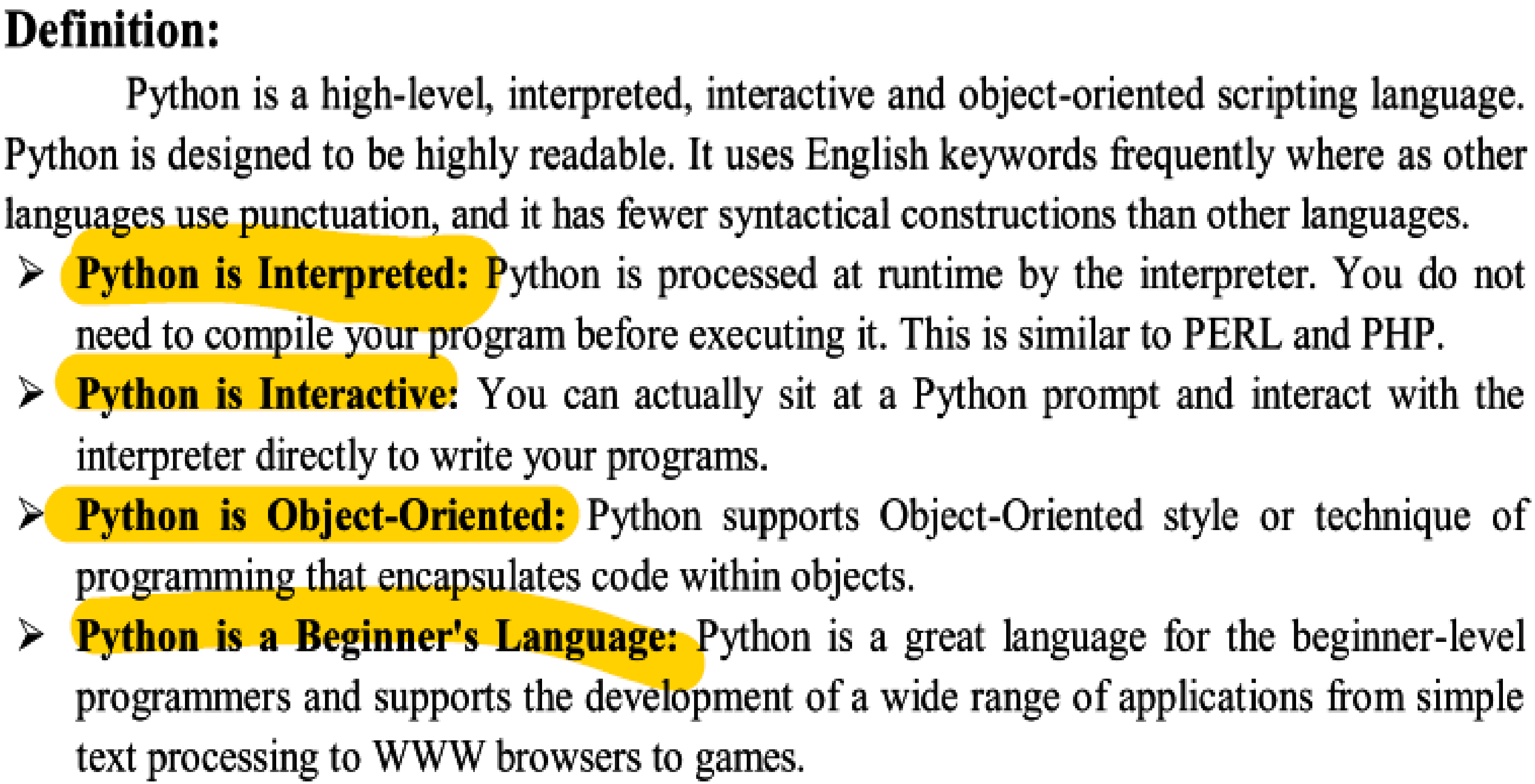
Unit-1

- Python Basics
- Python Objects
- Standard Types
- Other Built-in Types
- Internal Types
- Standard Type Operators
- Standard Type Built-in Functions
- Categorizing the Standard Types

- Unsupported Types
- Numbers - Introduction to Numbers
- Integers, Floating Point Real Numbers
- Complex Numbers
- Operators
- Built-in Functions
- Related Modules
- Sequences - Strings, Lists and Tuples,
- Mapping and Set Types

Python



Definition:

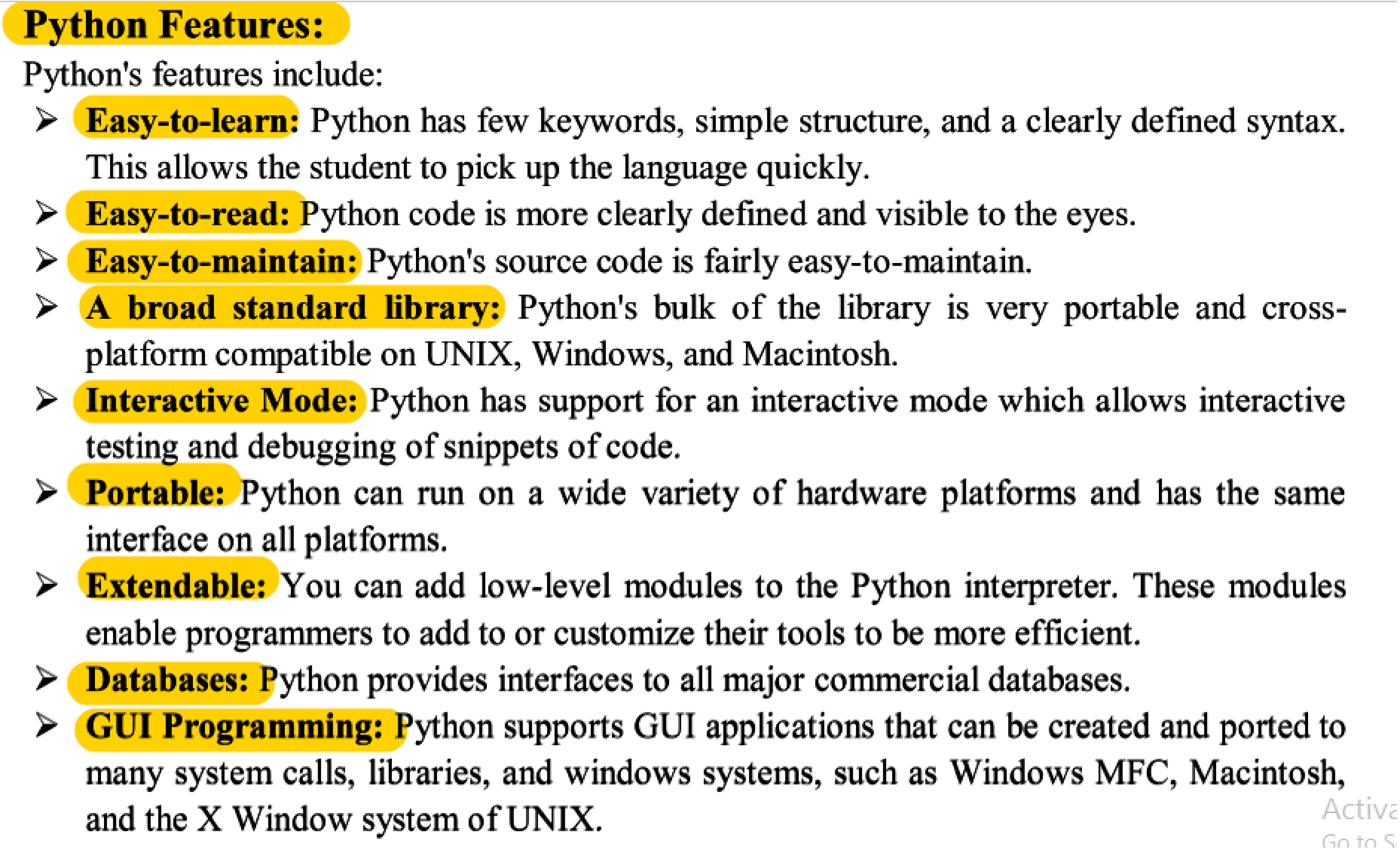
Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

- **Python is Interpreted:** Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive:** You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented:** Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language:** Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

History of Python

- Python was developed by **Guido van Rossum** in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.
- Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, Unix shell, and other scripting languages.

- Python is now maintained by a core development team at the institute, although **Guido van Rossum** still holds a vital role in directing its progress.
- Python 1.0 was released on **20 February, 1991.**
- Python 2.0 was released on **16 October 2000** and had many major new features, including a cycle detecting garbage collector and support for Unicode. With this release the development process was changed and became more transparent and community-backed.
- Python 3.0 (which early in its development was commonly referred to as Python 3000 or py3k), a major, backwards-incompatible release, was released on **3 December 2008** after a long period of testing. Many of its major features have been back ported to the backwards-compatible Python 2.6.x and 2.7.x version series.
- In January 2017 Google announced work on a Python 2.7 to go transcompiler, which The Register speculated was in response to Python 2.7's planned end-of-life.



Python Features:

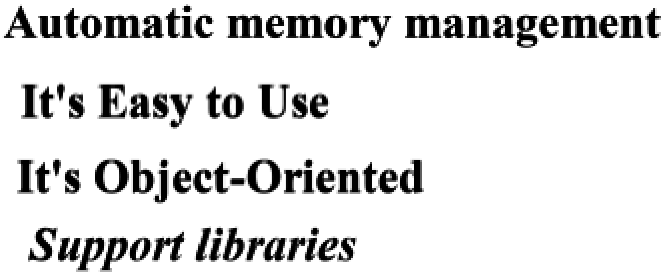
Python's features include:

- **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read:** Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.
- **A broad standard library:** Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode:** Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases:** Python provides interfaces to all major commercial databases.
- **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries, and windows systems, such as Windows MFC, Macintosh, and the X Window system of UNIX.

Need for python:

Portable

Open Source Soft ware

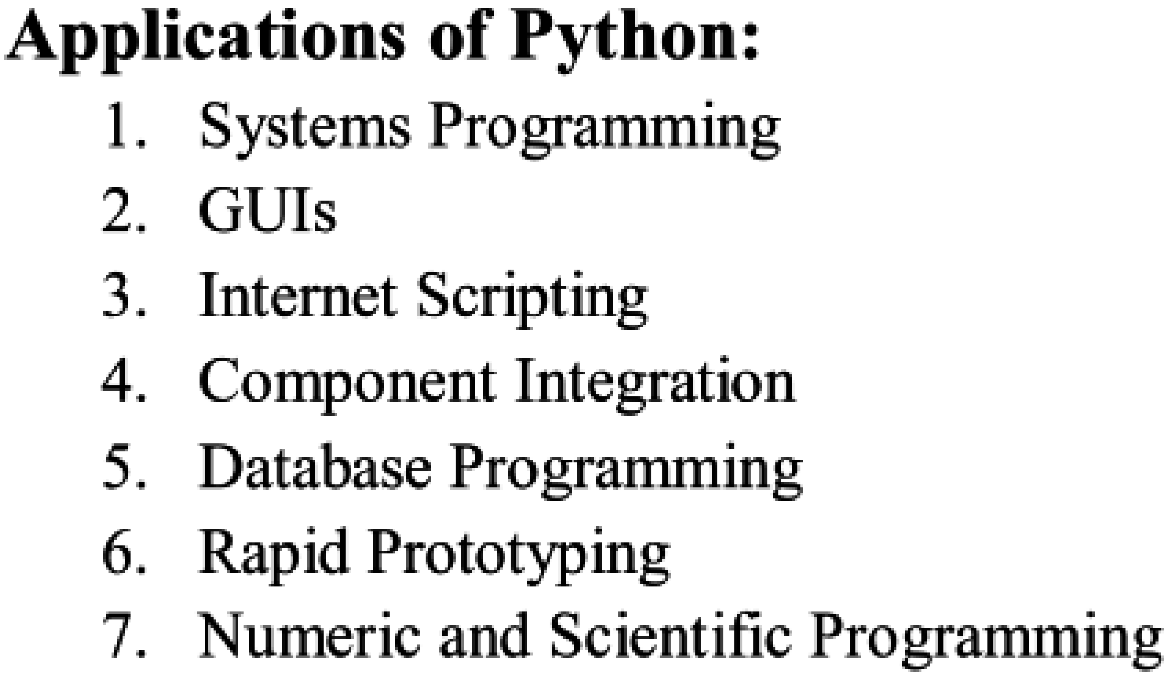


Automatic memory management

It's Easy to Use

It's Object-Oriented

Support libraries



Applications of Python:

1. Systems Programming
2. GUIs
3. Internet Scripting
4. Component Integration
5. Database Programming
6. Rapid Prototyping
7. Numeric and Scientific Programming

Indentation

Code blocks are identified by indentation rather than using symbols like curly braces. Without extra symbols, programs are easier to read. Also, indentation clearly identifies which block of code a statement belongs to. Of course, code blocks can consist of single statements, too. When one is new to Python, indentation may come as a surprise. Humans generally prefer to avoid change, so perhaps after many years of coding with brace delimitation, the first impression of using pure indentation may not be completely positive. However, recall that two of Python's features are that it is simplistic in nature and easy to read.

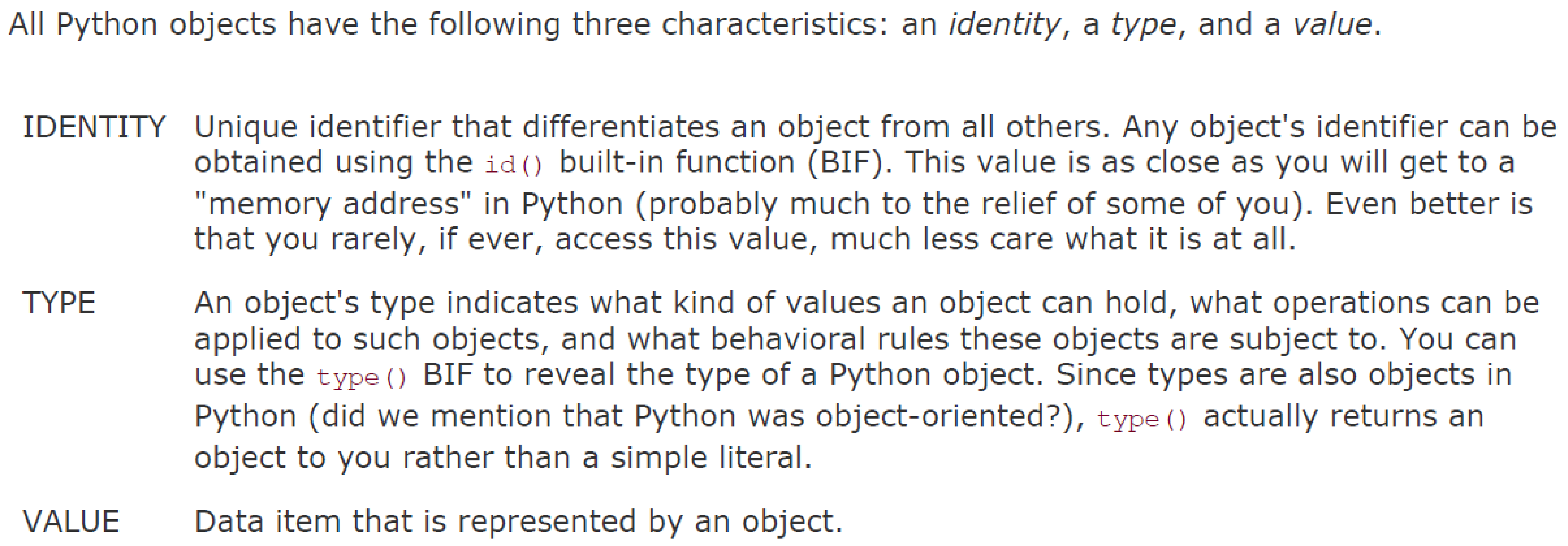
Python does not support braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation. All the continuous lines indented with same number of spaces would form a block. Python strictly follow indentation rules to indicate the blocks.

Python Objects

Python uses the object model abstraction for data storage.

Any construct that contains any type of value is an object.

Python is an "object-oriented programming (OOP) language.



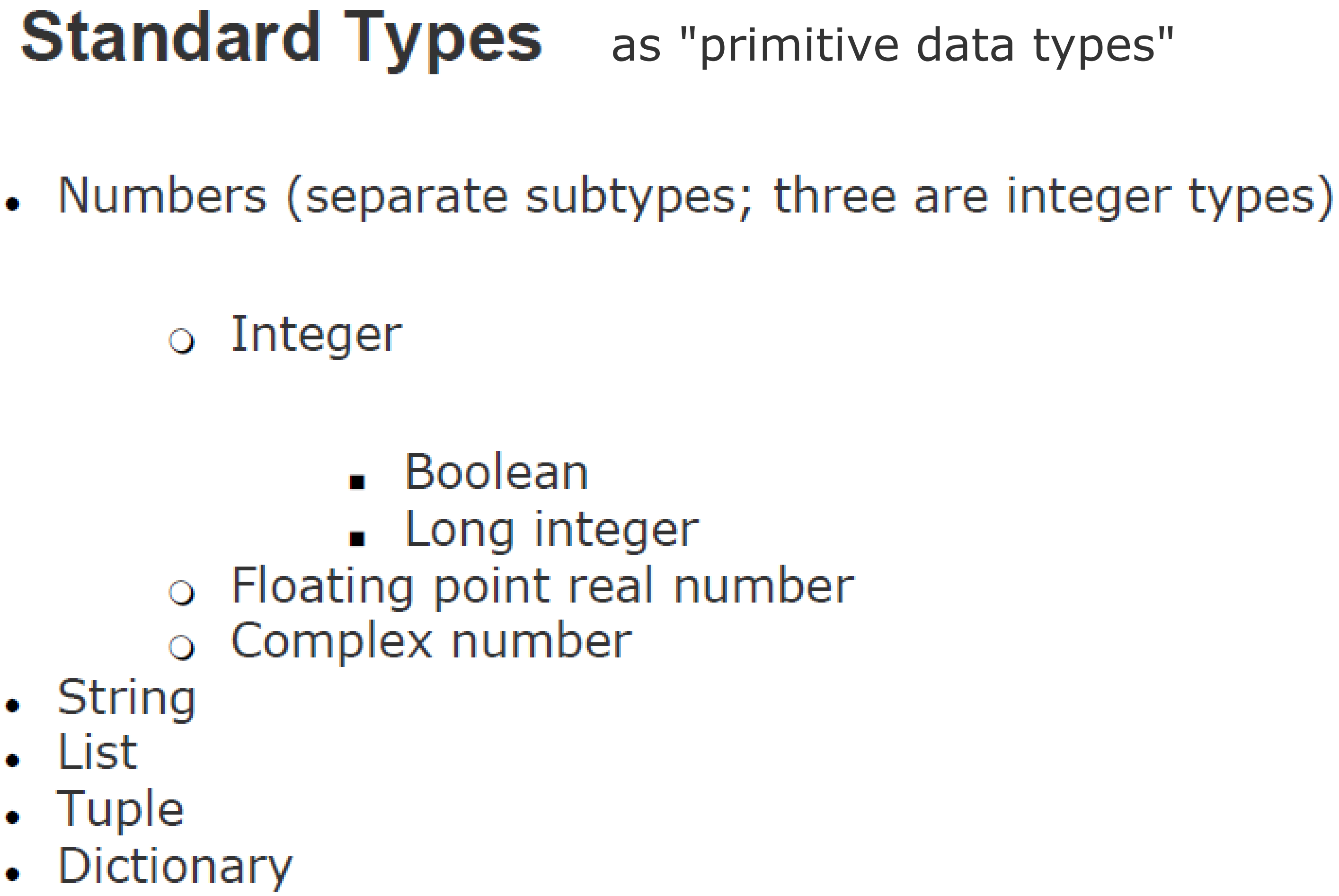
All Python objects have the following three characteristics: an *identity*, a *type*, and a *value*.

IDENTITY Unique identifier that differentiates an object from all others. Any object's identifier can be obtained using the `id()` built-in function (BIF). This value is as close as you will get to a "memory address" in Python (probably much to the relief of some of you). Even better is that you rarely, if ever, access this value, much less care what it is at all.

TYPE An object's type indicates what kind of values an object can hold, what operations can be applied to such objects, and what behavioral rules these objects are subject to. You can use the `type()` BIF to reveal the type of a Python object. Since types are also objects in Python (did we mention that Python was object-oriented?), `type()` actually returns an object to you rather than a simple literal.

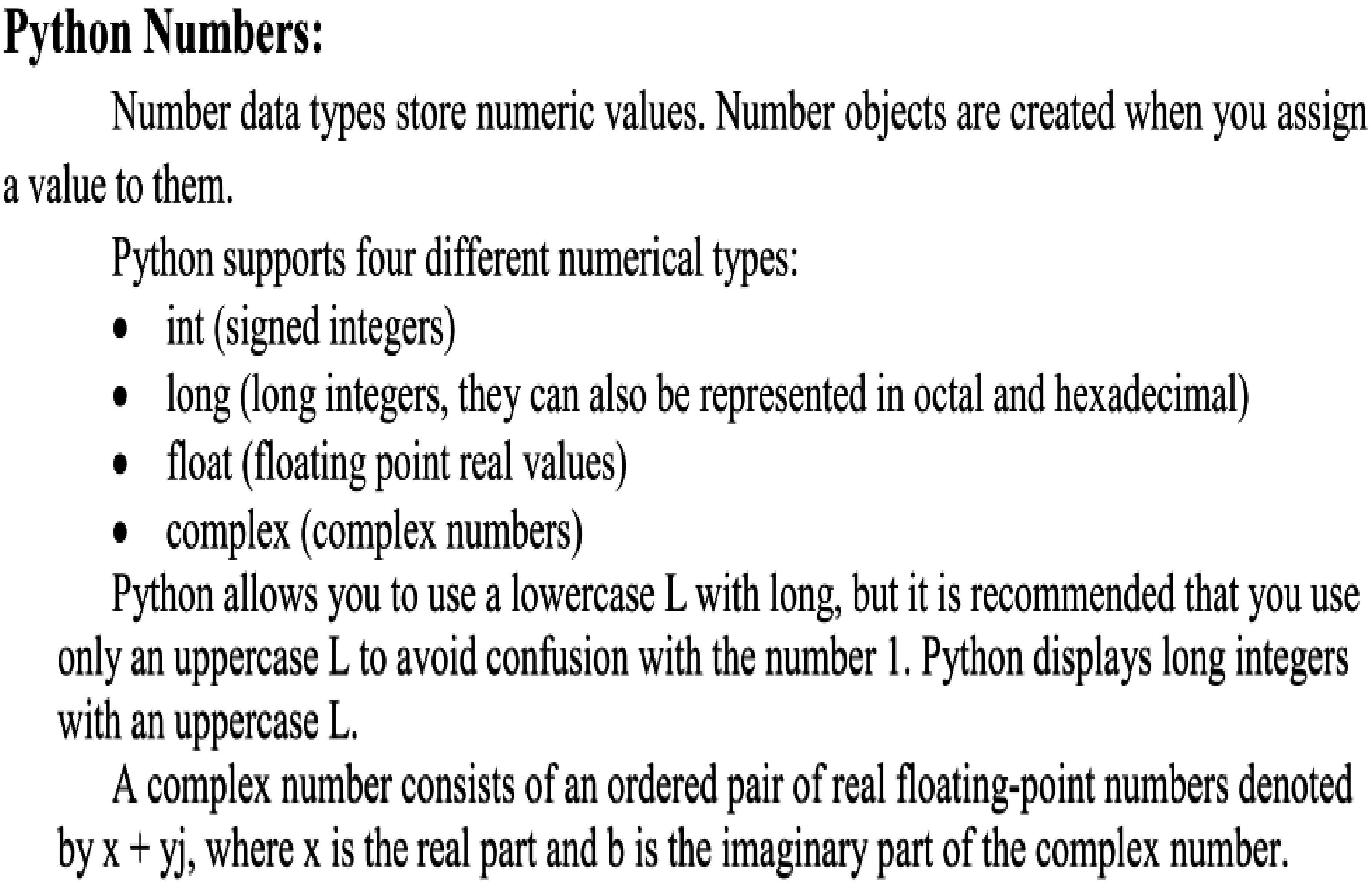
VALUE Data item that is represented by an object.

Id() method return the address of the object
Type() method return the type of the object
Value return the data in the object



Standard Types as "primitive data types"

- Numbers (separate subtypes; three are integer types)
  - Integer
    - Boolean
    - Long integer
  - Floating point real number
  - Complex number
- String
- List
- Tuple
- Dictionary



Python Numbers:

Number data types store numeric values. Number objects are created when you assign a value to them.

Python supports four different numerical types:

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
- complex (complex numbers)

Python allows you to use a lowercase L with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.

A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x is the real part and b is the imaginary part of the complex number.

Python Numbers

There are three numeric types in Python:

- Int:integer, is a whole number, positive or negative, without decimals, of unlimited length.
- float: "floating point number" is a number, positive or negative, containing one or more decimals.
- complex: Complex numbers are written with a "j" as the imaginary part

Example

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
```

‘’’To verify the type of any object in Python, use the type() function:’’’

```
print(type(x))
print(type(y))
print(type(z))
```

```
x = 3+5j
y = 5j
z = -5j

print(type(x))
print(type(y))
print(type(z))
```

Type Conversion:

You can convert from one type to another with the int(), float(), and complex() methods.

Example: Convert from one type to another

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex

#convert from int to float:
a = float(x)
#convert from float to int:
b = int(y)
#convert from int to complex:
c = complex(x)
print(a)
print(b)
print(c)
print(type(a))
print(type(b))
print(type(c))
```

Python Boolean:

Boolean represent one of two values: True or False.
You can evaluate any expression in Python, and get one of two answers, True or False.
When you compare two values, the expression is evaluated and Python returns the Boolean answer

```
Example1:
print(10 > 9)
print(10 == 9)
print(10 < 9)
```

```
Example2:
a = 200
b = 33
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than
a")
```

Evaluate Values and Variables:

The bool() function allows you to evaluate any value, and give you True or False in return

Example1

```
#Evaluate a string and a number
print(bool("Hello"))
print(bool(15))
```

Example2

```
#Evaluate two variables:
x = "Hello"
y = 15

print(bool(x))
print(bool(y))
```

Most Values are True

Almost any value is evaluated to True if it has some sort of content.

Any string is True, except empty strings.

Any number is True, except 0.

Any list, tuple, set, and dictionary are True, except empty ones.

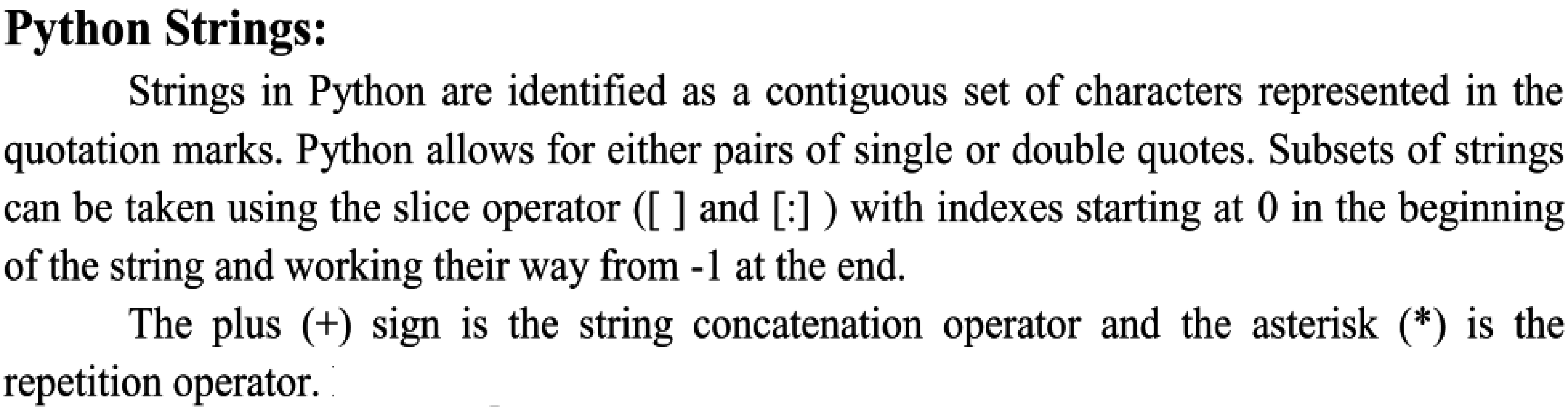
```
bool("abc")
bool(123)
bool(["apple", "cherry", "banana"])
```

Some Values are False

There are not many values that evaluate to False, except empty values, such as (), [], {}, "", the number 0, and the value None. And of course the value False evaluates to False.

Example:

```
bool(False)
bool(None)
bool(0)
bool("")
bool(())
bool([])
bool({})
```



Python Strings:

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator.

String

Strings in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

You can display a string literal with the print() function:

Example:

```
print("Hello")
print('Hello’)
```

Assign String to a Variable:

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

Example:

```
a = "Hello"
print(a)
```

Multiline Strings

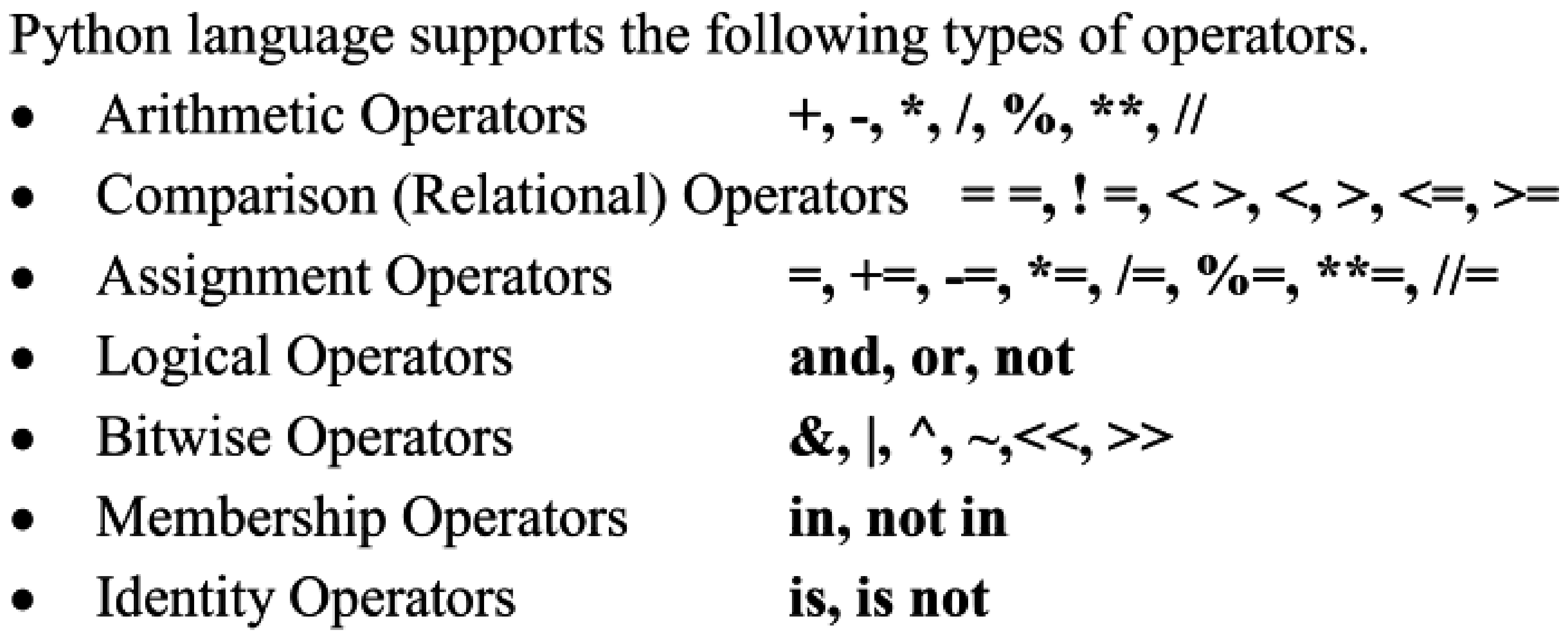
You can assign a multiline string to a variable by using three quotes:

Example:

```
a = """It may seem unusual to regard types themselves as objects
since we are attempting to just describe all
of Python's types."
print(a)
```

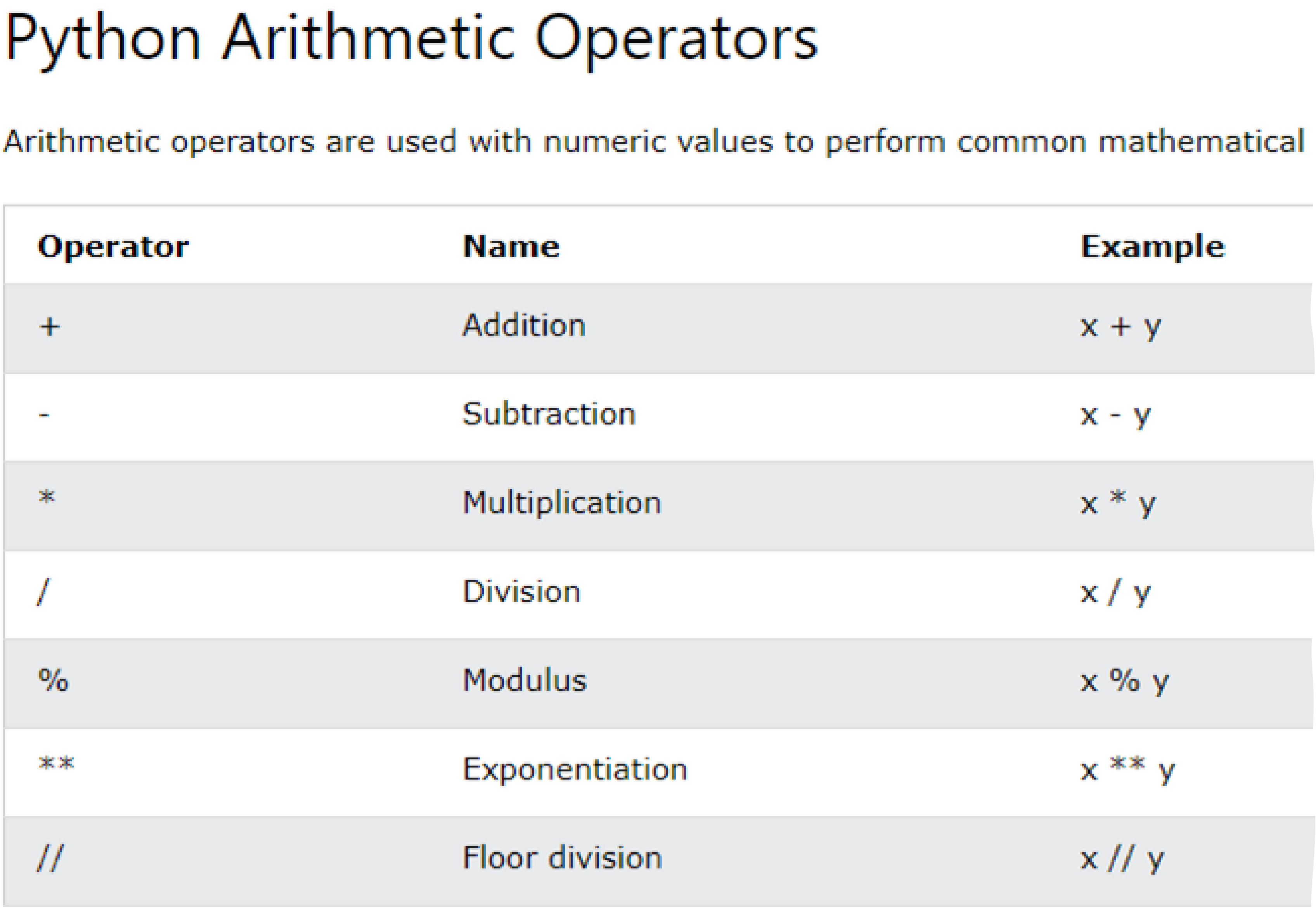
Types of Operators:

Operators are used to perform operations on variables and values.



Python language supports the following types of operators.

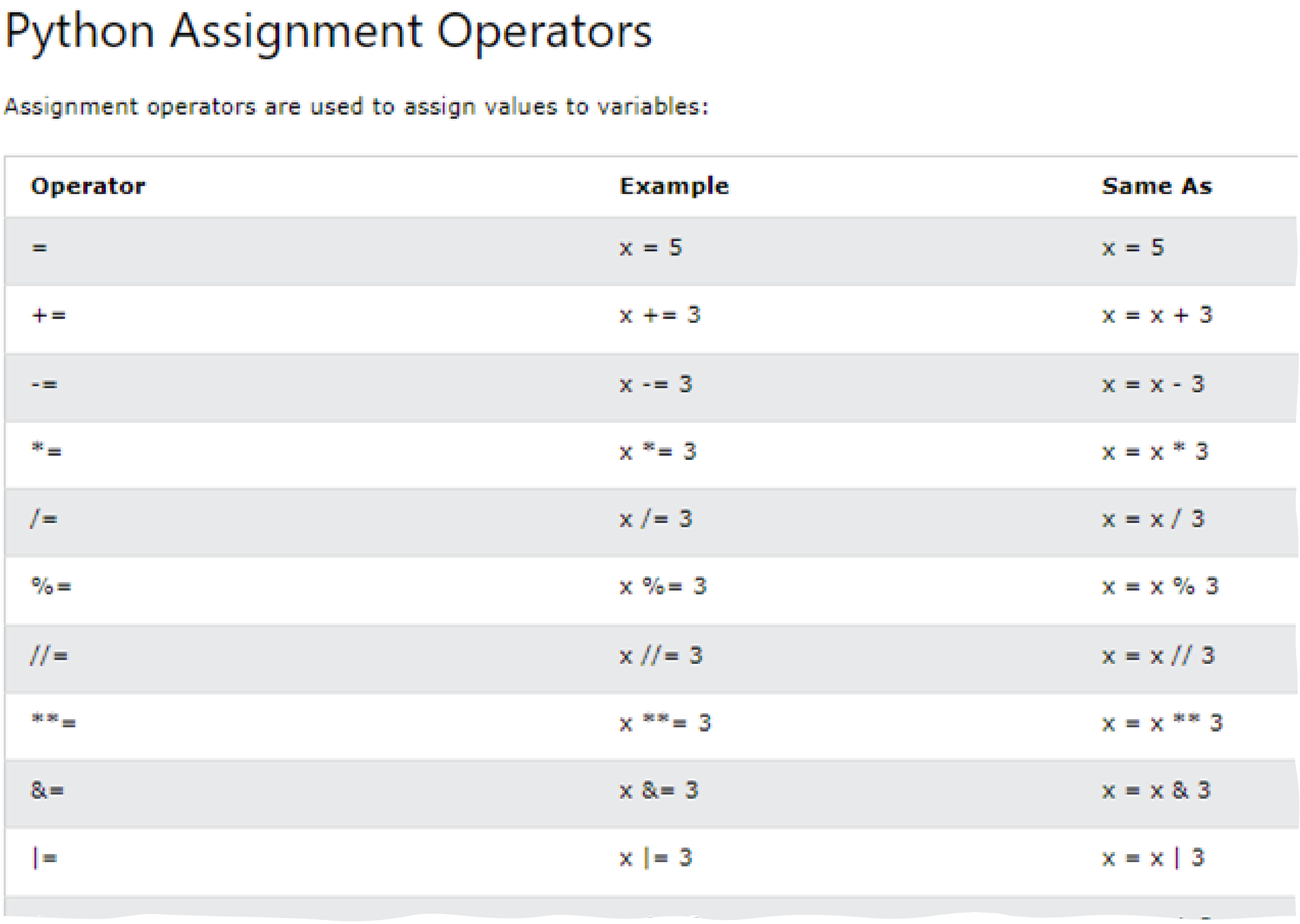
• Arithmetic Operators	+, -, *, /, %, **, //
• Comparison (Relational) Operators	= =, ! =, < >, <, >, <=, >=
• Assignment Operators	=, +=, -=, *=, /=, %=, **=, //=
• Logical Operators	**and, or, not**
• Bitwise Operators	&, |, ^, ~,<<, >>
• Membership Operators	**in, not in**
• Identity Operators	**is, is not**



Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical

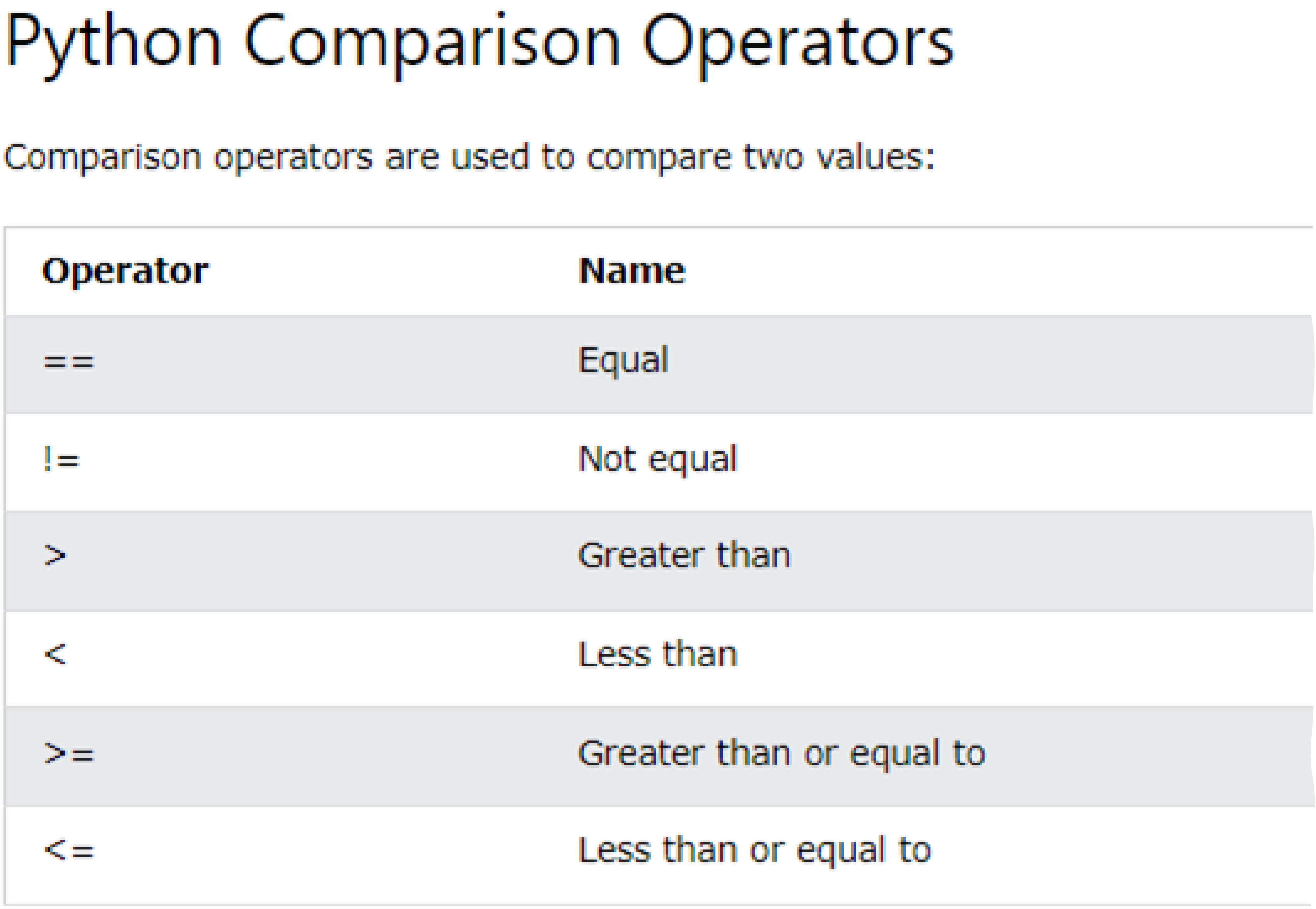
Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y



Python Assignment Operators

Assignment operators are used to assign values to variables:

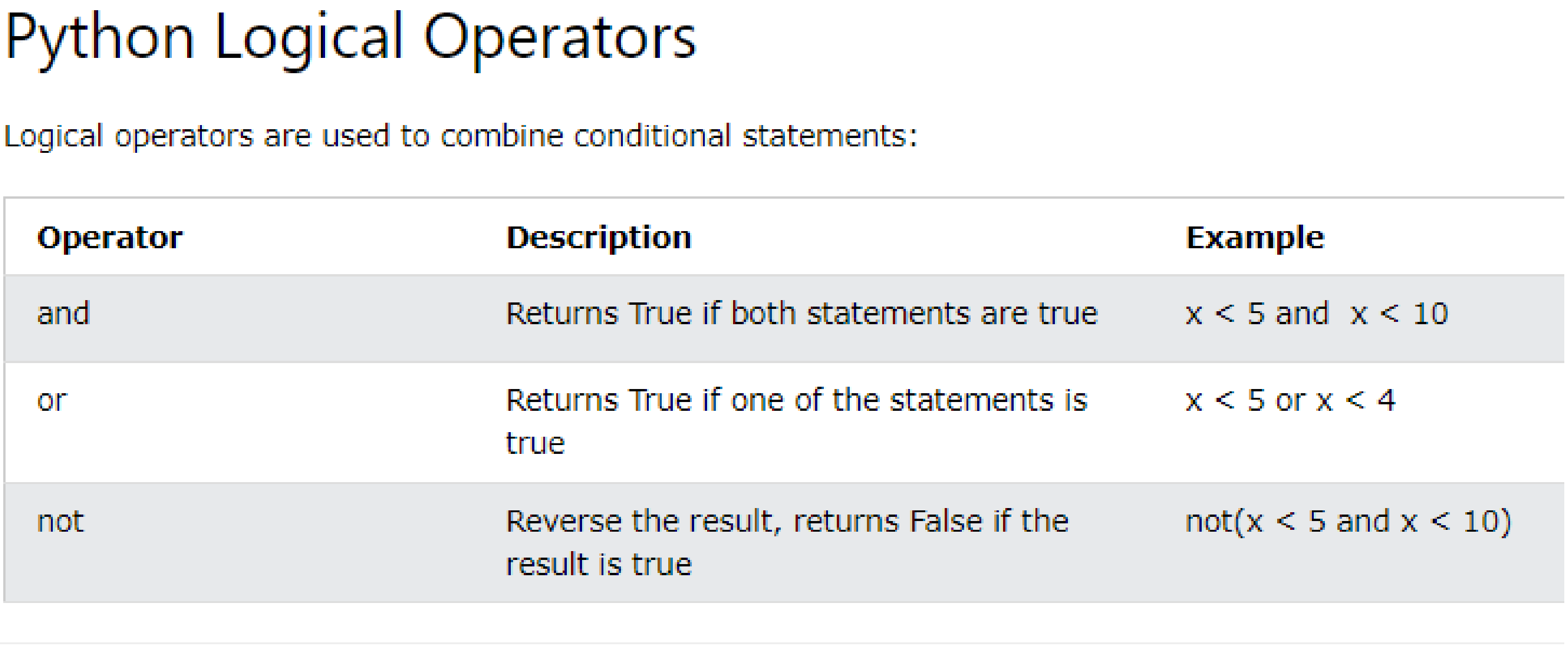
Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3



Python Comparison Operators

Comparison operators are used to compare two values:

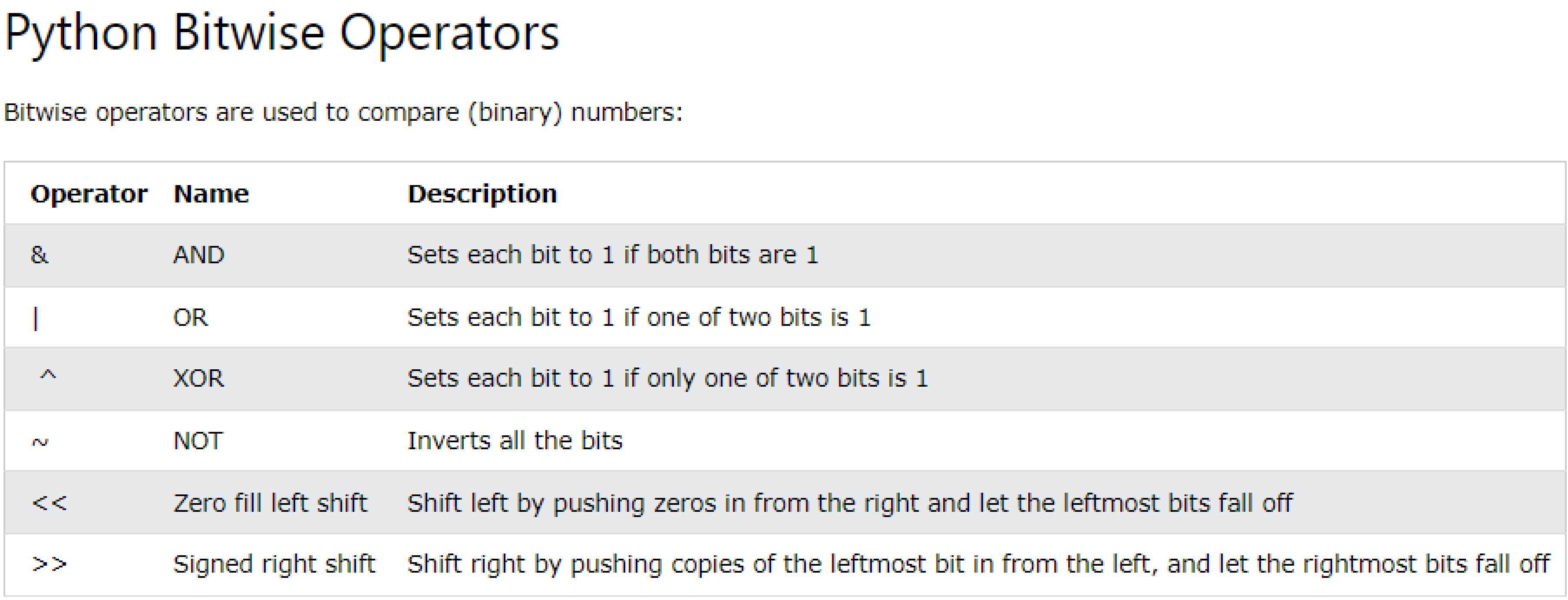
Operator	Name
==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to



Python Logical Operators

Logical operators are used to combine conditional statements:

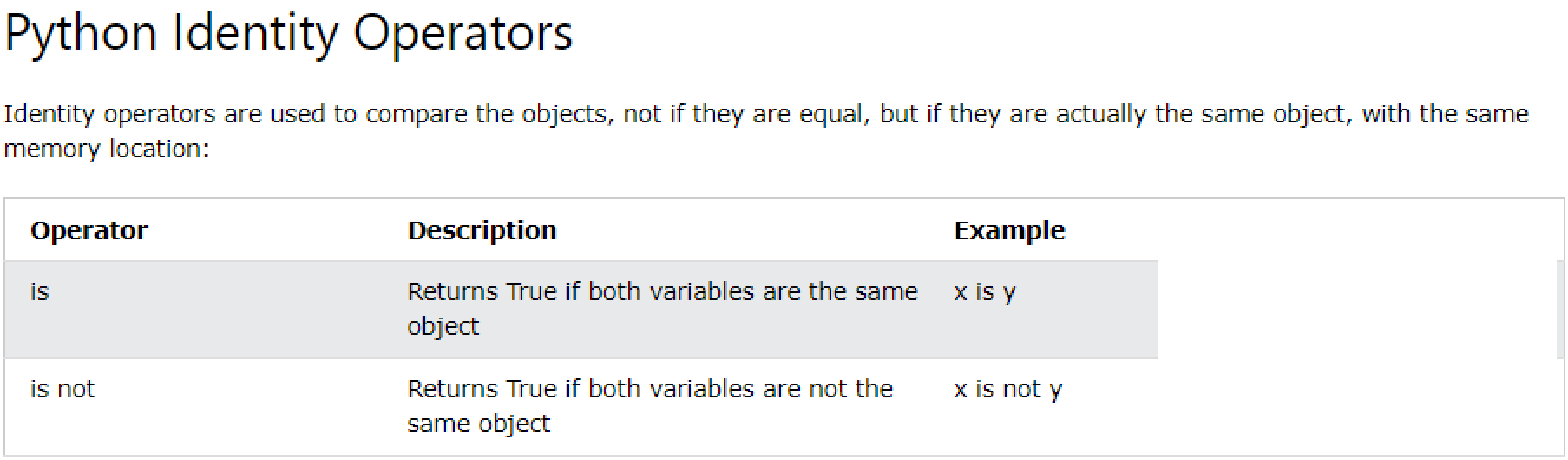
Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)



Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off



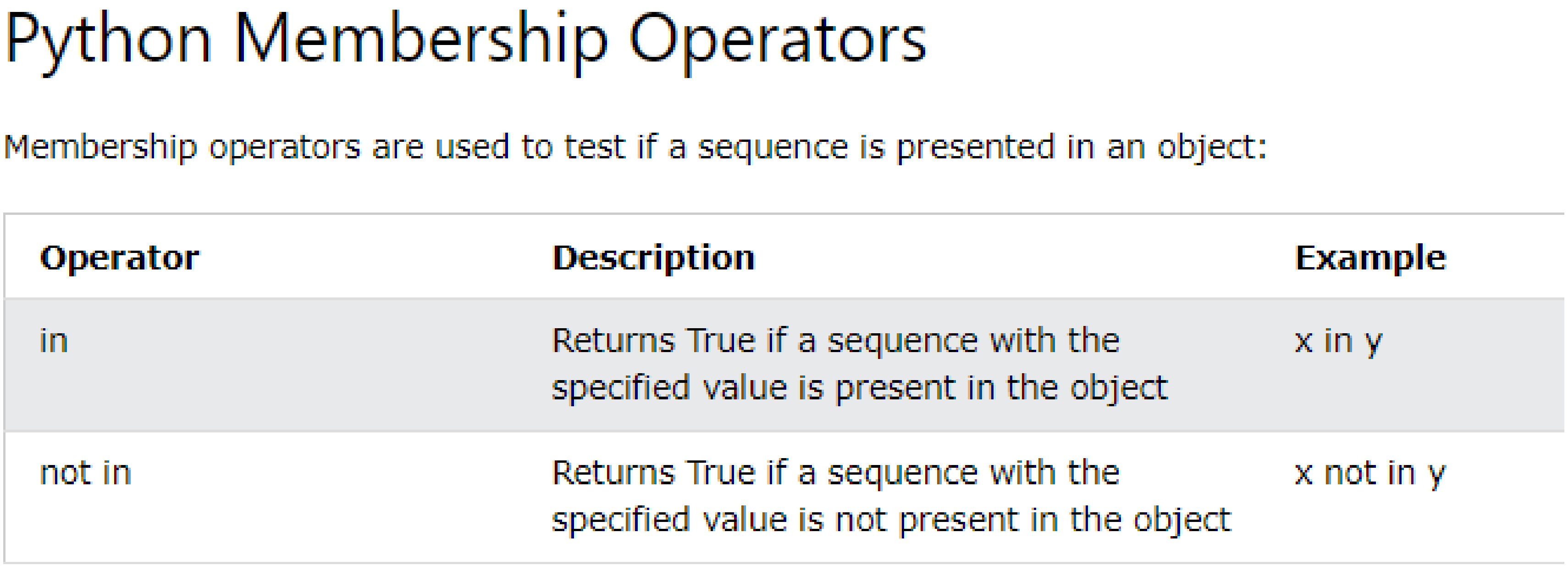
Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x

print(x is z)

# returns True because z is the same object as x

print(x is y)

# returns False because x is not the same object as y, even if they have the
same content

print(x == y)

# to demonstrate the difference betweeen "is" and "==": this comparison
returns True because x is equal to y
```

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x

print(x is not z)

# returns False because z is the same object as
x

print(x is not y)

# returns True because x is not the same object
as y, even if they have the same content

print(x != y)

# to demonstrate the difference betweeen "is
not" and "!=": this comparison returns False
because x is equal to y
```



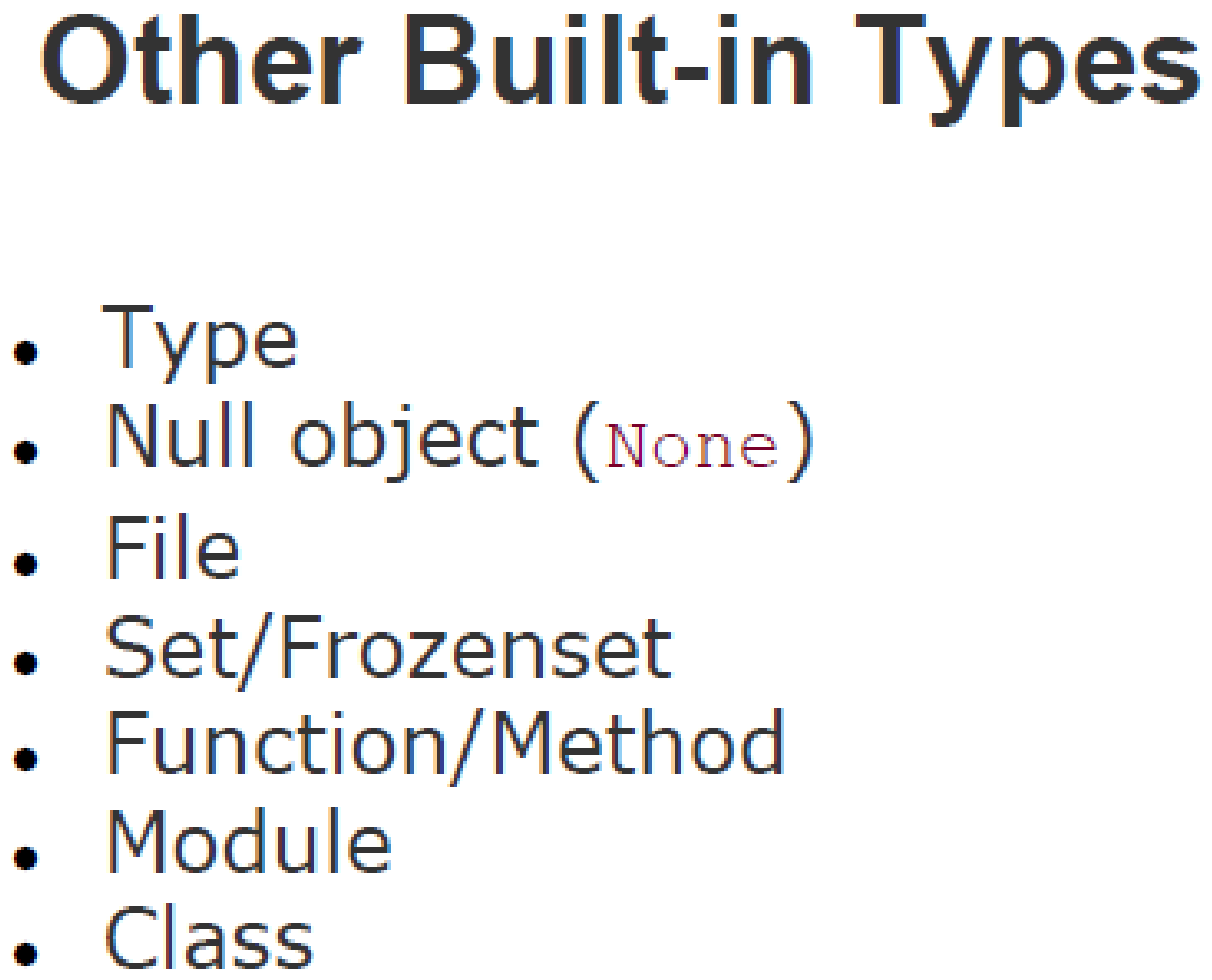
Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

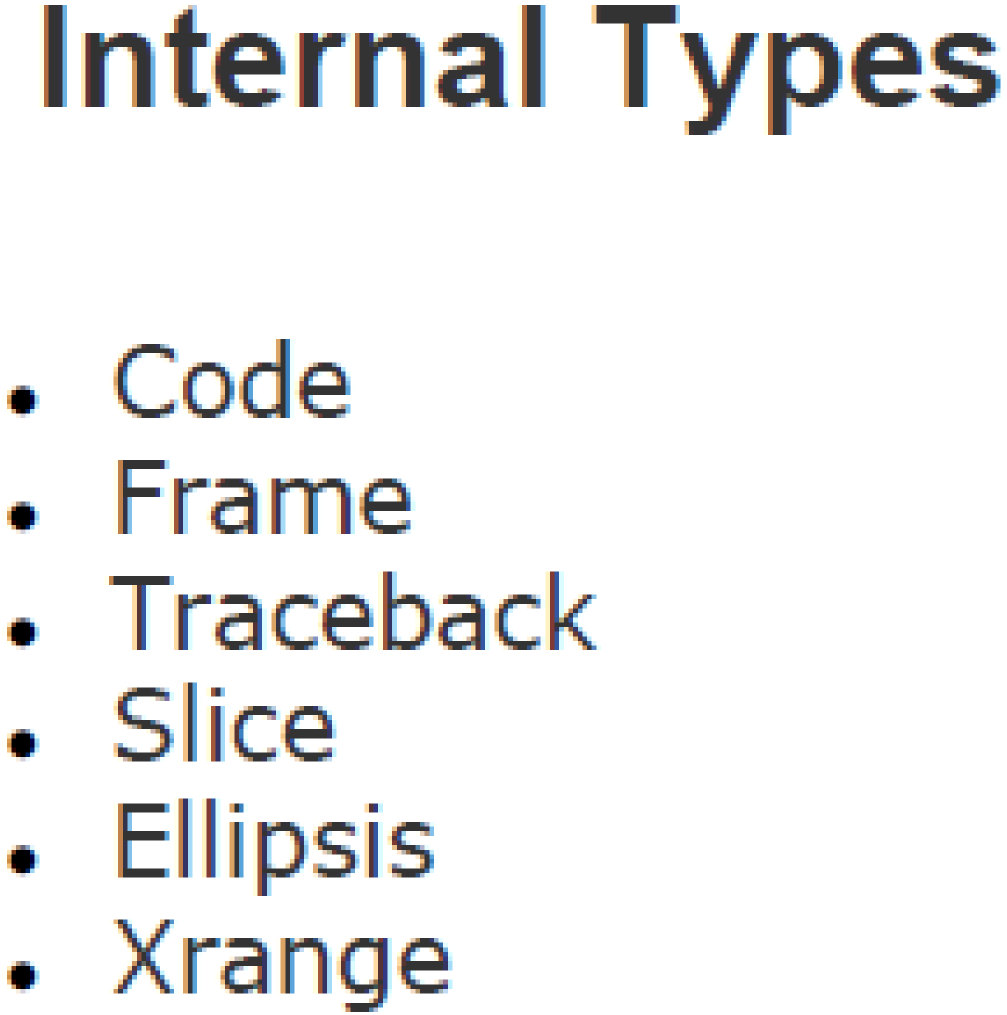
```
x = ["apple", "banana"]

print("banana" in x)

# returns True because a sequence with the value "banana" is in the list
```

```
x = ["apple", "banana"]

print("pineapple" not in x)

# returns True because a sequence with the value "pineapple" is not in the list
```



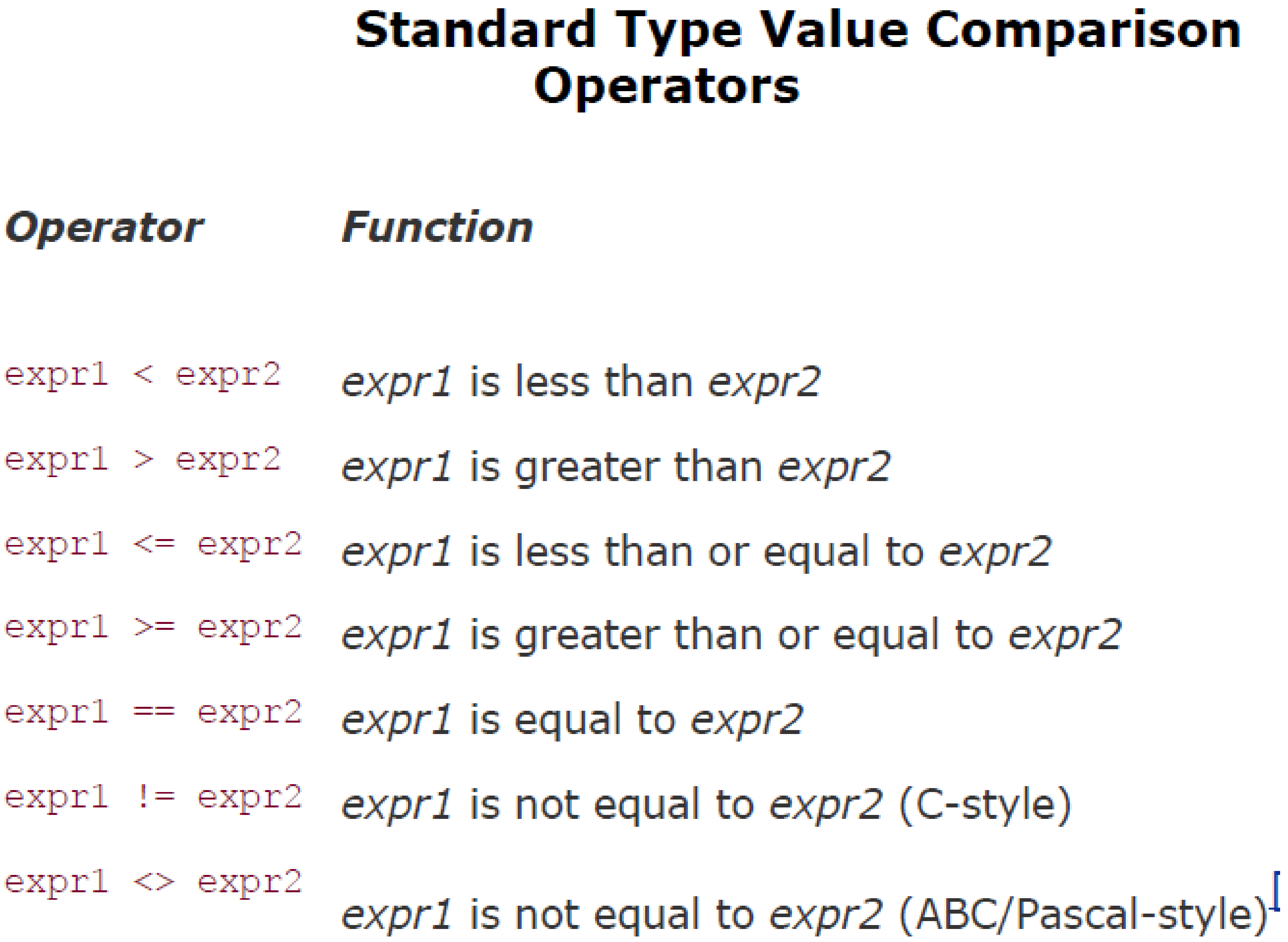
Other Built-in Types

- Type
- Null object (`None`)
- File
- Set/Frozenset
- Function/Method
- Module
- Class



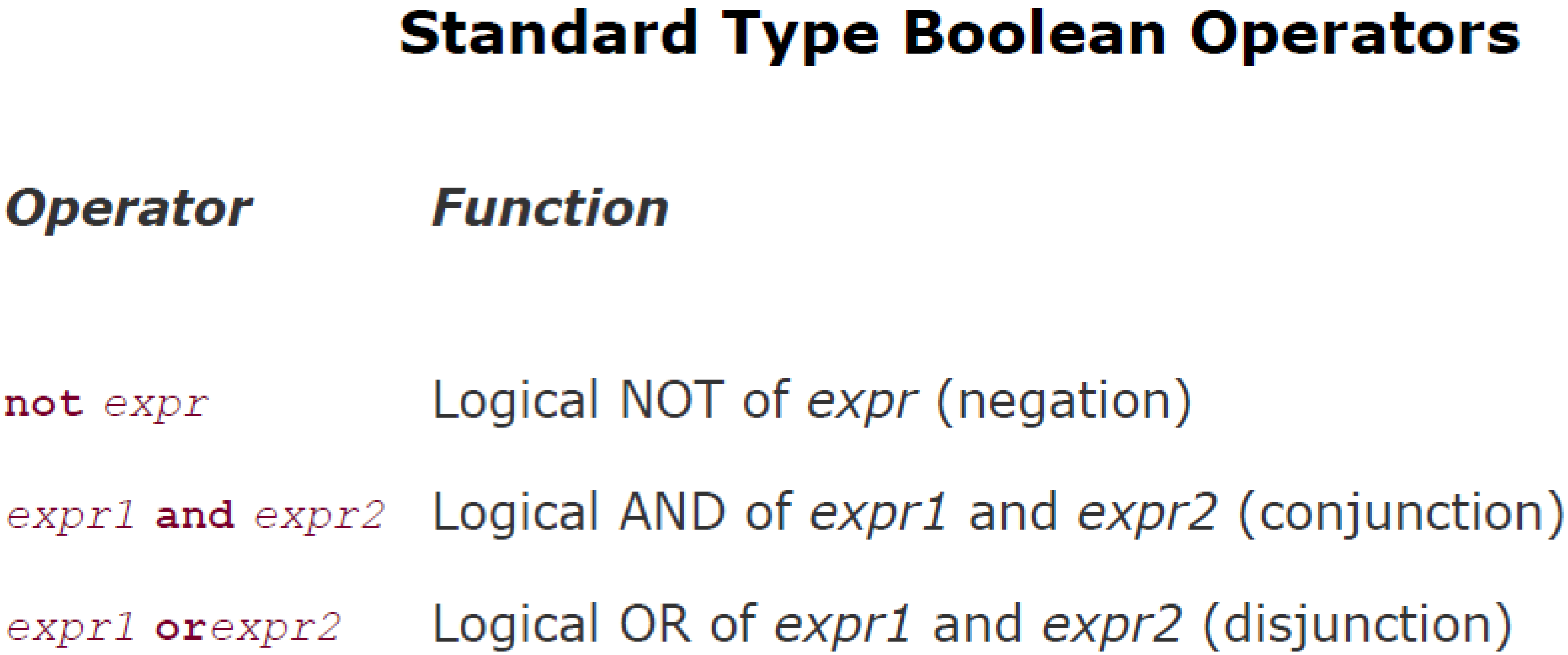
Internal Types

- Code
- Frame
- Traceback
- Slice
- Ellipsis
- Xrange



Standard Type Value Comparison Operators

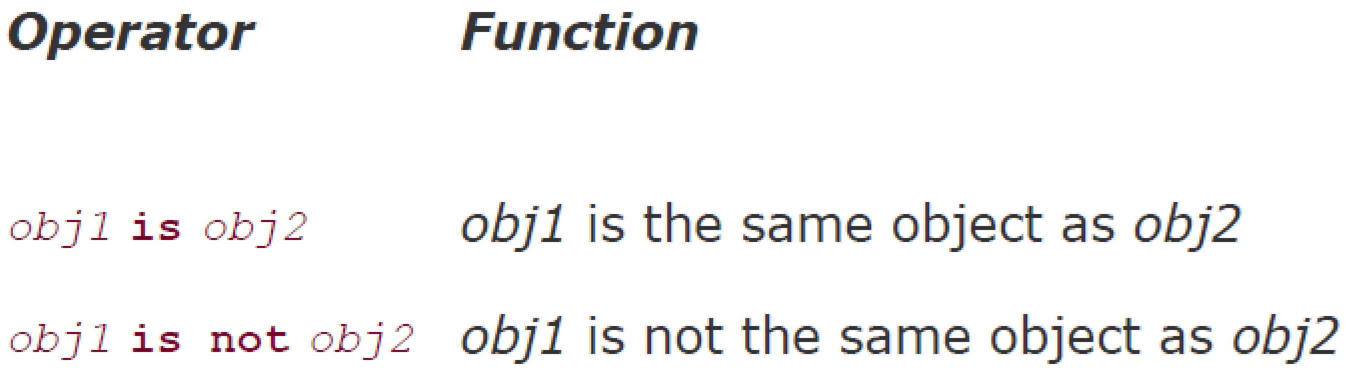
Operator	*Function*
`expr1 < expr2`	*expr1* is less than *expr2*
`expr1 > expr2`	*expr1* is greater than *expr2*
`expr1 <= expr2`	*expr1* is less than or equal to *expr2*
`expr1 >= expr2`	*expr1* is greater than or equal to *expr2*
`expr1 == expr2`	*expr1* is equal to *expr2*
`expr1 != expr2`	*expr1* is not equal to *expr2* (C-style)
`expr1 <> expr2`	*expr1* is not equal to *expr2* (ABC/Pascal-style)



Standard Type Boolean Operators

Operator	*Function*
not *expr*	Logical NOT of *expr* (negation)
expr1 **and** *expr2*	Logical AND of *expr1* and *expr2* (conjunction)
expr1 **or***expr2*	Logical OR of *expr1* and *expr2* (disjunction)

Table 4.2. Standard Type Object Identity Comparison Operators



Operator	***Function***
`obj1 is obj2`	*obj1* is the same object as *obj2*
`obj1 is not obj2`	*obj1* is not the same object as *obj2*

Table 4.4. Standard Type Built-in Functions

Function	***Operation***
`cmp(obj1, obj2)`	Compares *obj1* and *obj2*, returns integer *i* where: *i* < 0 if *obj1* < *obj2* *i* > 0 if *obj1* > *obj2* *i* == 0 if *obj1* == *obj2*
`repr(obj)` or `` `obj` ``	Returns evaluatable string representation of *obj*
`str(obj)`	Returns printable string representation of *obj*
`type(obj)`	Determines type of *obj* and return type object

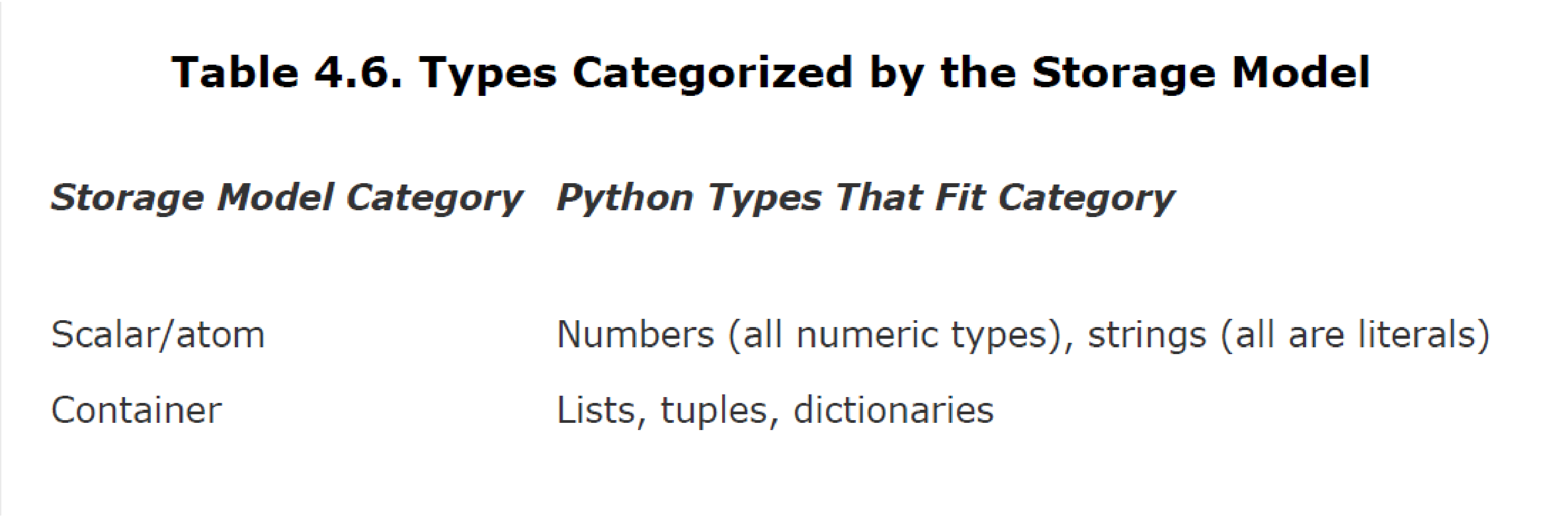


Table 4.6. Types Categorized by the Storage Model

Storage Model Category	*Python Types That Fit Category*
Scalar/atom	Numbers (all numeric types), strings (all are literals)
Container	Lists, tuples, dictionaries

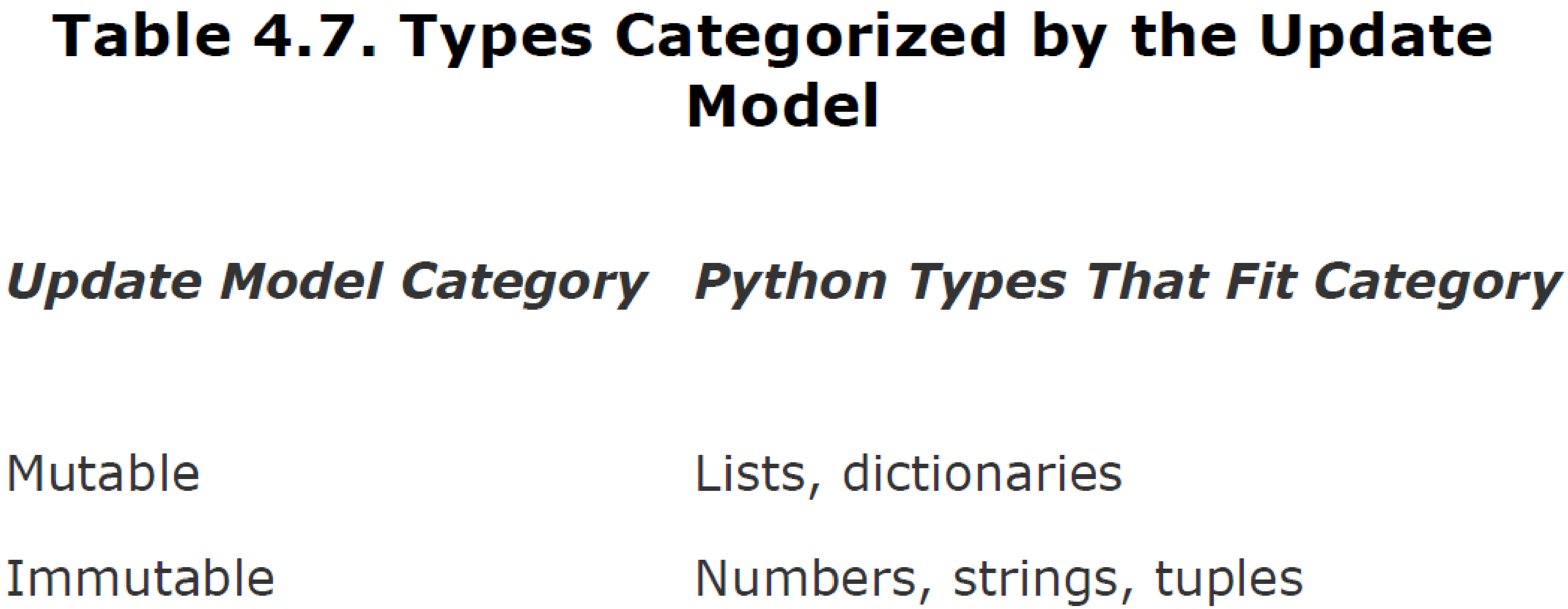
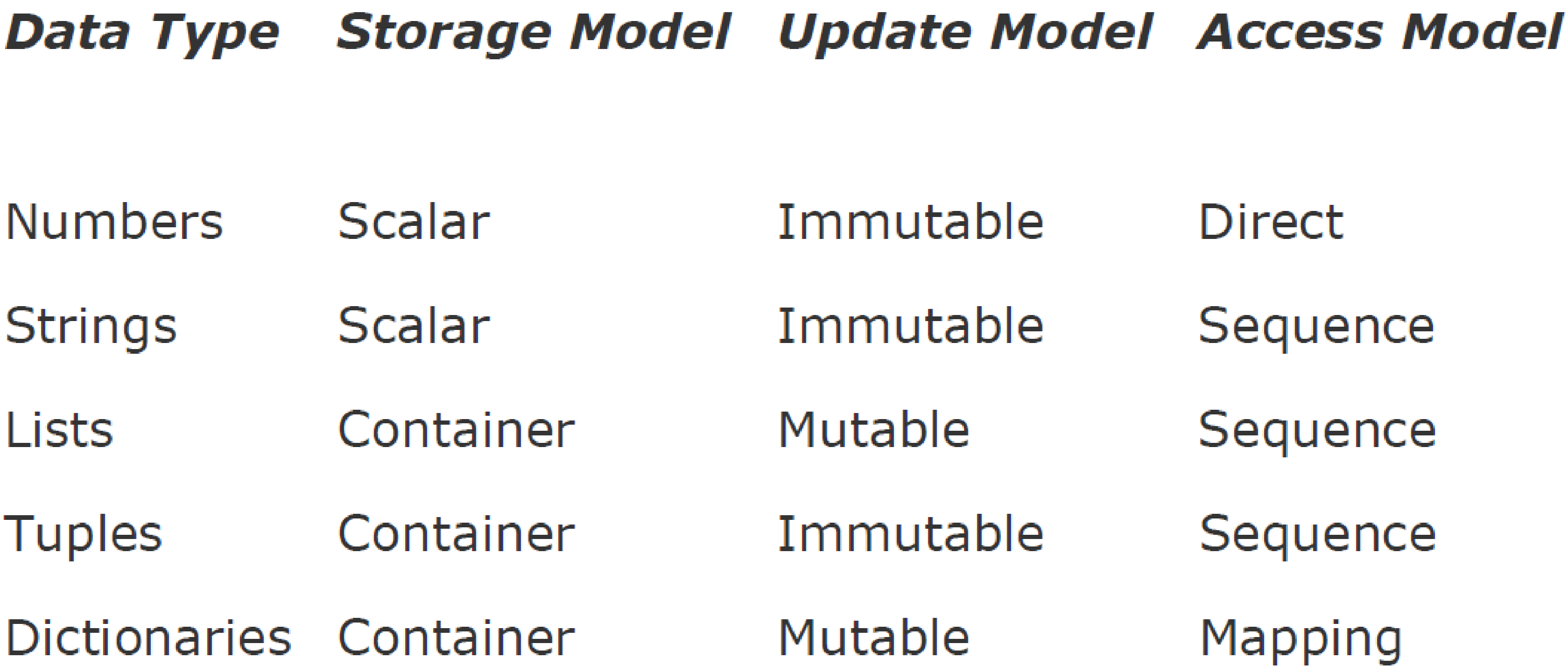


Table 4.7. Types Categorized by the Update Model

Update Model Category	*Python Types That Fit Category*
Mutable	Lists, dictionaries
Immutable	Numbers, strings, tuples

Table 4.8. Types Categorized by the Access Model

Access Model Category	*Types That Fit Category*
Direct	Numbers
Sequence	Strings, lists, tuples
Mapping	Dictionaries



Data Type	***Storage Model***	***Update Model***	***Access Model***
Numbers	Scalar	Immutable	Direct
Strings	Scalar	Immutable	Sequence
Lists	Container	Mutable	Sequence
Tuples	Container	Immutable	Sequence
Dictionaries	Container	Mutable	Mapping

Unsupported Types

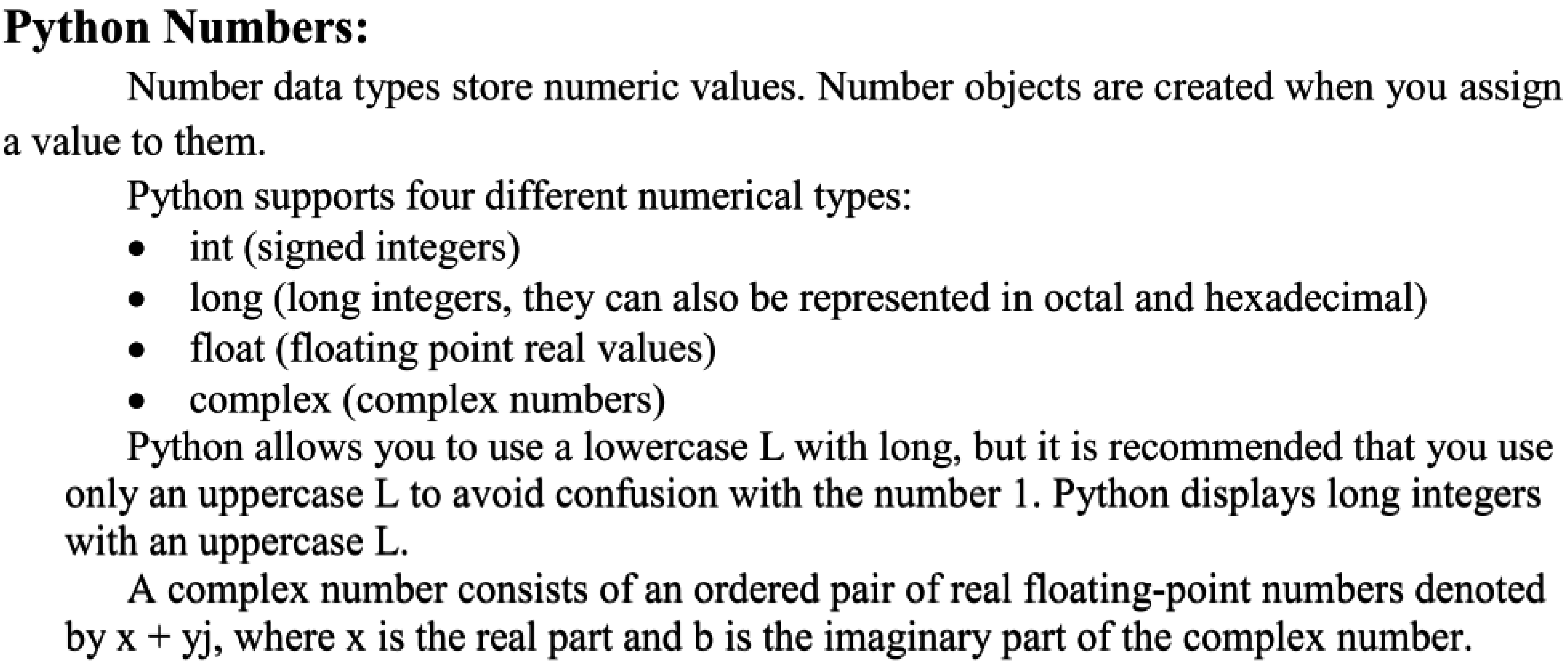
list of types that are not supported by Python.

`char` or `byte`

`int` versus `short` versus `long`

`float` versus `double`

Numbers - Introduction to Numbers Integers, Floating Point Real Numbers ,Complex Numbers



Python Numbers:

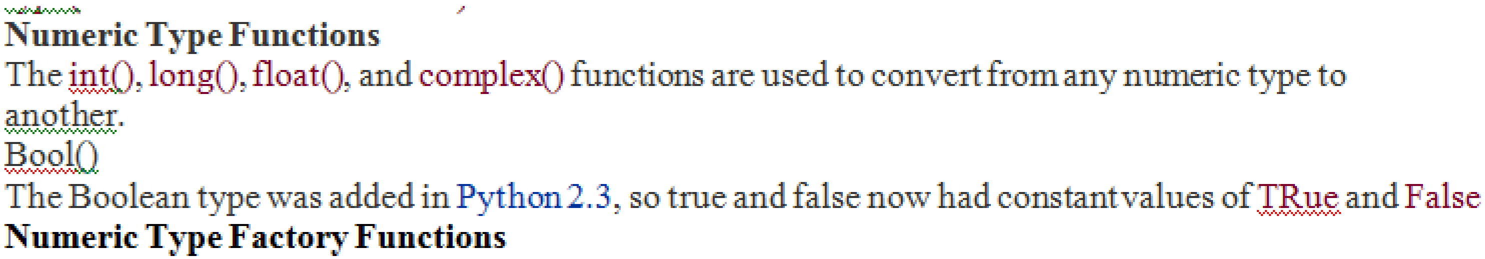
Number data types store numeric values. Number objects are created when you assign a value to them.

Python supports four different numerical types:

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
- complex (complex numbers)

Python allows you to use a lowercase L with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.

A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x is the real part and b is the imaginary part of the complex number.



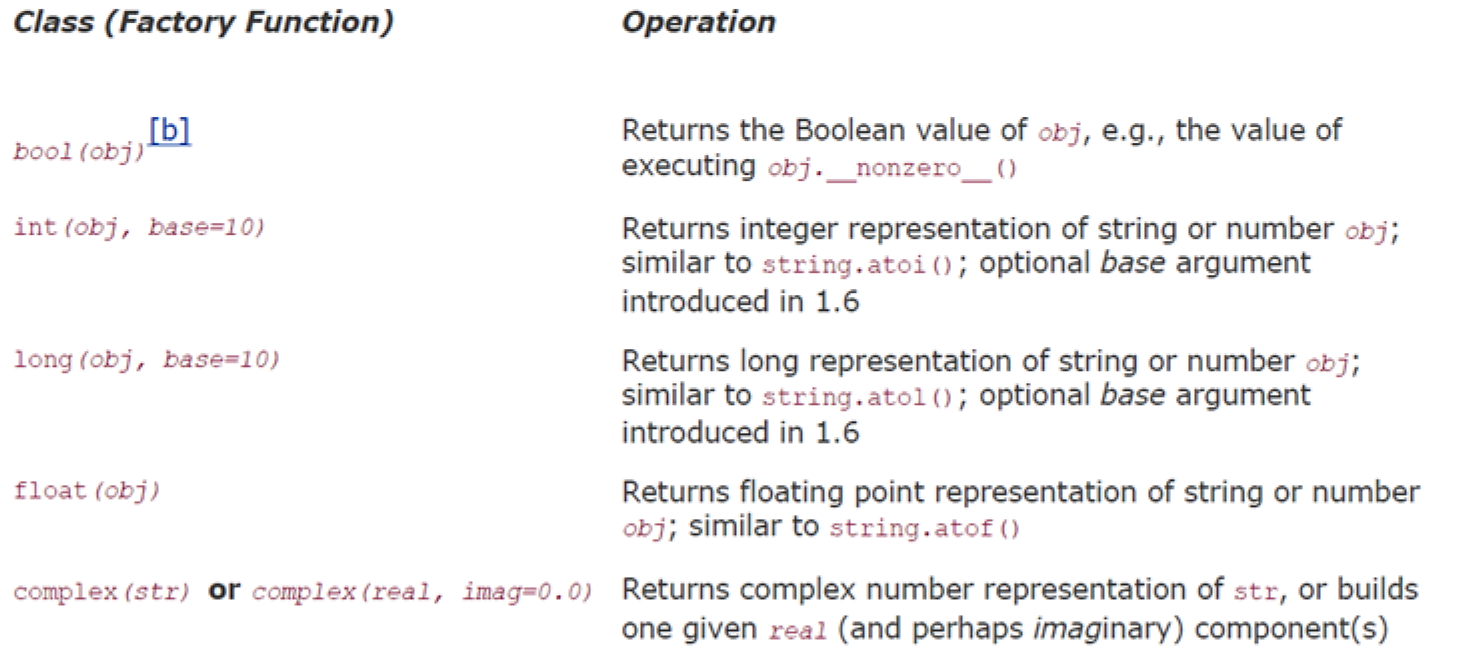
Numeric Type Functions

The int(), long(), float(), and complex() functions are used to convert from any numeric type to another.

Bool()

The Boolean type was added in Python 2.3, so true and false now had constant values of TRue and False

Numeric Type Factory Functions



Class (Factory Function)	*Operation*
`bool(obj)`[b]	Returns the Boolean value of `obj`, e.g., the value of executing `obj.__nonzero__()`
`int(obj, base=10)`	Returns integer representation of string or number `obj`; similar to `string.atoi()`; optional *base* argument introduced in 1.6
`long(obj, base=10)`	Returns long representation of string or number `obj`; similar to `string.atol()`; optional *base* argument introduced in 1.6
`float(obj)`	Returns floating point representation of string or number `obj`; similar to `string.atof()`
`complex(str)` **or** `complex(real, imag=0.0)`	Returns complex number representation of `str`, or builds one given `real` (and perhaps *imag*inary) component(s)

Examples on Numbers::

Example for integers:

```
a=90;
b=2;
C=a>>2;
Print(c)
Print(type(c))
***
```

Example double numbers:

```
 a=9.0;
B=4.7
C=a/b
Print(c)
Print(type(c))

**
```

Example1 on Complex number:

```
a=7+8j
Print(a)
Print(type(a))
```

Example2 on Complex number:

```
a=complex("3+2j")
print (a)
print("a is",a)
```

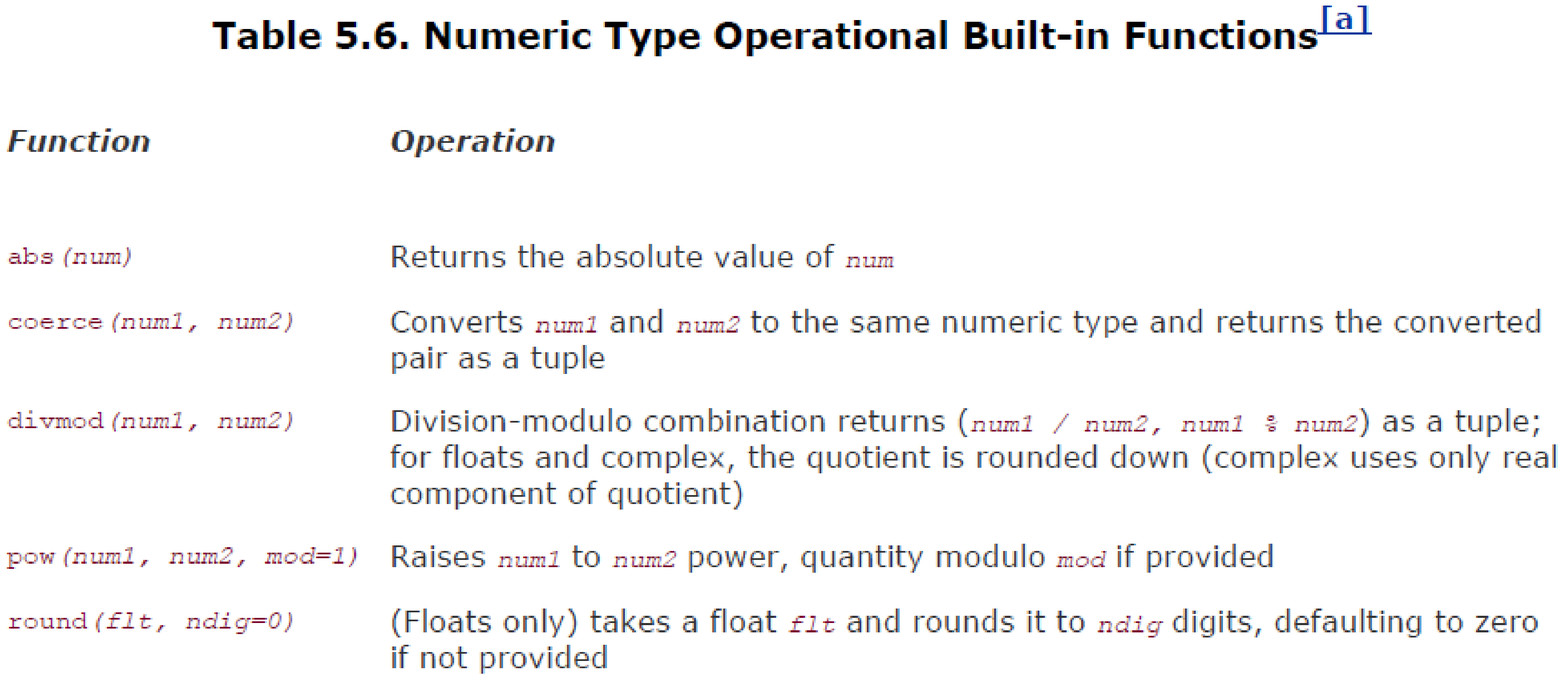


Table 5.6. Numeric Type Operational Built-in Functions[a]

Function	*Operation*
`abs(num)`	Returns the absolute value of *num*
`coerce(num1, num2)`	Converts *num1* and *num2* to the same numeric type and returns the converted pair as a tuple
`divmod(num1, num2)`	Division-modulo combination returns (*num1 / num2, num1 % num2*) as a tuple; for floats and complex, the quotient is rounded down (complex uses only real component of quotient)
`pow(num1, num2, mod=1)`	Raises *num1* to *num2* power, quantity modulo *mod* if provided
`round(flt, ndig=0)`	(Floats only) takes a float *flt* and rounds it to *ndig* digits, defaulting to zero if not provided

Standard Type Functions

```
cmp(-6, 2)
str(0xFF)
str(55.3e2)
type(0xFF)
type(98765432109876543210L)
```

Sequences:

Sequence types all share the same access model. ordered set with sequentially indexed offsets to get to each elemen
Multiple elements may be selected by using the slice operators

Figure 6.1. How sequence elements are stored and accessed

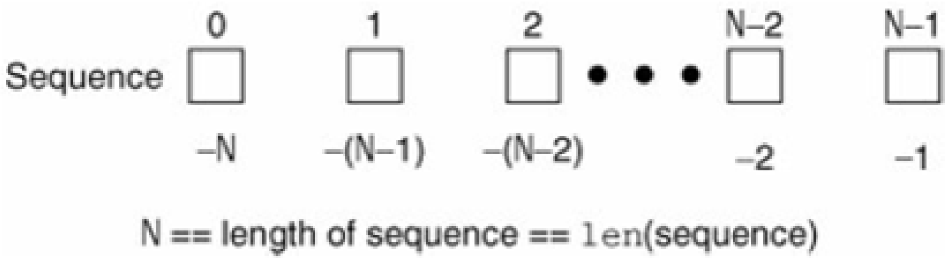
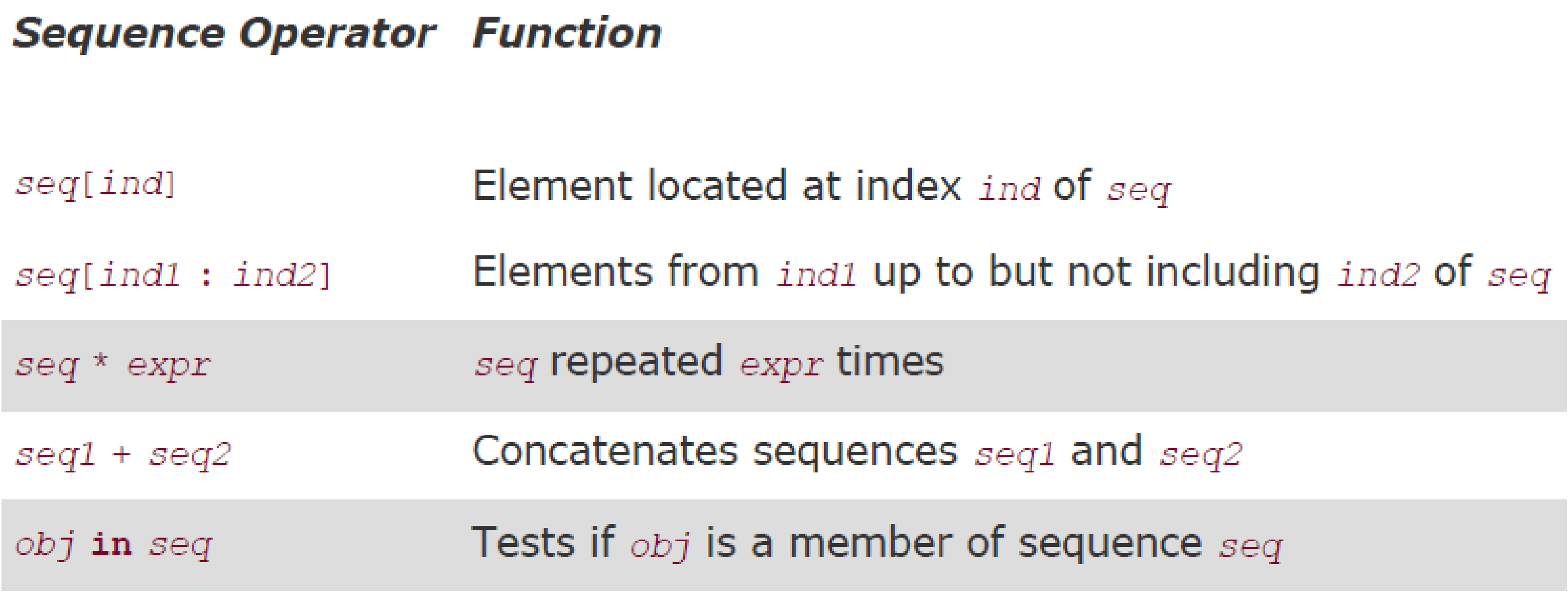
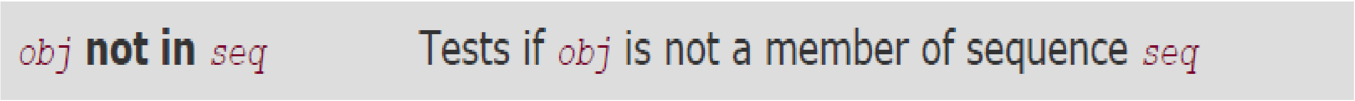


Table 6.1. Sequence Type Operators



Sequence Operator	***Function***
seq[*ind*]	Element located at index *ind* of *seq*
seq[*ind1* : *ind2*]	Elements from *ind1* up to but not including *ind2* of *seq*
seq * *expr*	*seq* repeated *expr* times
seq1 + *seq2*	Concatenates sequences *seq1* and *seq2*
obj **in** *seq*	Tests if *obj* is a member of sequence *seq*



obj **not in** *seq*	Tests if *obj* is not a member of sequence *seq*

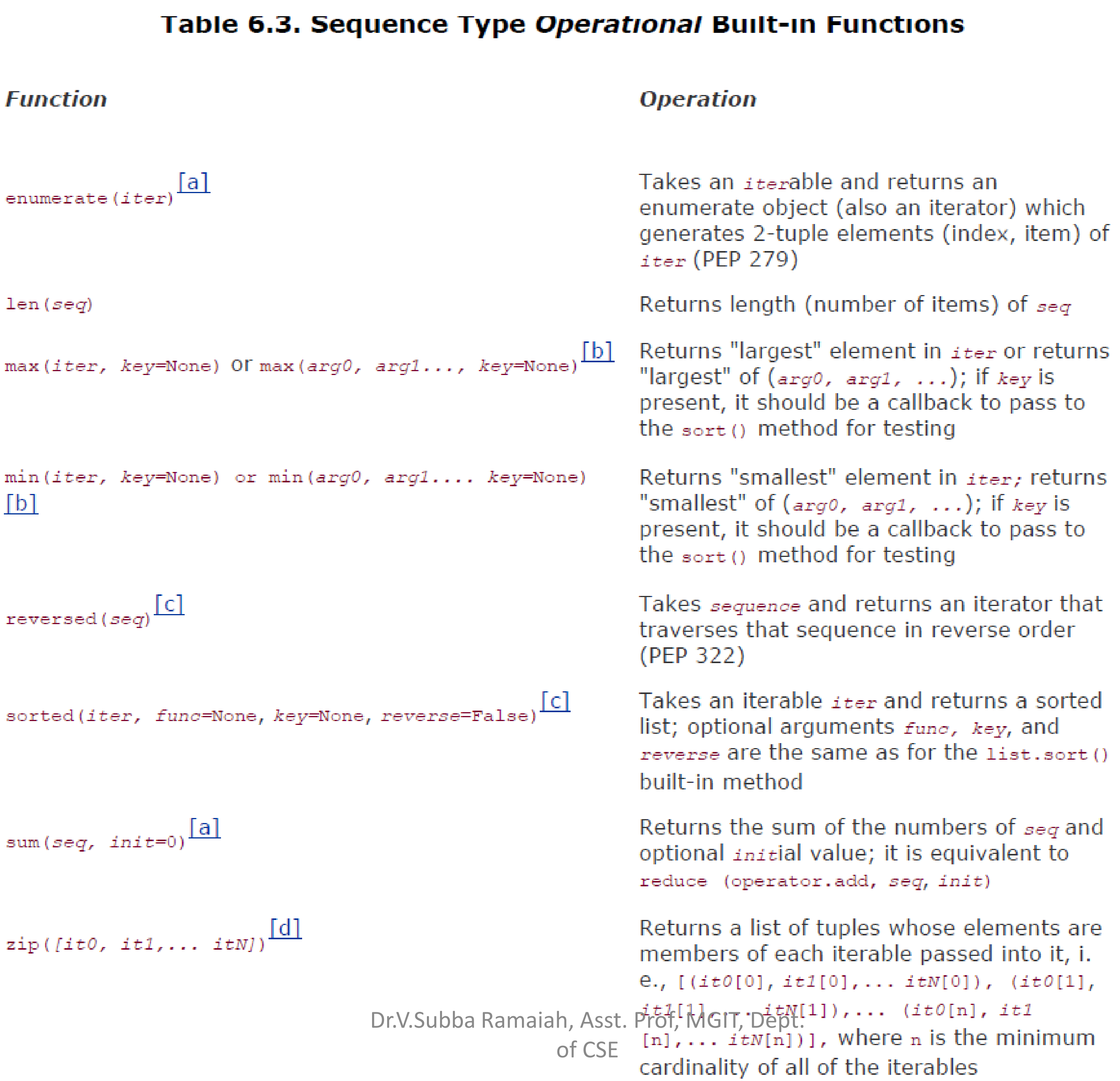
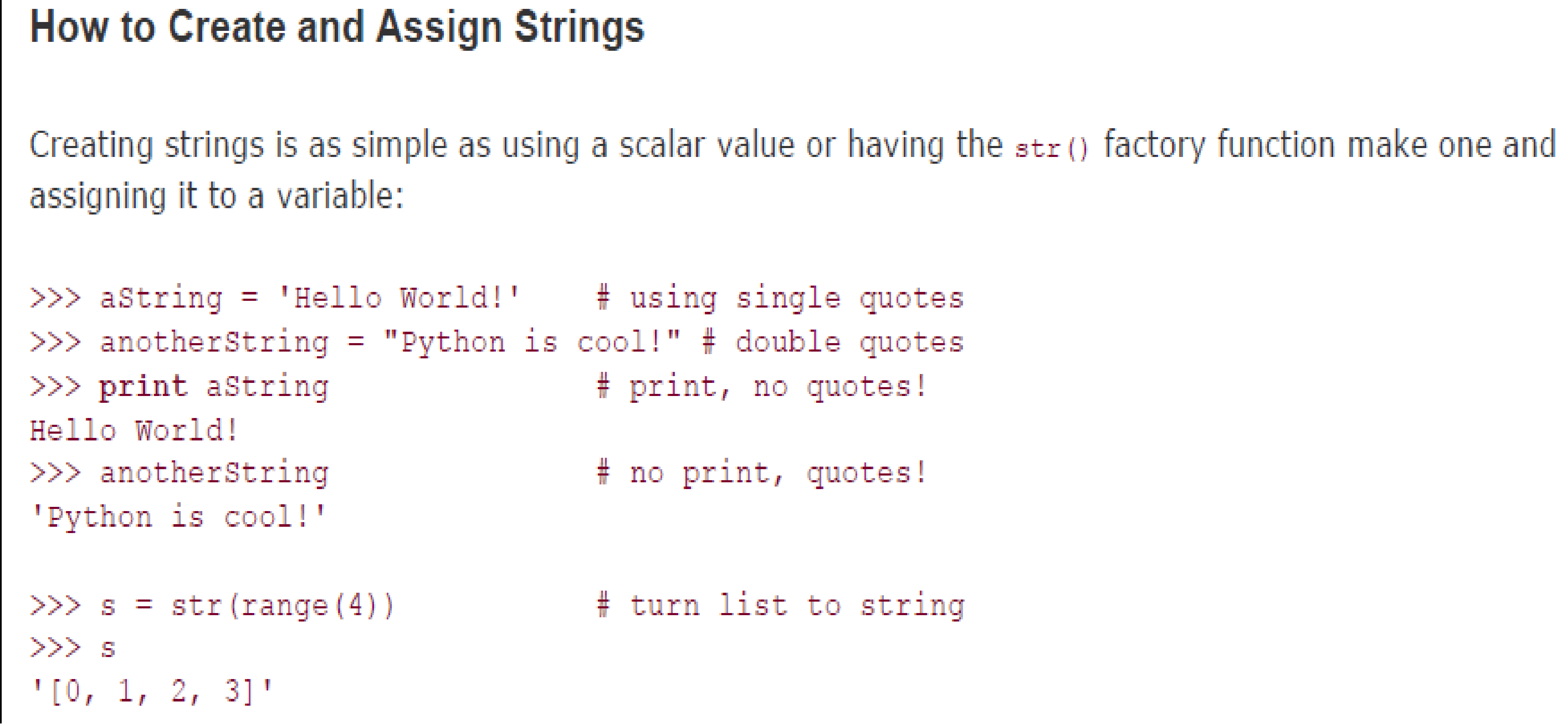


Table 6.3. Sequence Type *Operational* Built-in Functions

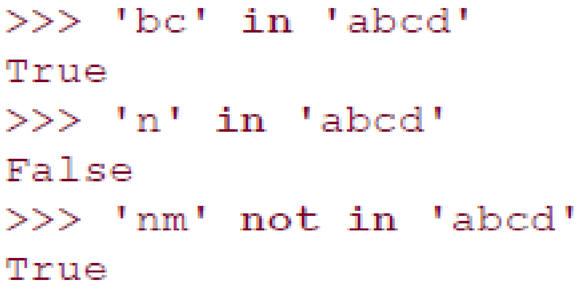
Function	*Operation*
`enumerate(iter)`[a]	Takes an *iter*able and returns an enumerate object (also an iterator) which generates 2-tuple elements (index, item) of *iter* (PEP 279)
`len(seq)`	Returns length (number of items) of *seq*
`max(iter, key=None)` or `max(arg0, arg1..., key=None)`[b]	Returns "largest" element in *iter* or returns "largest" of (*arg0, arg1, ...*); if *key* is present, it should be a callback to pass to the `sort()` method for testing
`min(iter, key=None) or min(arg0, arg1.... key=None)`[b]	Returns "smallest" element in *iter;* returns "smallest" of (*arg0, arg1, ...*); if *key* is present, it should be a callback to pass to the `sort()` method for testing
`reversed(seq)`[c]	Takes *sequence* and returns an iterator that traverses that sequence in reverse order (PEP 322)
`sorted(iter, func=None, key=None, reverse=False)`[c]	Takes an iterable *iter* and returns a sorted list; optional arguments *func, key*, and *reverse* are the same as for the `list.sort()` built-in method
`sum(seq, init=0)`[a]	Returns the sum of the numbers of *seq* and optional *init*ial value; it is equivalent to `reduce (operator.add, seq, init)`
`zip([it0, it1,... itN])`[d]	Returns a list of tuples whose elements are members of each iterable passed into it, i. e., `[(it0[0], it1[0],... itN[0]), (it0[1], it1[1],... itN[1]),... (it0[n], it1[n],... itN[n])]`, where `n` is the minimum cardinality of all of the iterables



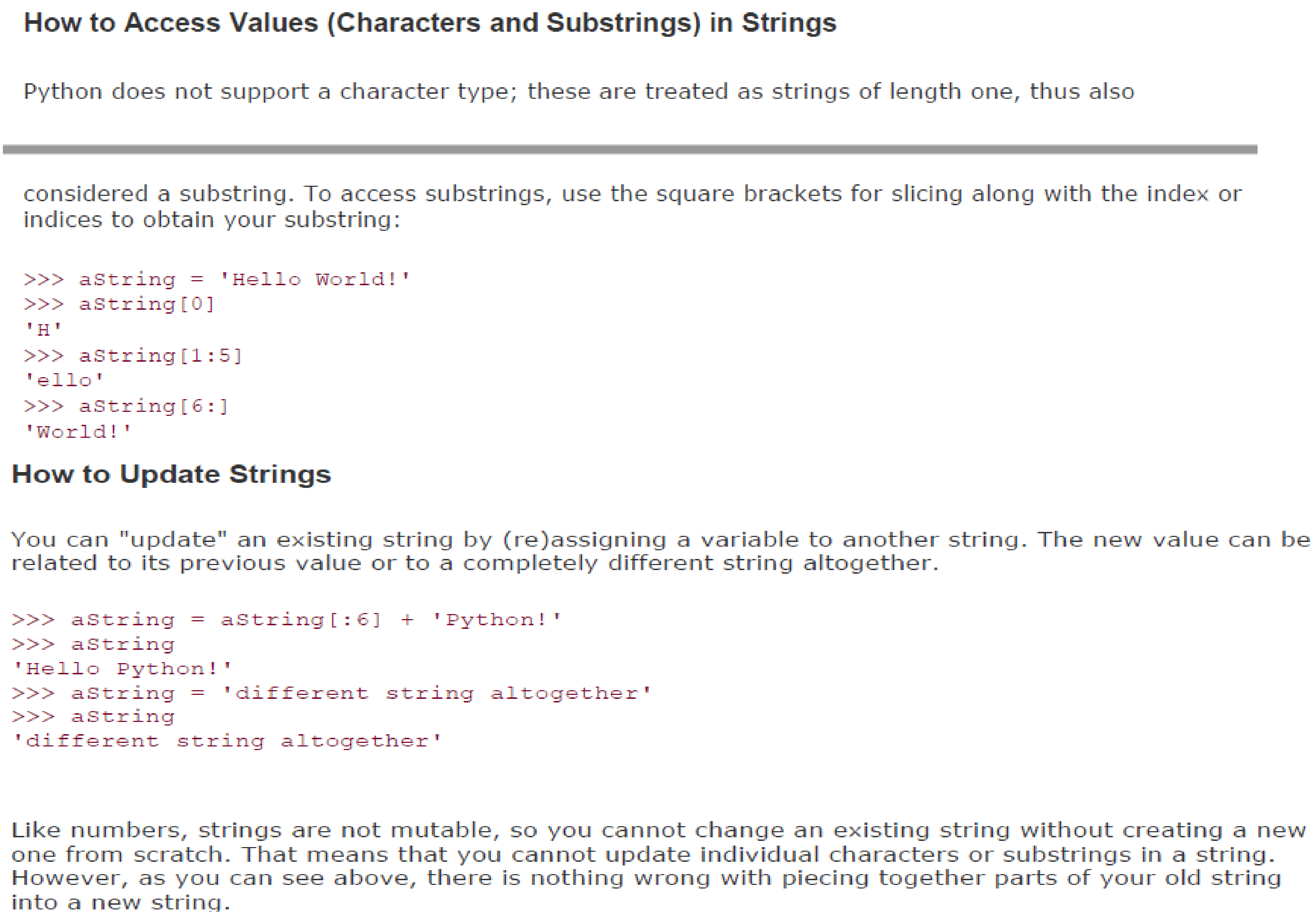
How to Create and Assign Strings

Creating strings is as simple as using a scalar value or having the `str()` factory function make one and assigning it to a variable:

```
>>> aString = 'Hello World!'    # using single quotes
>>> anotherString = "Python is cool!" # double quotes
>>> print aString               # print, no quotes!
Hello World!
>>> anotherString               # no print, quotes!
'Python is cool!'

>>> s = str(range(4))          # turn list to string
>>> s
'[0, 1, 2, 3]'
```



```
>>> 'bc' in 'abcd'
True
>>> 'n' in 'abcd'
False
>>> 'nm' not in 'abcd'
True
```



How to Access Values (Characters and Substrings) in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring. To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring:

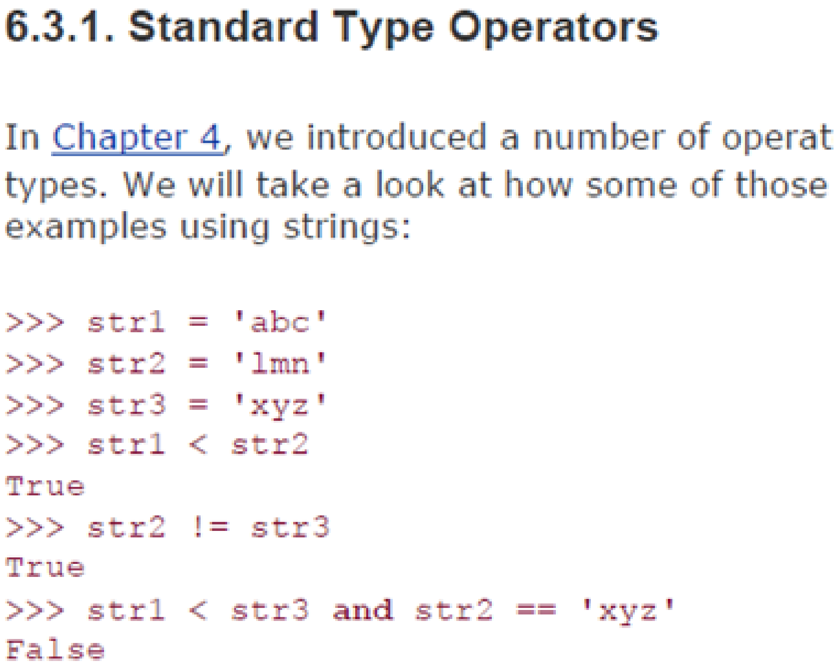
```
>>> aString = 'Hello World!'
>>> aString[0]
'H'
>>> aString[1:5]
'ello'
>>> aString[6:]
'World!'
```

How to Update Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether.

```
>>> aString = aString[:6] + 'Python!'
>>> aString
'Hello Python!'
>>> aString = 'different string altogether'
>>> aString
'different string altogether'
```

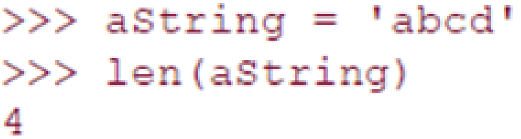
Like numbers, strings are not mutable, so you cannot change an existing string without creating a new one from scratch. That means that you cannot update individual characters or substrings in a string. However, as you can see above, there is nothing wrong with piecing together parts of your old string into a new string.



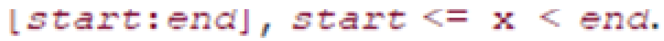
6.3.1. Standard Type Operators

In Chapter 4, we introduced a number of operat
types. We will take a look at how some of those
examples using strings:

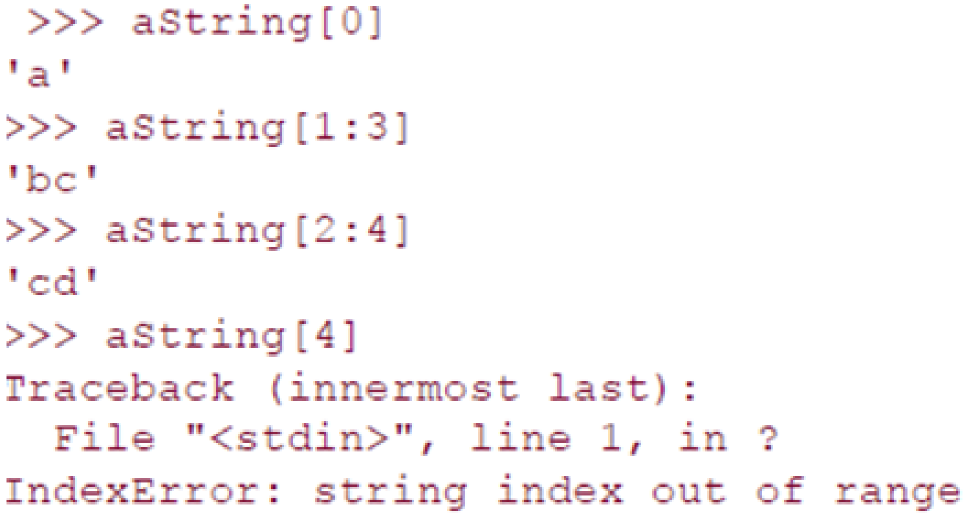
```
>>> str1 = 'abc'
>>> str2 = 'lmn'
>>> str3 = 'xyz'
>>> str1 < str2
True
>>> str2 != str3
True
>>> str1 < str3 and str2 == 'xyz'
False
```



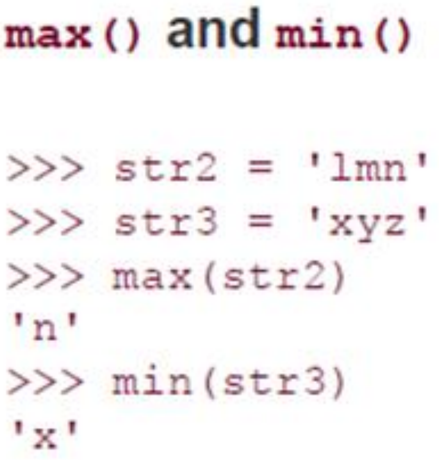
```
>>> aString = 'abcd'
>>> len(aString)
4
```



[start:end], start <= x < end.

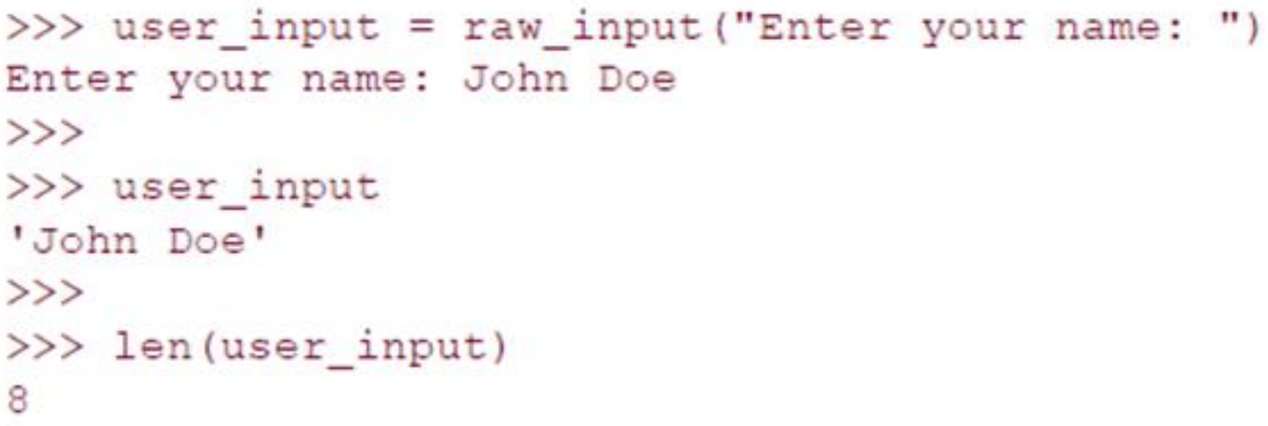


```
 >>> aString[0]
'a'
>>> aString[1:3]
'bc'
>>> aString[2:4]
'cd'
>>> aString[4]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```



max() and min()

```
>>> str2 = 'lmn'
>>> str3 = 'xyz'
>>> max(str2)
'n'
>>> min(str3)
'x'
```



```
>>> user_input = raw_input("Enter your name: ")
Enter your name: John Doe
>>>
>>> user_input
'John Doe'
>>>
>>> len(user_input)
8
```

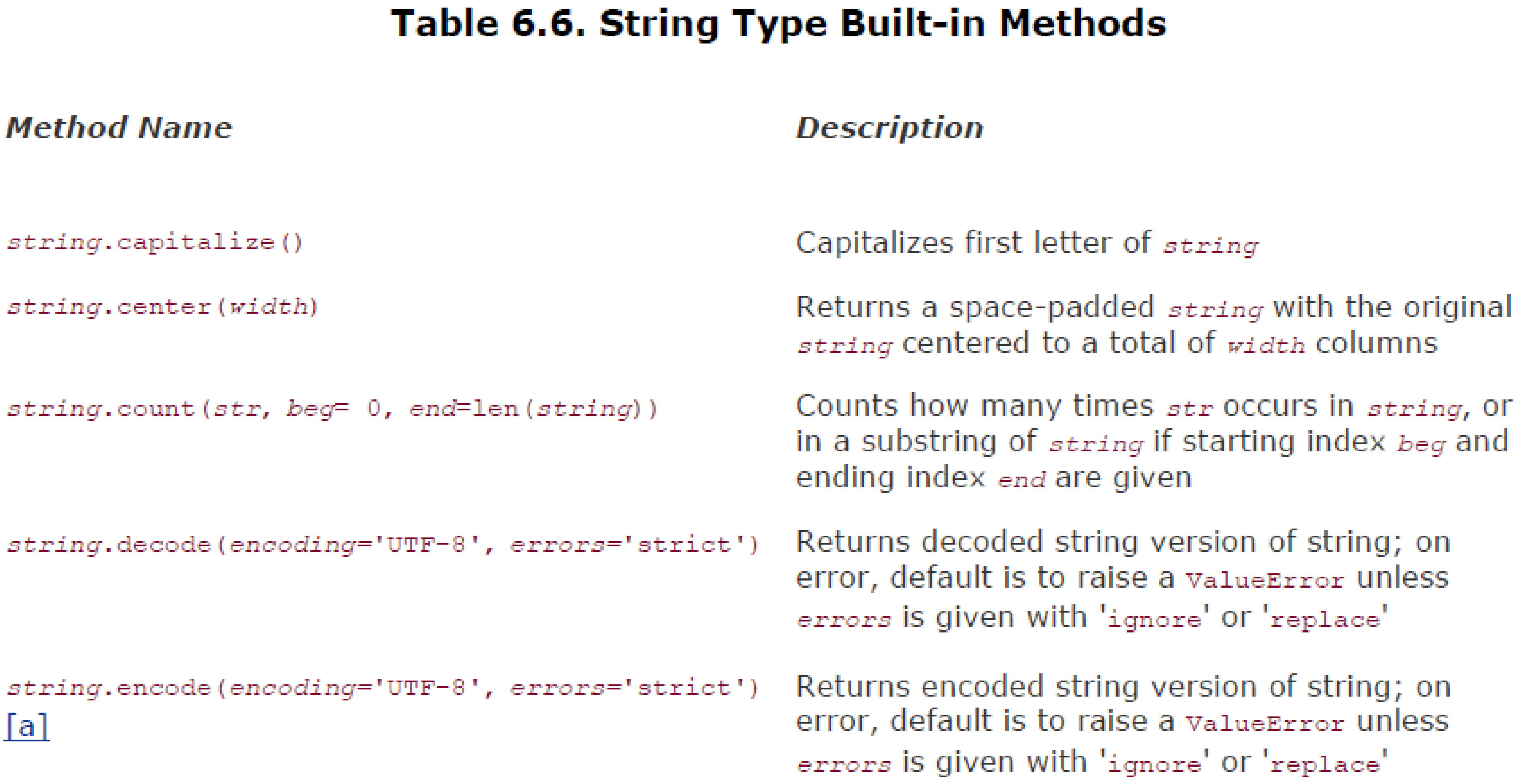
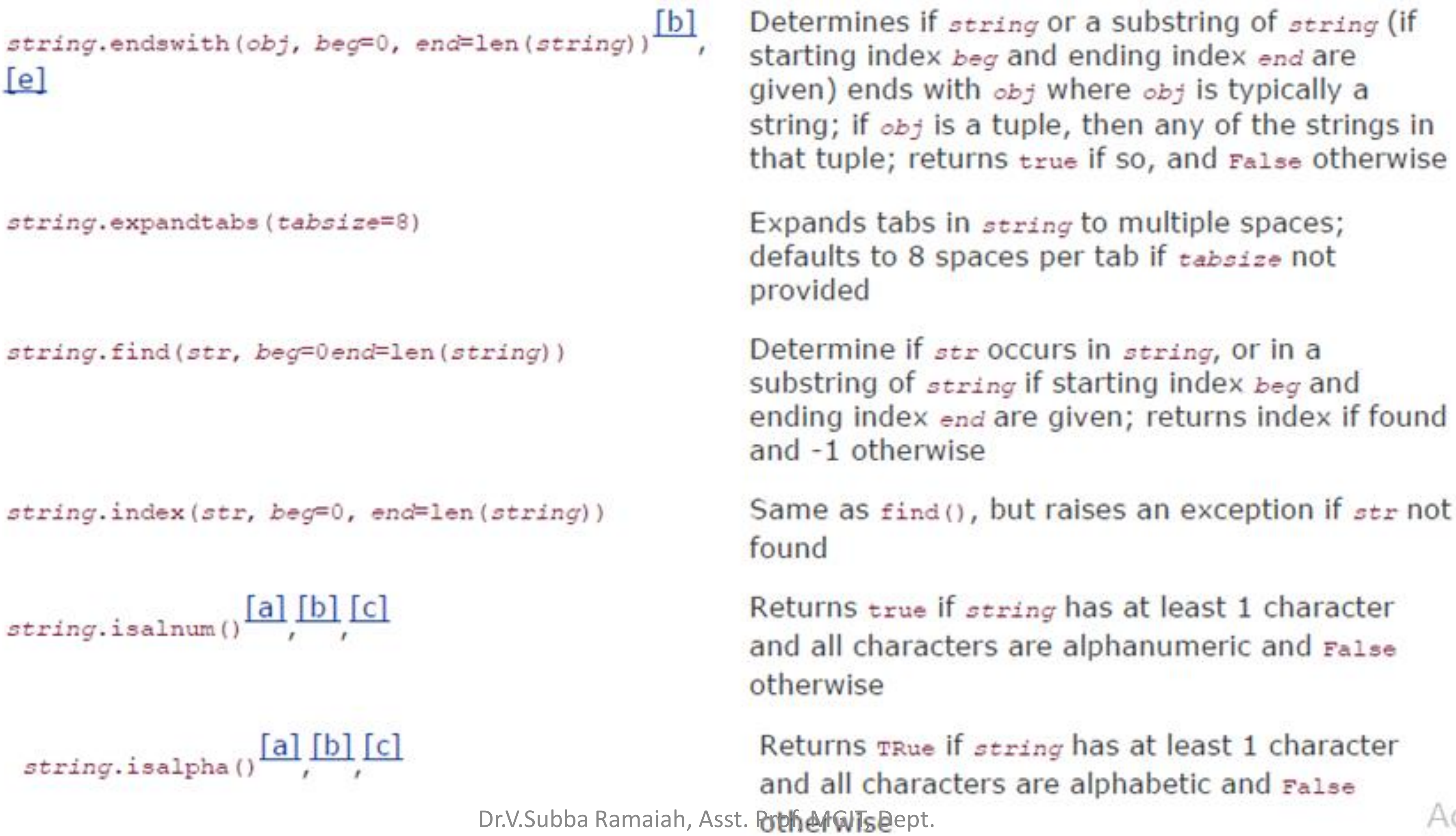


Table 6.6. String Type Built-in Methods

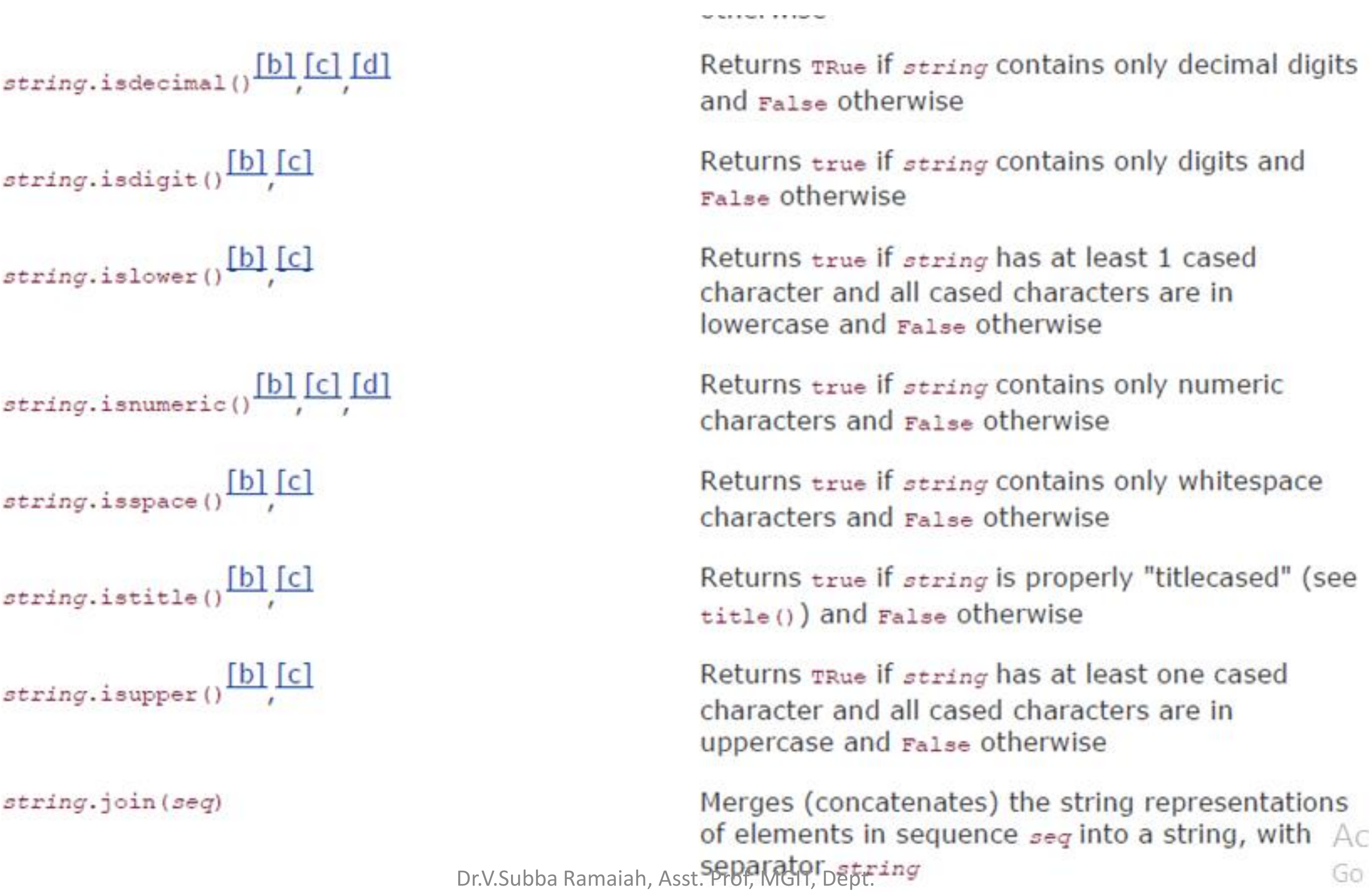
Method Name	*Description*
string.capitalize()	Capitalizes first letter of *string*
string.center(*width*)	Returns a space-padded *string* with the original *string* centered to a total of *width* columns
string.count(*str*, *beg*= 0, *end*=len(*string*))	Counts how many times *str* occurs in *string*, or in a substring of *string* if starting index *beg* and ending index *end* are given
string.decode(*encoding*='UTF-8', *errors*='strict')	Returns decoded string version of string; on error, default is to raise a ValueError unless *errors* is given with 'ignore' or 'replace'
string.encode(*encoding*='UTF-8', *errors*='strict') [a]	Returns encoded string version of string; on error, default is to raise a ValueError unless *errors* is given with 'ignore' or 'replace'

Built in Methods on Strings cont..



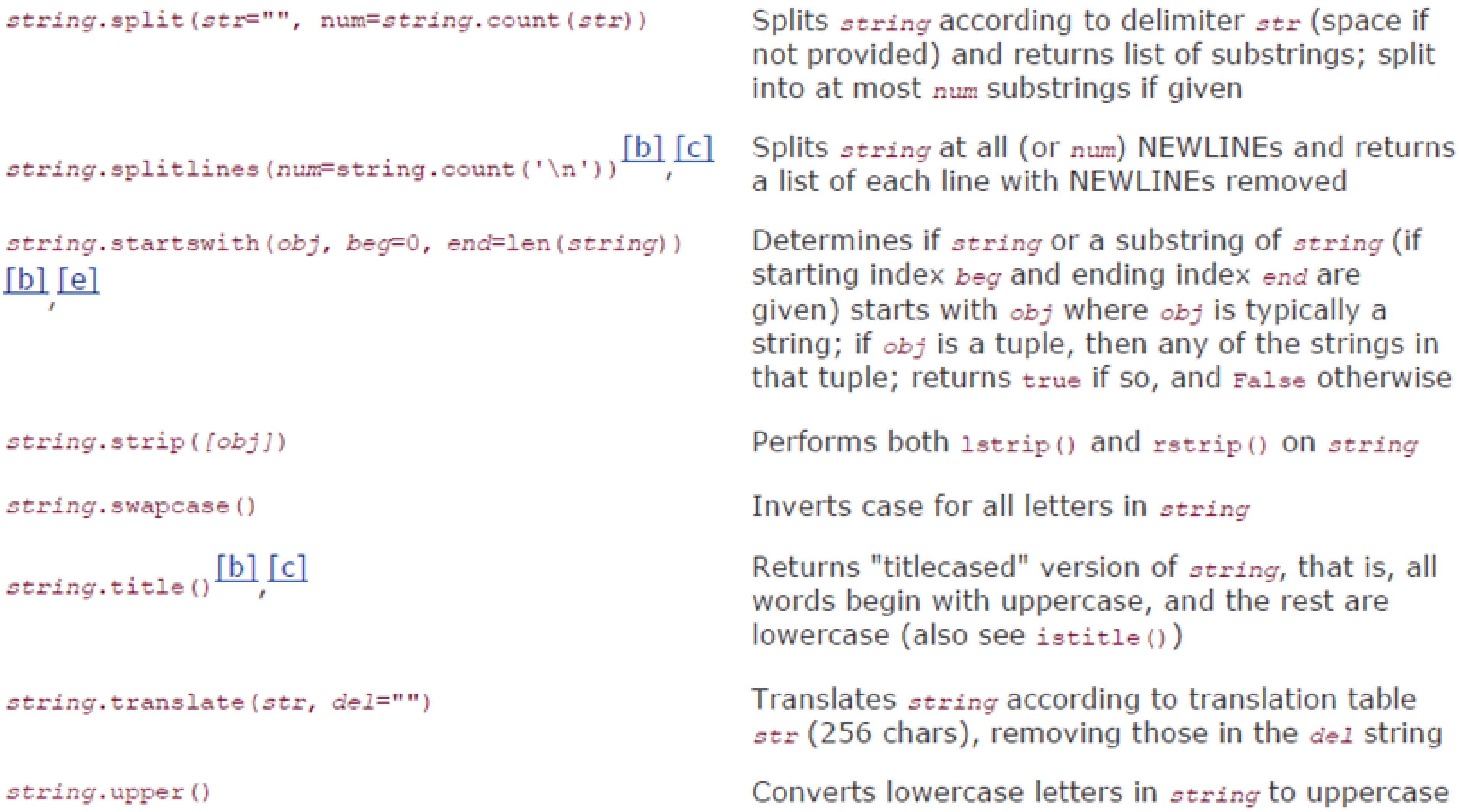
`string.endswith(obj, beg=0, end=len(string))`[b], [e]	Determines if `string` or a substring of `string` (if starting index `beg` and ending index `end` are given) ends with `obj` where `obj` is typically a string; if `obj` is a tuple, then any of the strings in that tuple; returns `true` if so, and `False` otherwise
`string.expandtabs(tabsize=8)`	Expands tabs in `string` to multiple spaces; defaults to 8 spaces per tab if `tabsize` not provided
`string.find(str, beg=0end=len(string))`	Determine if `str` occurs in `string`, or in a substring of `string` if starting index `beg` and ending index `end` are given; returns index if found and -1 otherwise
`string.index(str, beg=0, end=len(string))`	Same as `find()`, but raises an exception if `str` not found
`string.isalnum()`[a], [b], [c]	Returns `true` if `string` has at least 1 character and all characters are alphanumeric and `False` otherwise
`string.isalpha()`[a], [b], [c]	Returns `TRue` if `string` has at least 1 character and all characters are alphabetic and `False` otherwise

Built in Methods on Strings cont..



	otherwise
string.isdecimal()[b], [c], [d]	Returns TRue if *string* contains only decimal digits and False otherwise
string.isdigit()[b], [c]	Returns true if *string* contains only digits and False otherwise
string.islower()[b], [c]	Returns true if *string* has at least 1 cased character and all cased characters are in lowercase and False otherwise
string.isnumeric()[b], [c], [d]	Returns true if *string* contains only numeric characters and False otherwise
string.isspace()[b], [c]	Returns true if *string* contains only whitespace characters and False otherwise
string.istitle()[b], [c]	Returns true if *string* is properly "titlecased" (see title()) and False otherwise
string.isupper()[b], [c]	Returns TRue if *string* has at least one cased character and all cased characters are in uppercase and False otherwise
string.join(*seq*)	Merges (concatenates) the string representations of elements in sequence *seq* into a string, with separator *string*

Built in Methods on Strings cont..



`string.split(str="", num=string.count(str))`	Splits *string* according to delimiter *str* (space if not provided) and returns list of substrings; split into at most *num* substrings if given
`string.splitlines(num=string.count('\n'))`[b], [c]	Splits *string* at all (or *num*) NEWLINEs and returns a list of each line with NEWLINEs removed
`string.startswith(obj, beg=0, end=len(string))`[b], [e]	Determines if *string* or a substring of *string* (if starting index *beg* and ending index *end* are given) starts with *obj* where *obj* is typically a string; if *obj* is a tuple, then any of the strings in that tuple; returns `true` if so, and `False` otherwise
`string.strip([obj])`	Performs both `lstrip()` and `rstrip()` on *string*
`string.swapcase()`	Inverts case for all letters in *string*
`string.title()`[b], [c]	Returns "titlecased" version of *string*, that is, all words begin with uppercase, and the rest are lowercase (also see `istitle()`)
`string.translate(str, del="")`	Translates *string* according to translation table *str* (256 chars), removing those in the *del* string
`string.upper()`	Converts lowercase letters in *string* to uppercase

Sequence

- A sequence is a datatype that represents a group of elements.
- The purpose of any sequence is to store and process group elements.
- In python, strings, lists, tuples and dictionaries are very important sequence datatypes.

List

- A list is similar to an array that consists of a group of elements or items.
- The Difference is ……………
- An array can store only one type of elements whereas a list can store different types of elements.
- To create a List as putting different comma-separated values between square brackets [ ].

List

- **Example:**

  Student=[556, "mothi", 84, 96, 84, 75, 84]

- To Create empty List without any elements by simply writing empty square brackets as:

  Student=[   ]

List

- **Accessing values in List:**
  - use the square brackets for slicing along with the index or indices to obtain value available at that index.

negative Indexing	**-7**	**-6**	**-5**	**-4**	**-3**	**-2**	**-1**
Positive Indexing	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Student	**556**	**“Mothi”**	**84**	**96**	**84**	**75**	**84**

List

Output:

Program:

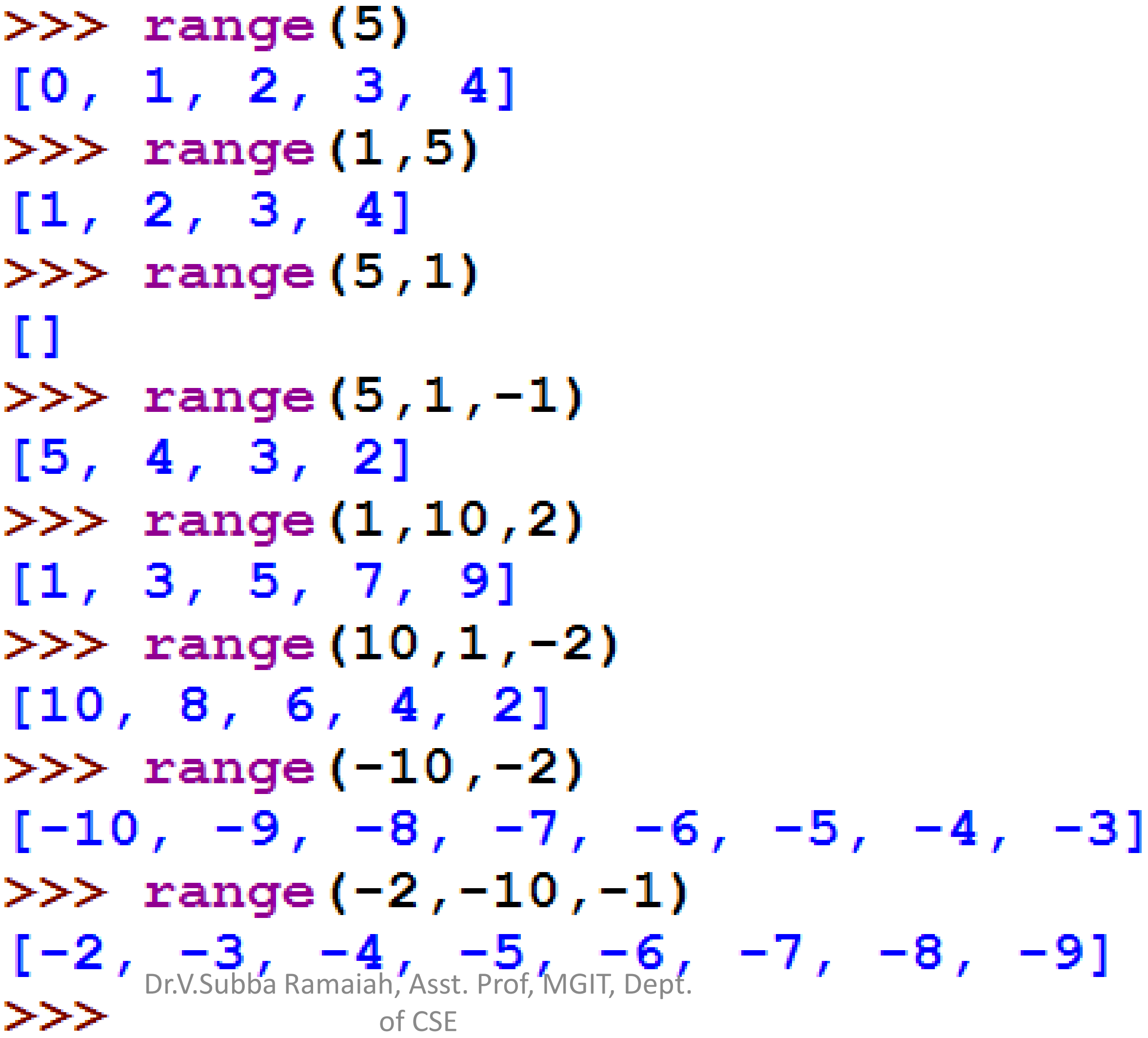
```
student = [556, “Mothi”, 84, 96, 84, 75, 84 ]
print   student                [556, "Mothi", 84, 96, 84, 75, 84]
print   student[0]             Mothi
print   student[0:2]           [556, "MOTHI"]
print   student[2: ]           [84, 96, 84, 75, 84]
print   student[ :3]           [556, "MOTHI", 84]
print   student[ : ]           [556, "MOTHI", 84, 96, 84, 75, 84]
print   student[-1]            84
print   student[-1:-7:-1]      [84, 75, 84, 96, 84, "MOTHI"]
```

range() function

- range() function used to print list of integer values.
- **Syntax:**
  - range(start, end [, step])

range() function

- **Example:**



```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(1,5)
[1, 2, 3, 4]
>>> range(5,1)
[]
>>> range(5,1,-1)
[5, 4, 3, 2]
>>> range(1,10,2)
[1, 3, 5, 7, 9]
>>> range(10,1,-2)
[10, 8, 6, 4, 2]
>>> range(-10,-2)
[-10, -9, -8, -7, -6, -5, -4, -3]
>>> range(-2,-10,-1)
[-2, -3, -4, -5, -6, -7, -8, -9]
>>>
```

List

- **Creating List using range() function:**

```
numbers=range(0,9)
print   numbers                  #[0,1,2,3,4,5,6,7,8]

numbers=range(0,9,2)
print   numbers                   #[0,2,4,6,8]
```

List

- **Looping over List:**

```
numbers=[1,2,3,4,5]
for i in numbers:
   print  i,
```

Output:

```
   1   2   3   4   5
```

List

- **Updating and Deleting List:**
  - Lists are *mutable*.
  - It means we can modify the contents of a list.
  - We can append, update or delete the elements of a list depending upon our requirements.

List

Program:

```
a = [4, 7, 6, 8, 9]
print  a
a[2] = 45
print  a
a[2:5] = 10, 11, 12
print   a
```

Output:

```
[4, 7, 6, 8, 9]

[4, 7, 45, 8, 9]

[4, 7, 10, 11, 12]
```

List

Program:

```
a = [4, 7, 6, 8, 9]
print  a
del  a[3]
print  a
```

Output:

```
[4, 7, 6, 8, 9]

[4, 7, 6, 9]
```

List

- **Concatenation of Two lists**
  - We can simply use ‘+’ operator on two lists to join them.

Program:

```
a = [4, 7, 6, 8, 9]
b = [1, 2, 3]
print   a+b
```

Output:

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

List

- **Repetition of Two lists**
  - We can repeat the elements of a list ‘n’ number of times using ‘ * ’ operator.

Program:

```
a = [4, 7, 6, 8, 9]
print   a*2
```

Output:

```
[4, 7, 6, 8,  4, 7, 6, 8]
```

List

- **Membership in lists**
  - We can check if an element is a member of a list by using ‘in’ and ‘not in’ operator.

Program:

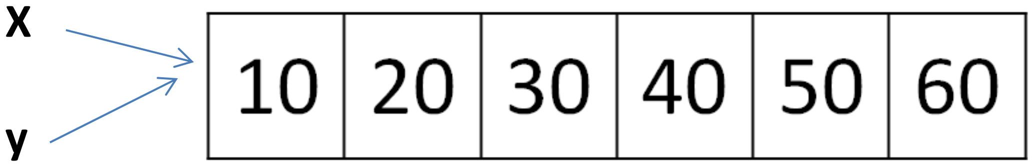
```
a = [4, 7, 6, 8, 9]
x = 7
print   x in a
y = 10
print   y not in a
```

Output:

```
True

True
```

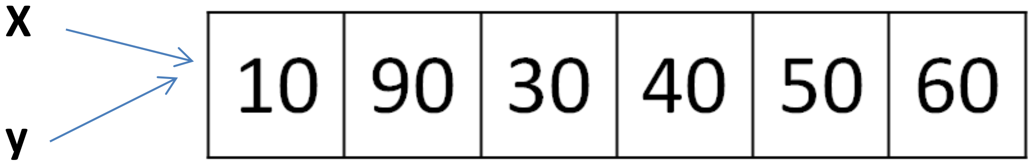
List

- **Aliasing lists**
  - Giving a new name to an existing list is called *‘aliasing’.*
  - To provide a new name to this list, we can simply use assignment operator (=).



10	20	30	40	50	60

Before Modifications



10	90	30	40	50	60

After Modifications

List

Program:

```
a = [10, 20, 30, 40, 50, 60]
x = a
print   a
print   x
a[1]= 90
print   a
print   x
```

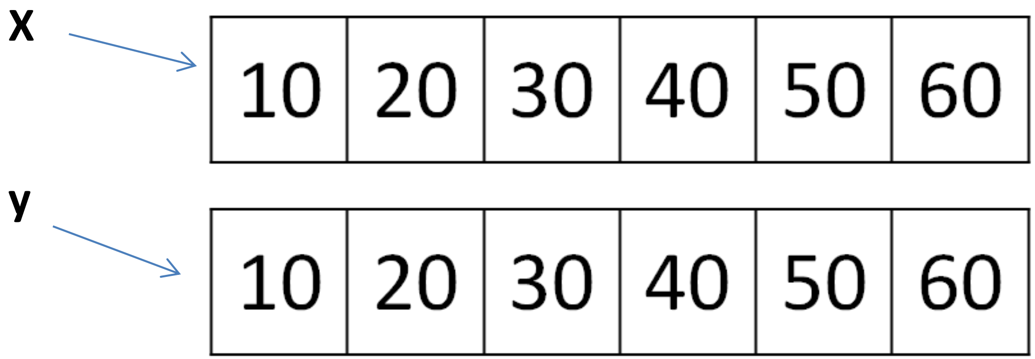
Output:

```
[10, 20, 30, 40, 50, 60]
[10, 20, 30, 40, 50, 60]

[10, 90, 30, 40, 50, 60]
[10, 90, 30, 40, 50, 60]
```

List

- **Cloning lists**
  - Obtaining exact copy of an existing object (or list) is called ‘*cloning*’.
  - To Clone a list, we can take help of the slicing operation [:].

List



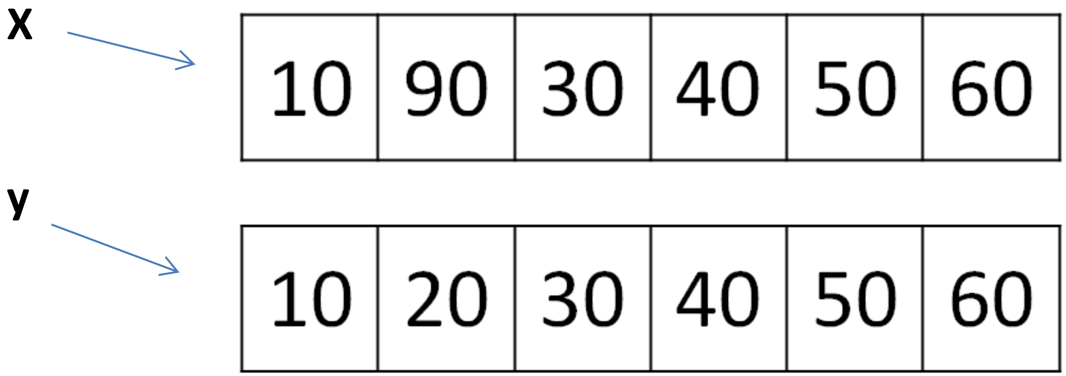
x

10	20	30	40	50	60

y

10	20	30	40	50	60

Before Modifications



x

10	90	30	40	50	60

y

10	20	30	40	50	60

After Modifications

List

Program:

```
a = [10, 20, 30, 40, 50, 60]
x = a[ : ]
print   a
print   x
a[1]= 90
print   a
print   x
```

Output:

```
[10, 20, 30, 40, 50, 60]
[10, 20, 30, 40, 50, 60]

[10, 90, 30, 40, 50, 60]
[10, 20, 30, 40, 50, 60]
```

List

Method	Description
lst.index(x)	Returns the first occurrence of x in the list.
lst.append(x)	Appends x at the end of the list.
lst.insert(i,x)	Inserts x to the list in the position specified by i.
lst.copy()	Copies all the list elements into a new list and returns it.
lst.extend(lst2)	Appends lst2 to list.
lst.count(x)	Returns number of occurrences of x in the list.
lst.remove(x)	Removes x from the list.
lst.pop()	Removes the ending element from the list.
lst.sort()	Sorts the elements of list into ascending order.
lst.reverse()	Reverses the sequence of elements in the list.
lst.clear()	Deletes all elements from the list.
max(lst)	Returns biggest element in the list.
min(lst)	Returns smallest element in the list.

List

- **Nested List:**
  - A list within another list is called a *nested list*.

Program:

```
a = [[1, 2],[3,4],[5,6]]
print   a[1]
print   a[2][1]
for i in a[2]:
   print  i,
```

Output:

```
[1,2]
6

5   6
```

List

- **List Comprehensions:**
  - List comprehensions represent creation of new lists from an iterable object that satisfy a given condition.

```
squares=[ ]
   for i in range(1,11):
           squares.append(i**2)
```

Can be rewritten as……….

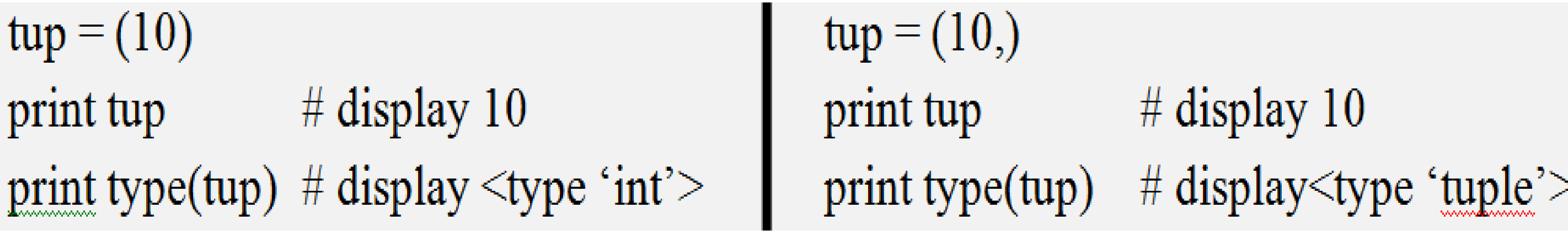
```
squares=[x**2 for x in range(1,11)]
```

Tuple

- A Tuple is a python sequence which stores a group of elements or items.
- Tuples are similar to lists but the main difference is tuples are immutable whereas lists are mutable.
- Once we create a tuple we cannot modify its elements.
- Tuples are generally used to store data which should not be modified and retrieve that data on demand.

Tuple

- Creating a tuple by writing elements separated by commas inside parentheses ( ).
  - tup = (10, 556, 22.3, “Mothi”)
- To create a tuple with only one element, we can, mention that element in parenthesis and after that a comma is needed.



```
tup = (10)
print tup         # display 10
print type(tup)   # display <type ‘int’>
```

```
tup = (10,)
print tup         # display 10
print type(tup)   # display<type ‘tuple’>
```

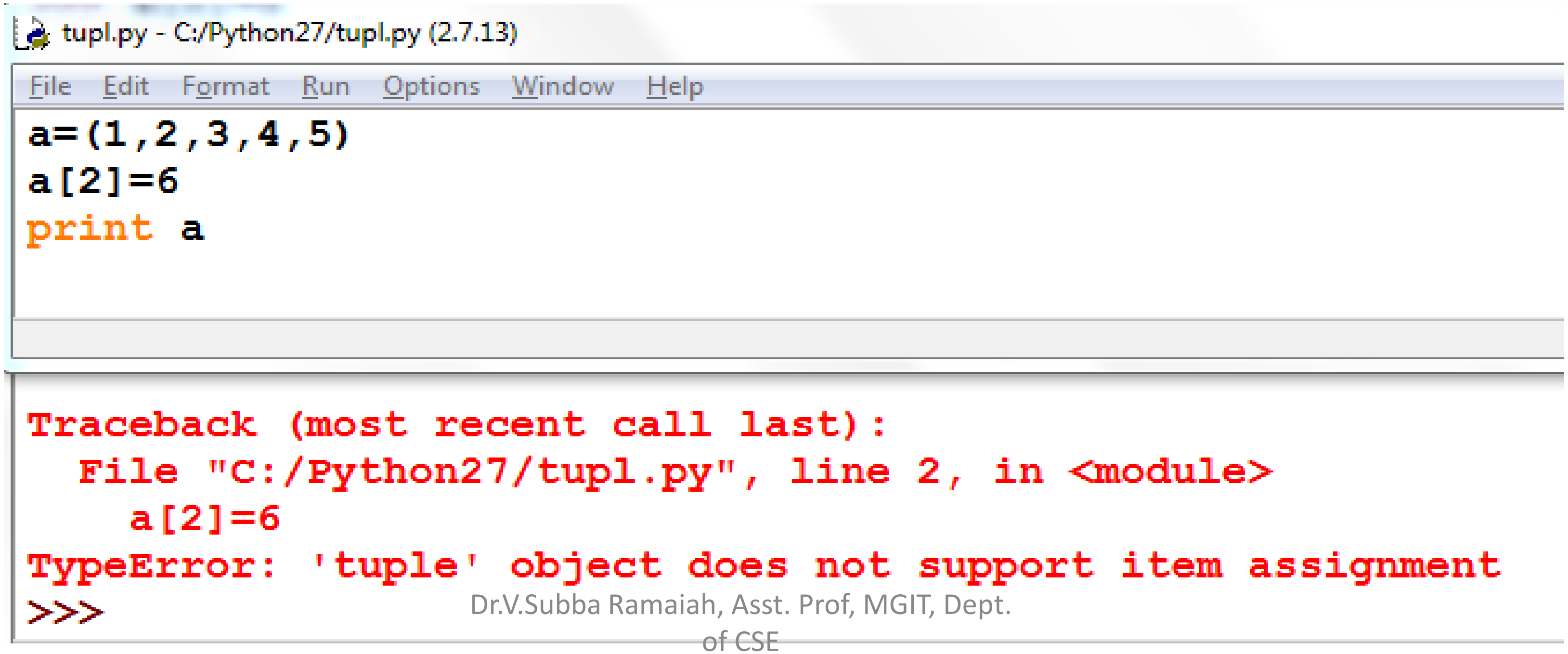
Tuple

- **Accessing values in Tuple:**
  - Accessing the elements from a tuple can be done using indexing or slicing.

```
tup = (50,60,70,80,90)
print   tup[0]            # 50
print   tup[1:4]          # (60,70,80)
print   tup[-1]           # 90
print   tup[-1:-4:-1]     # (90,80,70)
print   tup[-4:-1]        # (60,70,80)
```

Tuple

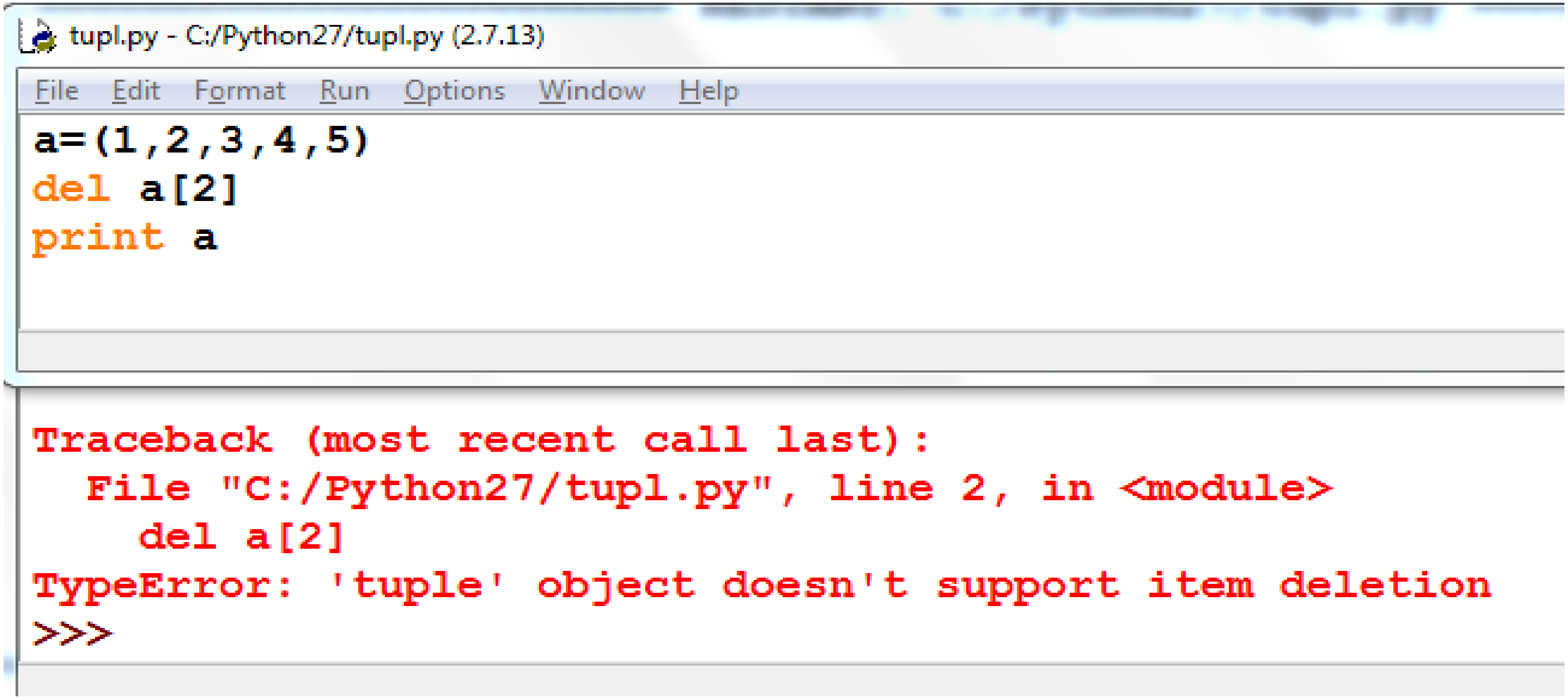
- **Updating and deleting in Tuple:**
  - Tuples are immutable which means you cannot update, change or delete the values of tuple elements.



```
a=(1,2,3,4,5)
a[2]=6
print a
```

```
Traceback (most recent call last):
  File "C:/Python27/tupl.py", line 2, in <module>
    a[2]=6
TypeError: 'tuple' object does not support item assignment
>>>
```

Tuple


tupl.py - C:/Python27/tupl.py (2.7.13)
File Edit Format Run Options Window Help

```
a=(1,2,3,4,5)
del a[2]
print a
```

```
Traceback (most recent call last):
  File "C:/Python27/tupl.py", line 2, in <module>
    del a[2]
TypeError: 'tuple' object doesn't support item deletion
>>>
```

Tuple

Operation	Description
len(t)	Return the length of tuple.
tup1+tup2	Concatenation of two tuples.
tup*n	Repetition of tuple values in n number of times.
x in tup	Return True if x is found in tuple otherwise returns False.
cmp(tup1,tup2)	Compare elements of both tuples
max(tup)	Returns the maximum value in tuple.
min(tup)	Returns the minimum value in tuple.
tuple(list)	Convert list into tuple.
tup.count(x)	Returns how many times the element ‘x’ is found in tuple.
tup.index(x)	Returns the first occurrence of the element ‘x’ in tuple. Raises ValueError if ‘x’ is not found in the tuple.
sorted(tup)	Sorts the elements of tuple into ascending order. sorted(tup,reverse=True) will sort in reverse order.

Tuple

- **Nested Tuple:**
  - A list within another list is called a *nested list*.

Program:

```
students=(("RAVI", "CSE", 92.00), ("RAMU", "ECE", 93.00),
                ("RAJA", "EEE", 87.00))
for i in students:
    print i
```

Output:

```
("RAVI", "CSE", 92.00)
("RAMU", "ECE", 93.00)
("RAJA", "EEE", 87.00)
```

Set

- Set is another data structure supported by python.
- Basically, sets are same as lists but with a difference that sets are lists with no duplicate entries.
- Technically a set is a mutable and an unordered collection of items. This means that we can easily add or remove items from it.

Set

- **Creating a set:**
  - A set is created by placing all the elements inside curly brackets { }.

```
s={1, 2.5, "abc" }
print s
```

Output:

```
set([1, 2.5, "abc" ])
```

Set

- **Converting list into a set:**
  - A set can have any number of items and they may be of different data types. *set()* function is used to converting list into set.

```
s=set([1, 2.5, "abc"])
print s
```

Output:

```
set([1, 2.5, "abc" ])
```

Set

Operation	Description
len(s)	number of elements in set *s* (cardinality)
s.issubset(t) **(or)** **s <= t**	test whether every element in *s* is in *t*
s.issuperset(t) **(or)** **s >= t**	test whether every element in *t* is in *s*
s.union(t) **(or)** **s|t**	new set with elements from both *s* and *t*
s.intersection(t) **(or)** **s & t**	new set with elements common to *s* and *t*
s.copy()	new set with a shallow copy of *s*
s.update(t)	return set s with elements added from t

Dictionary

- A dictionary represents a group of elements arranged in the form of key-value pairs. The first element is considered as ‘key’ and the immediate next element is taken as its ‘value’.
- The key and its value are separated by a colon (:). All the key-value pairs in a dictionary are inserted in curly braces { }.

Dictionary

- **Program:**

```
d= { 'Regd.No': 556, 'Name':'Mothi', 'Branch': 'CSE' }
print   d['Regd.No']                    # 556
print   d['Name']                       # Mothi
print   d['Branch']                     # CSE
```

Dictionary

- **Program:**

```
d={'Regd.No':556,'Name':'Mothi','Branch':'CSE'}
print d
d['Gender']="Male"
print d
```

Output:

```
{'Regd.No':556,'Name':'Mothi','Branch':'CSE'}
{'Gender': 'Male', 'Branch': 'CSE', 'Name': 'Mothi',
   'Regd.No': 556}
```

Dictionary

Method	Description
`d.clear()`	Removes all key-value pairs from dictionary‘d’.
`d2=d.copy()`	Copies all elements from‘d’ into a new dictionary d2.
`d.fromkeys(s [,v] )`	Create a new dictionary with keys from sequence‘s’ and values all set to ‘v’.
`d.get(k [,v] )`	Returns the value associated with key ‘k’. If key is not found, it returns ‘v’.
`d.items()`	Returns an object that contains key-value pairs of‘d’. The pairs are stored as tuples in the object.
`d.keys()`	Returns a sequence of keys from the dictionary‘d’.
`d.values()`	Returns a sequence of values from the dictionary‘d’.
`d.update(x)`	Adds all elements from dictionary ‘x’ to‘d’.
`d.pop(k [,v] )`	Removes the key ‘k’ and its value from‘d’ and returns the value. If key is not found, then the value ‘v’ is returned. If key is not found and ‘v’ is not mentioned then ‘KeyError’ is raised.
`d.setdefault(k [,v] )`	If key ‘k’ is found, its value is returned. If key is not found, then the k, v pair is stored into the dictionary‘d’.