('pa')
1
>>> S.find('pq')      # If the substring is not present it will return -1
-1
>>> S
'Spam'
>>> # To replace occurrences of a substring with another
>>> S.replace( 'pa', 'XYZ')
'SXYZm'
>>> S
'Spam'
>>>
```

Code Snippet 9: Type-Specific Methods - Strings

Other methods split a string to substrings on a delimiter, perform case conversions, test the content of the string and strip whitespace characters off the ends of the string:

```
>>>
>>> line = 'aaa, bbbb, ccccc, dd'
>>> line.split(',') # Split on a delimiter into a list of substrings
['aaa', ' bbbb', ' ccccc', ' dd']
>>> line
'aaa, bbbb, ccccc, dd'
>>> line.split(', ')
['aaa', 'bbbb', 'ccccc', 'dd']
>>>
>>> S = 'Spam'
>>> S.upper()          # Upper-case conversion
'SPAM'
>>> S.lower()          # Lower-case conversion
'spam'
>>> S                  # No change in the original string
'Spam'
>>> S.isalpha()        # Content test: isalpha, isdigit etc.
True
>>> S.isdigit()
False
>>> X = 'a2'
>>> X.isdigit()
False
>>> X = '2'
>>> X.isdigit()
True
>>>
>>> line = 'aaa,bbb,ccc,dd\n'
>>> line
'aaa,bbb,ccc,dd\n'
>>> line.rstrip()              # Remove whitespace
'aaa,bbb,ccc,dd'
>>> line
'aaa,bbb,ccc,dd\n'
>>> line.rstrip().split(',')       # Combining both the operations
['aaa', 'bbb', 'ccc', 'dd']
>>>
```

Code Snippet 10: Other String type-specific methods

Strings also support an advanced substitution operation known as *formatting*, available as both an expression and a string method call.

```
>>>
>>> # Formatting Expression
>>> '%s, eggs, and %s' % ('spam', 'SPAM!')
'spam, eggs, and SPAM!'
>>>
>>> # Formatting Method ( 2.6+, 3.0+)
>>> '{0}, eggs, and {1}'.format('spam', 'SPAM!')
'spam, eggs, and SPAM!'
>>>
>>> # Numbers optional (2.7+, 3.1+)
>>> '{}, eggs, and {}'.format('spam', 'SPAM!')
'spam, eggs, and SPAM!'
>>>
>>> ### Formatting to generate numeric reports
>>> '{:,.2f}'.format(296888.2567)    # Separators, decimal digits
'296,888.26'
>>> '%.2f | %+05d' % ( 3.14159, -42)     # Digits, padding and signs
'3.14 | -0042'
>>>
```

Code Snippet 11: Formatting Methods of String

2.2.4. Getting Help

The built-in ***dir*** function returns a list of all the attributes available to for any object passed to it.

```
>>>
>>> S
'Spam'
>>> dir(S)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',
'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>>
```

Code Snippet 12: Usage of dir *function*

The names with *double underscores* represent the implementation of the string object and are available to support customization.

For e.g.: the ***add*** method is what really performs concatenation. Python maps plus operation to this internally, though one should not use this form.

```
>>>
>>> S + 'NI!'
'SpamNI!'
>>> S.__add__('NI!')
'SpamNI!'
>>>
```

Code Snippet 13: Internal function to perform addition

Generally, leading and trailing double underscores is the naming pattern Python uses for implementation details. The names without the underscores in this list are the callable methods on the String objects.

The ```dir``` function simply gives the name of methods. To ask what they do, use the help function.

```
>>>
>>> help(S.replace)
Help on built-in function replace:

replace(...) method of builtins.str instance
    S.replace(old, new[, count]) -> str

    Return a copy of S with all occurrences of substring
    old replaced by new.  If the optional argument count is
    given, only the first count occurrences are replaced.

>>> help(S)
No Python documentation found for 'Spam'.
Use help() to get the interactive help utility.
Use help(str) for help on the str class.

>>>
```

Code Snippet 14: Usage of help funtion

PyDoc - a tool for extracting documentation from objects.

Both dir and help accepts either a real object (like S) or the name of a data type (like str, list and dict) as arguments. The latter form returns the same list for dir, but shows full type details for help and allows to ask about a specific method via type name.

```
>>>
>>> help(str)
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |      ####
 |      # Other Functions removed
 |      ####
 |
 |
 |  __str__(self, /)
 |      Return str(self).
 |
 |  capitalize(...)
 |      S.capitalize() -> str
 |
 |      Return a capitalized version of S, i.e. make the first character
 |      have upper case and the rest lower case.
 |
 |  casefold(...)
 |      S.casefold() -> str
 |
 |      Return a version of S suitable for caseless comparisons.
 |
 |              ####
 |              # Other Functions removed
 |              ####
 |
 |  zfill(...)
 |      S.zfill(width) -> str
 |
 |      Pad a numeric string S with zeros on the left, to fill a field
 |      of the specified width. The string S is never truncated.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  maketrans(x, y=None, z=None, /)
 |      Return a translation table usable for str.translate().
 |
 |      If there is only one argument, it must be a dictionary mapping
Unicode
 |      ordinals (integers) or characters to Unicode ordinals, strings or
None.
 |      Character keys will be then converted to ordinals.
```

Code Snippet 15: Usage of help by specifying data type name

2.2.5. Other Ways to Code Strings

Special characters can be represented as backslash escape sequences, which Python displays in \xNN hexadecimal escape notation, unless they represent printable characters.

```
>>>
>>> S = 'A\nB\tC'          # \n is end-of-line, \t is tab
>>> S
'A\nB\tC'
>>> len(S)               # Each stands for just one character
5
>>> P = 'k\o'
>>> P
'k\\o'
>>> P = 'k\\o'
>>> P
'k\\o'
>>> ord('\n')            # \n is a byte with binary value 10 in ASCII
10
>>>
>>> S = 'A\0B\0C'          # \0, a binary zero byte, does not terminate
string
>>> len(S)
5
>>> S                    # Non-printables are displayed as \xNN hex escapes
'A\x00B\x00C'
>>>
```

Code Snippet 16: Other ways to code string

Python allows strings to be enclosed in single or double quote characters - they mean the same thing, but allow the other type of quote to be embedded with an escape.

It also allows multiline string literals enclosed in triple quotes (single or double) - when this form is used, all the lines are concatenated together, and end-of-lone characters are added where line breaks appear.

```
>>> msg = """
aaaaaaaaaaaa
bb'''bbbbb""yy'yyyy
cccccccccc
"""
>>> msg
'\naaaaaaaaaaaa\nbb\'\'\'bbbbb""yy\'yyyy\ncccccccccc\n'
>>> nextMsg = """aaa
bb'bbbbbb"yy'uuuuuu"""
>>> nextMsg
'aaa\nbb\'bbbbbb"yy\'uuuuuu'
>>> changeQuote = '''aaa
bb"bbbbbb'yy"uuuuuu'''
>>> changeQuote
'aaa\nbb"bbbbbb\'yy"uuuuuu'
>>>
```

Code Snippet 17: Multi-line string literals

Python also supports a ***raw*** string literal that turns off the backslash mechanism. Such literals start with the letter ```r``` and are useful for strings like directory paths on Windows.

```
>>>
>>> pt = r'C:\text\new'
>>> pt
'C:\\text\\new'
>>> pt = r'C:/text/new'
>>> pt
'C:/text/new'
>>> pt = 'C:/text/new'
>>> pt
'C:/text/new'
>>> pt = 'C:\text\new'
>>> pt
'C:\text\new'
>>>
```

Code Snippet 18: raw String literal

2.2.6. Unicode Strings

Python's Strings supports full Unicode required for processing text in internationalized character sets.

In Python 3.X:-

- normal ```str``` string handles Unicode text (including ASCII)
- a distinct ```bytes``` string type represents raw byte values (including media and encoded text)
- for 2.X compatibility, 2.X Unicode literals are supported in 3.3 and later (they are treated the same as normal 3.X str strings)

```
>>>
>>> 'sp\xc4m'        # 3.X: normal str strings are Unicode texts
'spÄm'
>>> b'a\x01c'        # bytes strings are byte-based data
b'a\x01c'
>>> u'sp\u00c4m'     # The 2.X Unicode literal works in 3.3+: just str
'spÄm'
>>>
```

Code Snippet 19: String handling in 3.X

In Python 2.X:-

- the normal ```str``` string handles both 8-bit character strings (including ASCII text) and raw byte values
- a distinct ```unicode``` string type represents Unicode text
- For 3.X compatibility, 3.X bytes literals are supported in 2.6 and later (they are treated the same as normal 2.X ```str``` strings)

```
>>>
>>> print ( u'sp\xc4m' )     #2.X: Unicode strings as distinct types
spÄm
>>> 'a\x01c'                  # Normal str strings contain byte-based
text/data
'a\x01c'
>>> b'a\x01c'                 # The 3.X bytes literal works in 2.6+: just
str
b'a\x01c'
>>>
```

Code Snippet 20: String handling in 2.X

In both 2.X and 3.x, non-unicode strings are sequences of 8-bit bytes that print with ASCII characters when possible, and Unicode strings are sequences of Unicode code points - identifying numbers for characters, which do not necessarily map to single bytes when encoded to files or stored in memory. In fact, the notion of bytes doesn't apply to Unicode: some encodings include character code points too large for a byte, and even simple 7-bit ASCII text is not stored one byte per character under some encodings and memory storage schemes.

```
>>>
>>> 'spam'                  # Characters may be 1, 2 or 4 bytes in memory
'spam'
>>> 'spam'.encode( 'utf8' ) # Encoded to 4 bytes in UTF-8 files
b'spam'
>>> 'spam'.encode( 'utf16' )    # But encoded to 10 bytes in UTF-16
b'\xff\xfes\x00p\x00a\x00m\x00'
>>>
```

Code Snippet 21: Behaviour of non-unicode strings in 2.X and 3.X

Both 3.X and 2.X also supports the following:

- the ```bytearray``` string type,which is essentially a ```bytes``` string (a ```str``` in 2.X) that supports most of the list object's in-place mutable change operations.
- Support coding the non-ASCII characters with ***x*** hexadecimal and shot ***u*** and long ***U*** Unicode escapes, as well as file-wide encodings declared in program source files.

```
>>>
>>> # Sample code showing non-ASCII character coded in 3 ways in 3.X
>>> 'sp\xc4\u00c4\U000000c4m'
'spÄÄÄm'
>>> # In 2.X
>>> print(u'sp\xc4\u00c4\U000000c4m')
spÄÄÄm
>>>
```

Code Snippet 22: Behaviour of non-ASCII characters in 2.X and 3.X

What these values mean and how they are used differs between ***text*** strings and ***byte*** strings.

- text strings: which are the normal string in 3.X and Unicode in 2.X
- byte strings: which are bytes in 3.X and the normal string in 2.X

All these escapes can be used to embed actual Unicode code-point ordinal value integers in text strings.

By contrast, byte strings use only \x hexadecimal escapes to embed the encoded form of text, not its decoded code point values - encoded bytes are same as code points, only for some encodings and characters.

```
>>>
>>> '\u00A3', '\u00A3'.encode('latin1'), b'\xA3'.decode('latin1')
('£', b'\xa3', '£')
>>>
```

Code Snippet 23: text with escape values

Difference: -

- Python 2.X allows its normal and Unicode strings to be mixed in expressions as long as the normal string is all ASCII
- Python 3.X never allows its normal and byte strings to mix without explicit conversion.

```
u'x' + b'y'          # Works in 2.X, where b is optional and ignored
                     # Fails in 3.3, where u is optional and ignored
u'x' + 'y'           # Works in 2.X: u'xy' and Works in 3.3: 'xy'
'x' + b'y'.decode()  # Works in 3.X if decode bytes to str: 'xy'
'x'.encode() + b'y'  # Works in 3.X if encode str to bytes: b'xy'
```

Code Snippet 24: Difference between 3.X and 2.X string handling

Apart from these string types, Unicode processing mostly reduces to transferring text data to and from files - text is encoded to bytes when stored in a file, and decoded into characters (a.k.a. code points) when read back into memory. Once it is loaded, we usually process text as strings in decoded form only.

Because of this model, though, ***files are also content specific in 3.X***:

- text files implement named encodings and accept & return ```str``` strings
- binary files deal in ```bytes``` strings for raw binary data

In 2.X, normal files' content is ```str``` bytes, and a special ```codecs``` module handles Unicode and represents content with the ```unicode``` type.

2.2.7. Pattern Matching

None of the string objects in Python own methods to support pattern-based text processing.

To do pattern matching in Python, import a module ```re```. This module has analogous calls for searching, splitting and replacement.

An example to search for a substring that begins with the world "Hello", followed by zero or more tabs or spaces, followed by arbitrary characters to be saved as a matched group, terminated by the word "world".

```
>>>
>>> import re
>>> match = re.match( 'Hello[ \t]*(.*)world', 'Hello    Python world' )
>>> match.group( 1 )
'Python '
>>> match.groups()
('Python ',)
>>> match = re.match( 'Hello[\t]*(.*)world', 'Hello    Python world' )
>>> match.group( 1 )
'    Python '
>>> match = re.match( 'Hello[\t]*(.*)world', 'Hello~~~~~Python world' )
>>> match.group( 1 )
'~~~~~Python '
>>> match = re.match( 'Hello[\t]*(.*)world', 'Hello~~~~~Python~~~world' )
>>> match.group( 1 )
'~~~~~Python~~~'
>>> match = re.match( 'Hello[\t]*world', 'Hello~~~~~Python~~~world' )
>>> match.group( 1 )
###.....................Error Message...........
AttributeError: 'NoneType' object has no attribute 'group'
>>>
```

Code Snippet 25: Pattern Matching

If such a substring is found, portions of the substring matched by parts of the pattern enclosed in parentheses are available as groups.

Following code picks out three groups separated by slashes, and is similar to splitting by an alternative program

```
>>>
>>> match = re.match( '[/:](.*)[/:](.*)[/:](.*)', '/usr/home:lumberjack' )
>>> match.groups()
('usr', 'home', 'lumberjack')

>>> re.split( '[/:]', '/usr/home/lumberjack' )
['', 'usr', 'home', 'lumberjack']
>>>

>>> # Following code returns error.
>>> match = re.match( '[/](.*)[/](.*)[/](.*)', '/usr/home:lumberjack' )
>>> match.groups()
AttributeError: 'NoneType' object has no attribute 'groups'

>>> match = re.match( '[/](.*)[:](.*)[/](.*)', '/usr/home:lumberjack' )
>>> match.groups()
AttributeError: 'NoneType' object has no attribute 'groups'

>>> match = re.match( '[/](.*)[/](.*)[:](.*)', '/usr/home:lumberjack' )
>>> match.groups()
('usr', 'home', 'lumberjack')
>>>
```

Code Snippet 26: Pattern Matching by Groups

2.3. Lists

Most general sequence provided by the language.

__Lists are positionally ordered collections of arbitrary typed objects, and they have no fixed size.__

They are __mutable__: Lists can be modified in place by assignment to offsets as well as a variety of list method calls.

A very flexible tool for representing arbitrary collections: - lists of files in a folder, employees in a company etc.

2.3.1. Sequence operations:

List supports all sequence operations (discussed for strings)

Result of those operation will be a list itself.

```
>>>
>>> L = [123, 'spam', 1.23]      # A list of three different type objects
>>> len(L)
3
>>> L[0]                         # Indexing
123
>>> L[-1]
1.23
>>> L[:-1]                       # Slicing
[123, 'spam']
>>> L + [4, 5, 6]                # Concat/repeat makes new lists too
[123, 'spam', 1.23, 4, 5, 6]
>>> L * 2
[123, 'spam', 1.23, 123, 'spam', 1.23]
>>> L
[123, 'spam', 1.23]
>>>
```

Code Snippet 27: Sequence operations in a list

2.3.2. Type-Specific Operations:

- Have no fixed type constraint
- Have no fixed size: can grow and shrink on demand

```
>>> L.append('NI')      # Growing: add an object at the end of the list
>>> L
[123, 'spam', 1.23, 'NI']
>>> L.pop(2)               # Shrinking: deleting an item in the middle
1.23
>>> L
[123, 'spam', 'NI']

>>> L.append(1.23)
>>> L
[123, 'spam', 'NI', 1.23]
>>> del L[3]             # To remove an item from an index
>>> L
[123, 'spam', 'NI']
>>>

>>> # To insert an item at an arbitrary position
>>> L.insert(3, '4u')       # Item is inserted before the specified index.
>>> L
[123, 'spam', 'NI', '4u']
>>> L.insert(6, '4u')       # Item is inserted as the last item, even though
specified index is not valid.
>>> L
[123, 'spam', 'NI', '4u', '4u']

>>> # Removes a given item by value
>>> L.remove('4u')          # removes the first occurance of the value
>>> L
[123, 'spam', 'NI', '4u']

>>> # Add multiple items at the end
>>> L.extend( 'A1')      # Iterates through the given value and adds each .
>>> L
[123, 'spam', 'NI', '4u', 'A', '1']
>>> L.extend( 567 )
TypeError: 'int' object is not iterable
>>> L.extend( '5''67' )
>>> L
[123, 'spam', 'NI', '4u', 'A', '1', '5', '6', '7']

>>> M = ['bb', 'xx', 'aa' ]
>>> M
['bb', 'xx', 'aa']
>>> M.sort()
>>> M
['aa', 'bb', 'xx']
>>> M.reverse()
>>> M
['xx', 'bb', 'aa']
>>>
```

Code Snippet 28: Type specific operations in a list

2.3.3. Bounds Checking:

Pythons doesn't allow to refer items that are not present in the List

Indexing off the end of a list and Assigning off the end are mistakes. Python throws error.

```
>>> L
[123, 'spam', 1.23]
>>> L[99]
Traceback (most recent call last):
  File "<pyshell#336>", line 1, in <module>
    L[99]
IndexError: list index out of range
>>> L[99] = 21
Traceback (most recent call last):
  File "<pyshell#337>", line 1, in <module>
    L[99] = 21
IndexError: list assignment index out of range
>>> L.insert(99, 21)
>>> L
[123, 'spam', 1.23, 21]
>>>
```

Code Snippet 29: Bounds checking in a List

2.3.4. Nesting:

Supports arbitrary nesting.

```
>>> # A list containing 3 other lists - matrix
>>> Matrix = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]]
>>> Matrix
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> # To get second row
>>> Matrix[1]
[4, 5, 6]
>>> # Get the item at second row, third column
>>> Matrix[1][2]
6
>>>
```

Code Snippet 30: Nesting in Lists

2.3.5. Comprehensions:

List comprehensions derive from set notation; they are a way to build a new list by running an expression on each item in a sequence, one at a time, from left to right.

List comprehensions are coded in square brackets and are composed of an expression and a looping construct that share a variable name.

(_row_ in the following example)

```
>>>
>>> # To get the column of a matrix by list comprehension
>>> Matrix
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> col2 = [row[1] for row in Matrix]
>>> col2
[2, 5, 8]
>>> Matrix
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>>
```

Code Snippet 31: List Comprehension - Introduction

This comprehension means "Give me row[1] for each row in the matrix Matrix, in a new list." The result is a new list containing column 2 of the matrix.

List comprehensions can become more complex:

```
>>>
>>> # To add 1 to each item in column 2
>>> [row[1] + 1 for row in Matrix]
[3, 6, 9]
>>> # To filter out odd items
>>> [row[1] for row in Matrix if row[1] % 2 == 0 ]
[2, 8]
>>>
```

Code Snippet 32: List Comprehension - Complex Operations

List comprehensions make new lists of results, but they can be used to iterate over any *iterable* object.

Example: Can use list comprehensions to step over a hardcoded list of coordinate and string.

```
>>> # Collect diagonal from a matrix
>>> diag = [Matrix[i][i] for i in [0, 1, 2 ]]
>>> diag
[1, 5, 9]
>>>
>>> # Repeat characters in string
>>> doubles = [c * 2 for c in 'spam']
>>> doubles
['ss', 'pp', 'aa', 'mm']
>>>
```

Code Snippet 33: List comprehension - more samples

These expressions can also be used to collect multiple values, as long as those values are wrapped in a nested collection.

Usage of *range*: -

range is a built-in that generates successive integers, and requires a surrounding *list* to display all its value in 3.X only (2.X makes a physical list all at once):

```
>>>
>>> # 0..3 ( list() required in 3.X )
>>> list( range(4))
[0, 1, 2, 3]
>>> # -6 to +6 by 2
>>> list(range(-6, 7, 2))
[-6, -4, -2, 0, 2, 4, 6]
>>>
>>> # Multiple values, if filters
>>> [[x ** 2, x ** 3] for x in range(4)]
[[0, 0], [1, 1], [4, 8], [9, 27]]
>>> [[x, x/2, x * 2] for x in range(-6, 7, 2) if x > 0]
[[2, 1.0, 4], [4, 2.0, 8], [6, 3.0, 12]]
>>>
```

Code Snippet 34: Usage of range *built-in function*

List comprehensions will work on any type that is a sequence in Python, as well as some types that are not.

In recent Pythons, comprehension syntax has been generalized for other roles; it's not just for making roles.

Example: enclosing a comprehension in parentheses can also be used to create ***generators*** that produce results on demand.

The ***sum*** built-in sums items in a sequence.

```
>>>
>>> Matrix
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> # Create a generator of row sums
>>> G = ( sum(row) for row in Matrix)
>>> G
<generator object <genexpr> at 0x0408CC00>
>>> # iter(G) not required here
>>> next(G)
6
>>> next(G)
15
>>> next(G)
24
```

Code Snippet 35: Sample for "generator"

Other samples:

```
>>>
>>> # Map sum over items in the matrix
>>> list(map(sum, Matrix))
[6, 15, 24]
>>> # Create a set of row sums
>>> {sum(row) for row in Matrix}
{24, 6, 15}
>>> #Creates key/value table of row sums
>>> { i : sum( Matrix[i] ) for i in range(3) }
{0: 6, 1: 15, 2: 24}
>>>
>>> # List of character ordinals
>>> [ord(x) for x in 'spam']
[115, 112, 97, 109]
>>>
>>> # Sets remove duplicates
>>> { ord(x) for x in 'spaam' }
{112, 97, 115, 109}
>>>
>>> # Dictionary keys are unique
>>> { x: ord(x) for x in 'spaam' }
{'s': 115, 'p': 112, 'a': 97, 'm': 109}
>>>
>>> # Generator of values
>>> ( ord(x) for x in 'spaam' )
<generator object <genexpr> at 0x0408CB70>
>>>
```

Code Snippet 36: Other comprehension samples

2.4. Dictionaries

Dictionaries are not sequences, instead they are mappings.

Mappings are also collections of other objects, but they are stored by *key* instead of by relative position.

2.4.1. Mapping Operations

1. Mappings doesn't maintain any reliable left-to-right order; they simply map *keys* to associated *values*.
2. Dictionaries are the only mapping type in Python's core objects set
3. Dictionaries are mutable – they may be changed in place and can grow & shrink on demand.
4. Flexible tool for representing collections, like lists, but their more *mnemonic* keys are better suited when a collection's items are named or labelled.
5. Dictionaries are coded in curly braces and consists of a series of key-value pairs.
6. The dictionary can be indexed by key to fetch and change the key's associated values.

```
>>> #### Dictionary
>>>
>>> D = { 'food' : 'Spam', 'quantity' : 4, 'color' : 'pink' }
>>> D
{'food': 'Spam', 'quantity': 4, 'color': 'pink'}
>>> # Fetch the value by key
>>> D['food']
'Spam'
>>> # Add 1 to quantity value. In place modification happens here - i.e.
mutable
>>> D['quantity'] += 1
>>> D
{'food': 'Spam', 'quantity': 5, 'color': 'pink'}
```

Code Snippet 37: Indexing in Dictionary

7. Dictionaries can be built up during run time instead of using curly brackets at the time of creation.
8. Assignments to new dictionary keys created those keys, unlike the out of bound assignments in lists.

```
>>> # Building dictionary on the go
>>> D = {}
>>> # Create keys by assignment
>>> D['name'] = 'Bob'
>>> D['job'] = 'dev'
>>> D['age'] = 40
>>> D
{'name': 'Bob', 'job': 'dev', 'age': 40}
```

Code Snippet 38: Building dictionary on the go

9. Indexing a dictionary by key is often the fastest way to code a search in Python.
10. Dictionaries can also be made using the ***dict*** type name by passing either:
    a. *keyword arguments* (a special ***name=value*** syntax in function calls), or
    b. the result of ***zipping*** together sequences of keys and values obtained at runtime (e.g.: from files).

```
>>>
>>> # By keywords
>>> bob1 = dict(name='Bob', job='dev')
>>> bob1
{'name': 'Bob', 'job': 'dev'}
>>>
>>> # By Zipping
>>> bob2 = dict( zip([ 'name', 'job', 'age' ], ['Bob', 'dev', 40] ))
>>> bob2
{'name': 'Bob', 'job': 'dev', 'age': 40}
>>>
```

Code Snippet 39: Creating Dictionary using "dict" method

11. Mappings are not positionally ordered. They will come back in a different order that the typed order. The exact order may vary per Python.

2.4.2. Nesting Revisited

A dictionary can contain another dictionary or a list as value to support multiple items for a key.

```
>>>
>>> # Nesting in Dictionary
>>> rec = { 'name' : { 'first' : 'Bob', 'last' : 'Smith' },
    'jobs' : ['dev', 'mgr'],
    'age' : 40.5 }
>>> rec
{'name': {'first': 'Bob', 'last': 'Smith'}, 'jobs': ['dev', 'mgr'], 'age':
40.5}
>>>
>>> # 'name' is a nested dictionary
>>> rec['name']
{'first': 'Bob', 'last': 'Smith'}
>>> #Index the nested dictionary
>>> rec['name']['last']
'Smith'
>>> # 'jobs' is a nested list
>>> rec['jobs']
['dev', 'mgr']
>>> # Index the nested list
>>> rec['jobs'][-1]
'mgr'
>>> #Expand Bob's job description in place
>>> rec['jobs'].append( 'janitor' )
>>> rec
{'name': {'first': 'Bob', 'last': 'Smith'}, 'jobs': ['dev', 'mgr',
'janitor'], 'age': 40.5}
>>>
```

Code Snippet 40: Nesting in Dictionary

Here, the append operation was possible because the jobs list is a separate piece of memory from the dictionary that contains it, it can grow and shrink freely.

In Python, clean – up the objects' space is much easier than lower level languages. In Python, when we lose last reference to the object- by assigning it to something else-, all of the memory space occupied by that object's structure is automatically cleaned up.

```
>>>
>>> rec = 0 # Now the objects space is reclaimed
>>> rec
0
>>>
```

Code Snippet 41: Memory clean up

Python has a feature known as *garbage collection* that cleans up unused memory in the program.

2.4.3. Missing Keys: if Tests

Dictionaries also support type specific operations with method calls.

Referencing a non-existent key is an error.

```
>>>
>>> D = { 'a' : 1, 'b' : 2, 'c' : 3 }
>>> D
{'a': 1, 'b': 2, 'c': 3}
>>> # Assigning new keys grows dictionaries
>>> D['e'] = 99
>>> D
{'a': 1, 'b': 2, 'c': 3, 'e': 99}
>>> # referencing a nonexistent key is an error
>>> D['f']
...error text...
KeyError: 'f'
>>>
```

Code Snippet 42: Referencing a non-existent key

Python's sole selection statement: -

The dictionary ***in*** membership expression allows us to query the existence of a key and branch on the result with a Python ***if*** statement.

```
>>> 'f' in D
False
>>> if not 'f' in D:
    print( 'missing' )


missing
>>> if 'f' in D:
    D['f']


>>>
```

Code Snippet 43: Using if

In its full form the if statement can also have an else clause for a default case, and one or more ***elif*** (else if) clauses for other tests.

It's the main selection tool in Python.

Statement blocks are to be indented.

```
>>>
>>> if not 'f' in D:
    print( 'missing' )
    print ( 'no, really')


missing
no, really
>>>
```

Code Snippet 44: if block with indentation

Other ways to avoid accessing non-existent keys:

```
>>>
>>> # get method with a default
>>> value = D.get( 'x', 0)
>>> value
0
>>> # ifelse expression
>>> value = D['x'] if 'x' in D else -1
>>> value
-1
>>>
```

Code Snippet 45: Other methods to test missing keys

2.4.4. Sorting Keys: for Loops

How to impose an ordering on dictionary's items?

Common Solution: - Grab the list of keys in the dictionary with ***keys*** method, sort that with the list ***sort*** method and then iterate through the result with a ***for*** loop.

```
>>>
>>> D = { 'a' : 1, 'x' : 2, 'c' : 3 }
>>> D
{'a': 1, 'x': 2, 'c': 3}
>>> # Unordered keys list
>>> Ks = list( D.keys())
>>> Ks
['a', 'x', 'c']
>>> # Sort the keys list
>>> Ks.sort()
>>> Ks
['a', 'c', 'x']
>>> # Iterate through sorted keys:
>>> for key in Ks:
    print( key, '=>', D[key])


a => 1
c => 3
x => 2
>>>
```

Code Snippet 46: Ordering Dictionary items

Next solution: - Use the built-in function ***sorted***.

```
>>>
>>> D = {'a' : 1, 'c' : 3, 'b' : 2}
>>> D
{'a': 1, 'c': 3, 'b': 2}
>>> for key in sorted(D):
    print( key, '=>', D[key])


a => 1
b => 2
c => 3
>>>
```

Code Snippet 47: Using **sorted** *function*

The ***for*** loop is a simple and efficient way to step through all the items in a sequence and run a block of code for each item in turn. A user defined loop variable (key, here) is used to reference the current item each time though.

The for loop is a sequence operation, but works on some objects that are not.

Python's ***while*** loop is a more general sort of looping tool; it's not limited to stepping across sequences, but generally requires more code to do so.

```
>>>
>>> # Sample code of for loop
>>> for c in 'spam':
    print( c.upper())


S
P
A
M
>>>
>>> # Sample code for while loop
>>> x = 4
>>> while x > 0:
    print( 'spam!' * x )
    x -= 1


spam!spam!spam!spam!
spam!spam!spam!
spam!spam!
spam!
>>>
```

Code Snippet 48: Usage of for and while loops

2.4.5. Iteration and Optimization

An object is iterable if it is

- Either a physically stored sequence in memory

- Or, an object that generates one item at a time in the context of an iteration operation – a sort of "virtual" sequence.

Both types of objects are considered iterable because they support the iteration protocol – they respond to the ***iter*** call with an object that advances in response to the ***next*** calls and raises an exception when finished producing values.

Examples:

- ***generator*** comprehension expression – it's values are produced on request (usually by iteration tools), not stored in memory.
- ***File objects*** similarly iterate line by line when used by an iteration tool
- ***Range***
- Map

Every Python tool that scans an object from left to right uses the iteration protocol. That's why the *sorted* call works on the dictionary directly.

2.5. Tuples

Tuples are sequences that are immutable.
They are used to represent fixed collections of items.
They are normally coded in parentheses instead of square brackets. (Parentheses enclosing can usually be omitted).
They support arbitrary types, arbitrary nesting and the usual sequence operations.

```
>>>
>>> T = (1, 2, 3, 4)     # A 4-item tuple
>>> T
(1, 2, 3, 4)
>>> len(T)        # length
4
>>> T + (5, 6)        # Concatenation'
(1, 2, 3, 4, 5, 6)
>>> T[0]          # Indexing, slicing and more
1
>>>
```

Code Snippet 49: Tuple sequence operations

Tuples also have type specific callable methods:

```
>>>
>>> T
(1, 2, 3, 4)
>>> T.index(4)  # Get the index of value 4
3
>>> T.count(4)  # Get the count of value 4 in tuple T
1
>>>
```

Code Snippet 50: Type specific methods for Tuple

```
>>>
>>> T
(1, 2, 3, 4)

>>> # Tuples are immutable
>>> T[0] = 2
Traceback (most recent call last):
  File "<pyshell#586>", line 1, in <module>
    T[0] = 2
TypeError: 'tuple' object does not support item assignment
>>>
>>> # Make a new tuple for a new value
>>> T = (2,) + T[1:]
>>> T
(2, 2, 3, 4)
>>>
>>>
>>>
>>> # Parentheses can be omitted while creating a Tuple
>>> T = 'spam', 3.0, [11, 22, 33]
>>> T
('spam', 3.0, [11, 22, 33])
>>> T[1]
3.0
>>> T[2][1]
22
>>> T.append(4)
Traceback (most recent call last):
  File "<pyshell#597>", line 1, in <module>
    T.append(4)
AttributeError: 'tuple' object has no attribute 'append'
>>>
```

Code Snippet 51: Immutability of Tuples

Because of their immutability, tuples provide a sort of integrity constraint that is convenient in larger programs.

2.6. Files

File objects are Python code's main interface to external files in the computer. They can be used to read and writes text memos, audio clips, Excel documents, save email messages and so on.

Files are a core type, but there is no specific literal syntax for creating them.

To create a file object, call the built-in *open* function, passing in an external file name and an optional processing name as Strings.

The filename can be a full directory path, if you need to access a file elsewhere in the system.

Example: -

To create a text output file that writes data (‘w’ processing mode string is specified):

```
>>>
>>>
>>> # Make a new file object in output mode ('w' is write)
>>> f = open( 'data.txt', 'w' )
>>>
>>> # Write strings of characters to ot
>>> f.write( 'Hello\n' )
6
>>> # It returns the number of characters written, in Python 3.X
>>> f.write( 'world\n' )
6
>>> # Close to flush output buffers to disk
>>> f.close()
>>>
```

Code Snippet 52: Creating an output file

To read a file, open the file in ‘r’ processing mode. This is the default, if no modes are specified in the open call.

A files contents are always a string, regardless of the type of data the file contains.

```
>>>
>>> # 'r' read is the default processing mode
>>> f = open( 'data.txt')
>>> #Read the entrie file into a string
>>> text = f.read()
>>> text
'Hello\nworld\n'
>>>
>>> # print interprets control characters
>>> print( text )
Hello
world

>>> # File content is always a string
>>> text.split()
['Hello', 'world']
>>>
```

Code Snippet 53: Reading a file

Files provide an iterator that automatically reads line by line in for loops and other contexts:

```
>>>
>>> for line in open( 'data.txt' ) : print(line)

Hello

world

>>>
```

Code Snippet 54: File reading using for loop

2.6.1. Binary Bytes Files

Text files always encode strings in 3.X and blindly write string content in 2.X. This is irrelevant for simple ASCII data, which maps to and from file bytes unchanged. But for richer types of data, the file interfaces can vary depending on both content and the Python line you use. (They rely on either platform's Unicode encoding default in Python 3.X, or the 8-bit byte nature of files in Python 2.X.)

Python 3.X draws a sharp distinction between text and binary data in files:

- *Text files* represent content as a normal ```*str*``` strings and perform Unicode encoding and decoding automatically when writing and reading data.
- *Binary files* represent content as special *bytes* string and allow one to access file content unaltered.

Python 2.X supports the same dichotomy, but doesn't impose it as rigidly, and its tools differ.

Binary file handling Sample:

Pythons ***struct*** module can both create and unpack packed binary data- raw bytes that record values that are not Python objects – to be written to a file in binary mode.

```
>>>
>>> import struct
>>> # Create packed binary data
>>> packed = struct.pack( 'i4sh', 7, b'spam', 8 )
>>> packed  # 10 bytes, not objects or texts
b'\x07\x00\x00\x00spam\x08\x00'
>>>
>>> # Open binary output file
>>> file = open( 'data.bin', 'wb' )
>>> # Write the packed data
>>> file.write( packed )
10
>>> file.close()
>>>
>>> # To read the binary data
>>> # Open and read the binary data file
>>> data = open( 'data.bin', 'rb' ).read()
>>> data
b'\x07\x00\x00\x00spam\x08\x00'
>>>
>>> # Slice the bytes in the middle
>>> data[4:8]
b'spam'
>>> # A sequence of 8 bit bytes
>>> list(data)
[7, 0, 0, 0, 115, 112, 97, 109, 8, 0]
>>> # Unpack into object again
>>> struct.unpack( 'i4sh', data )
(7, b'spam', 8)
>>>
```

Code Snippet 55: Binary file handling

2.6.2. Unicode Text Files

Text files are used to process all sorts of text-based data.

To access files containing non-ASCII Unicode test, simply pass in an encoding name if the text in the file doesn't match the default encoding of our platform. In this mode, Python text files automatically encode on writes and decode on reads per the encoding scheme name provided.

In Python 3.X:

```
>>>
>>> S = 'sp\xc4m'    # Non--ASCII Unicode text
>>> S
'spÄm'
>>> S[2]
'Ä'
>>> # Write/encode UTF-8 text
>>> file = open( 'unidata.txt', 'w', encoding='utf-8' )
>>> file.write(S)    # 4 characters are written
4
>>> file.close()
>>>
>>> # Read/decode UTF-8 text
>>> text = open( 'unidata.txt', encoding='utf-8').read()
>>> text
'spÄm'
>>> len(text)
4
>>>
```

Code Snippet 56: Encoding/Decoding per the encoding scheme

To read the encoded data as raw data:

```
>>> raw = open( 'unidata.txt', 'rb').read() # Read raw encoded data
>>> raw
b'sp\xc3\x84m'
>>> len(raw)
5
```

Code Snippet 57: Read raw encoded bytes

To encode and decode a Unicode data from a source other than a file manually: -

```
>>> # Manual encoding and decoding
>>>
>>> text
'spÄm'
>>> # Manual encoding to bytes
>>> text.encode('utf-8')
b'sp\xc3\x84m'
>>>
>>> raw
b'sp\xc3\x84m'
>>> # Manual decoding to str
>>> raw.decode( 'utf-8' )
'spÄm'
>>>
```

Code Snippet 58: Manual encoding/decoding of Unicode data

Encoding of same string under different encoding schemes automatically:

```
>>>
>>> text
'spÄm'
>>> text.encode( 'latin-1' )
b'sp\xc4m'
>>> text.encode( 'utf-16' )
b'\xff\xfes\x00p\x00\xc4\x00m\x00'
>>>
>>> text.encode( 'utf-8' )
b'sp\xc3\x84m'
>>>
>>>
>>> len(text.encode( 'latin-1' )), len( text.encode( 'utf-16' )), len(
text.encode( 'utf-8' ))
(4, 10, 5)
>>>
>>> # Decoding
>>> b'\xff\xfes\x00p\x00\xc4\x00m\x00'.decode( 'utf-16' )
'spÄm'
>>> b'sp\xc4m'.decode( 'latin-1')
'spÄm'
>>>
```

Code Snippet 59: Encoding under different schemes

In Python 2.X:

Almost similar working, but Unicode strings are coded and display with a leading "u", byte strings don't require or show a leading "b", and Unicode text files must be opened with ***codecs.open***, which accepts an encoding name just like 3.X's ***open***, and uses a special ***Unicode*** string to represent content in memory.

Binary file mode may seem optional in 2.X since normal files are just byte-based data, but it's required to avoid changing line ends if present.

```
>>>
>>> import codecs
>>> #2.X: read/decode text
>>> codecs.open( 'unidata.txt', encoding='utf-8').read()
u'sp\xc4m'
>>> # 2.X: read raw bytes
>>> open( 'unidata.txt', 'rb').read()
'sp\xc3\x84m'
>>> # 2.X: raw/undecoded too
'sp\xc3\x84m'
>>>
```

Code Snippet 60: Encoding & decoding in 2.X

Python also supports non-ASCII file names, not just content, but it's largely automatic (by tools like walkers and listers).

2.7. Other Core Types

2.7.1. Sets

- They are neither mappings nor sequences
- They are unordered collections of unique and immutable objects.
- Can create sets in two ways:- using built in function or by using set literals.

```
>>>
>>> # Make a set out of a sequence in 2.X and 3.X
>>> X = set( 'spam' )
>>>
>>> # Make a set with literals in 3.X and 2.7
>>> Y = { 'h', 'a', 'm' }
>>>
>>> # A tuple of two sets without parentheses
>>> X, Y
({'m', 's', 'p', 'a'}, {'a', 'm', 'h'})
>>>
>>> # Intersection
>>> X & Y
{'m', 'a'}
>>>
>>> # Union
>>> X | Y
{'m', 's', 'p', 'a', 'h'}
>>>
>>> # Difference
>>> X - Y
{'s', 'p'}
>>> Y - X
{'h'}
>>>
>>> # Superset
>>> X > Y
False
>>>
>>> # Set comprehensions in 3.X and 2.7
>>> { n ** 2 for n in [1, 2, 3, 4]}
{16, 1, 4, 9}
>>>
```

Code Snippet 61: Usage of Set

- Sets are useful for filtering out duplicates, isolating differences, and performing order neutral equality tests without sorting – in lists, strings and all other iterable objects.

```
>>>
>>> # Filtering out duplicates ( possibly reordered)
>>> list( set([1, 2, 1, 3, 1]))
[1, 2, 3]
>>> # Finding differences in collections
>>> set('spam')  - set('ham')
{'s', 'p'}
>>> set('spam')
{'m', 's', 'p', 'a'}
>>> # Order neutral equality tests ( == is False)
>>> set('spam') == set('asmp')
True
>>>
```

Code Snippet 62: Example of set operations in iterable objects

Set also support in membership tests, though all other collection types in Python do too:

```
>>>
>>> 'p' in set( 'spam' ), 'p' in 'spam', 'ham' in ['eggs', 'spam', 'ham']
(True, True, True)
>>>
```

Code Snippet 63: In membership support of set

2.7.2. Decimals and Fractions

Both decimal numbers and fraction numbers can be used to work around the limitations and inherent accuracies of floating-point math:

```
>>>
>>> # Floating-point ( add a .0 in Python 2.X )
>>> 1 / 3
0.3333333333333333
>>> (2/3) + (1/2)
1.1666666666666665
>>>
>>>
>>> # Decimals fixed precision
>>> import decimal
d
>>> d = decimal.Decimal( '3.141' )
>>> d + 1
Decimal('4.141')
>>>
>>> decimal.getcontext().prec = 2
>>> decimal.Decimal( '1.00' ) / decimal.Decimal( '3.00' )
Decimal('0.33')
>>>
>>> # Fractions: numerator + denominator
>>> from fractions import Fraction
>>> f = Fraction(2, 3)
>>> f + 1
Fraction(5, 3)
>>> f + Fraction(1, 2)
Fraction(7, 6)
>>>
```

Code Snippet 64: Decimals & Fractions

2.7.3. Booleans

Python also comes with ***Booleans***

- Has predefined *True* and *False* objects that are essentially just integers 1 and 0 with custom display logic
- Has long supported a special placeholder called *None* commonly used to initialize names and objects.

```
>>>
>>> # Booleans
>>> 1 > 2, 1 < 2
(False, True)
>>> # Object's Boolean value
>>> bool( 'spam')
True
>>>
>>> # None placeholder
>>> X = None
>>> print(X)
None
>>> # Initialize a list of 100 Nones
>>> L = [None] * 100
>>> L
[None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None]
>>>
```

Code Snippet 65: Booleans

2.8. The TYPE Object

The *type* object, returned by the ***type*** built-in function, is an object that gives the type of another object. It's result slightly differs in 3.X, because types have merged with classes completely.

```
>>> # In Python 2.X
>>> type( L)             # Types: type of L is a list type object
<type 'list'>
>>> type( type ( L ) )   # Even types are objects
<type 'type'>

>>> # In Python 3.X
>>> type(L)              # 3.X: types are classes and viceversa
<class 'list'>
>>> type( type (L))
<class 'type'>
>>>
```

Code Snippet 66: Behaviour of type *in 2.X and 3.x*

The *type* object allows the code to check the types of the objects it processes. There are atleast 2 ways to do so in Python:

```
>>>
>>> # Type testing
>>> if type(L) == type([]):
    print( 'yes' )


yes
>>> # Using the type name
>>> if type(L) == list:
    print( 'yes' )


yes
>>> # Object-oriented tests
>>> if isinstance( L, list ):
    print( 'yes' )


yes
```

Code Snippet 67: type testing

2.9. User Defined Classes

Classes define new types of objects that extend the core set.

Calling the classes like a function generates instances of the new type, and the class' methods automatically receive the instance being processed by a given method call (in the *self* argument).

Example:

```
>>>
>>> class Worker:
    def __init__( self, name, pay ):     # Initialize when created
        self.name = name          # Self is the new object
        self.pay = pay
    def lastName( self ):
        return self.name.split()[-1]     # Split string on blanks
    def giveRaise( self, percent ):
        self.pay *= (1.0 + percent )     # Update pay in place


>>> # Here, the class attributes/state information are name and pay
>>> # Two functions/methods that defines the behaviour.
>>>
>>> # Make two instances. Each has name and pay attirbutes
>>> bob = Worker('Bob Smith', 50000)
>>> sue = Worker('Sue Jones', 60000)
>>> #Call method: bob is self
>>> bob.lastName()
'Smith'
>>> # Next call, sue is the self subject
>>> sue.lastName()
'Jones'
>>> # Updates sue's pay
>>> sue.giveRaise(.10)
>>> sue.pay
66000.0
>>> sue.name
'Sue Jones'
>>>
```

Code Snippet 68: User defined classes - sample code

The inheritance mechanism of the classes supports the software hierarchies that lend themselves to customization by *extension*. Software is extended by writing new classes, not by changing that already works.

Extra Notes:

- They are called “core” types because they are part of the Python language itself and are always available.
- Most of the core types have specific syntax for generating the objects
- Immutable object is an object that cannot be changed after it is created. Numbers, strings and tuples are immutable.
- A sequence is a positionally ordered collection of objects. E.g.: Strings, lists, tuples.
- Iterable means either a physical sequence or a virtual one that produces its items on request.
- Polymorphism means that the meaning of an operation depends on the objects being operated on.