**Intro to Python3 for CSC148 – PART1 –** Version - October 2024

Posted at: <http://athena.ecs.csus.edu/~mitchell/csc148/Python3/Python3languageElements_f24_PART1.doc>x

by W Mitchell, Sacramento State U., College of Engineering and Computer Science, Computer Science Department

This document is a focused intro to Python version 3. Typical of modern languages, simulation in Python is done with “add-on” modules, in Python’s case, called “SimPy”.

SimPy implements a process-oriented DES environment that was strongly influenced by the original “process oriented” DES language SIMULA-67 (in the late 1960s).

The small sections in this document colored BLUE draw attention to unusual and/or useful properties of Python.

**Python and simpy versions**

There are very significant differences between Python 2.x and 3.x, and between SimPy2 and SimPy3. Also, SimPy version 2 is incompatible with version 3. DES developers should use SimPy version 3 or 4 as needed. Unless stated otherwise, “Python” and “simpy” refer to Python 3.9 and SimPy3.0.10+. *Caution: be version-aware when reading web material on Python/simpy.*

**This is a short Introduction to the Python language**

Items covered here enable development of DES models using the **simpy (aka SimPy) extension to Python3**.

**Python Language Documentation**

The language’s documentation has several major sources –

1 ) In the idle Python shell, the language reference manual is available via: the idle shell window’s main menu sequence Help - - > Python Doc F1 - - > Python x.y.z - - > Language Reference. This is the definitive/authoritative source for a given version of Python, Thus, it is conveniently available directly in an idle shell session.

2 ) Another doc source is via the host OS command line: Python -m pydoc moduleName

Example: Python -m pydoc finally (for documentation about the Python ‘finally’ statement)

3 ) Future language development includes the Python “PEP” process, as outlined next.

<< … Python's development is conducted largely through the *Python Enhancement Proposal* (PEP) process, the primary mechanism for proposing major new features, collecting community input on issues and documenting Python design decisions.[[104]](https://en.wikipedia.org/wiki/Python_(programming_language)#cite_note-PepCite000-104) Outstanding PEPs are reviewed and commented on by the Python community and Guido Van Rossum, Python's [Benevolent Dictator For Life](https://en.wikipedia.org/wiki/Benevolent_Dictator_For_Life) (the language inventor) … >>

Source: https://en.wikipedia.org/wiki/Python\_(programming\_language)#Development

4 ) Full language grammar – Refr: [https://docs.Python.org/3/reference/grammar.html](https://docs.python.org/3/reference/grammar.html)

**Launching idle**

**idle** (Interactive Development Language Environment) is the name of the default command interface window that can access Python and simpy. By default, commands are displayed in this idle window, and output is sent to this window.

The default idle command prompt is: “>>>”

Starting an idle session depends on the how/where of a Python install. On IRT servers, launch an idle window with OS command idle. In-class demos use both a) an idle command window and b) a source file editing window.

**Python language background**

**Language version and Language keywords**

The import statement loads and executes an import statement’s argument one time.

How to get the current language version’s keywords (case sensitive):

>>> import os

>>> os.popen

<function popen at 0x000002243FE94510>

>>> os.popen("Python -V").read() # Get the Python version

'Python 3.11.8\n' < -- Instructor laptop Python version

>>> import keyword

>>> print(keyword.kwlist) # Display the current Python version’s keywords (note the remarkably FEW language keywords):

['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',

'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

*(This list has not changed through many versions of Python 3)*

Some examples display idle (command window) IDE prompt string “>>>” to show the context of command examples.

*Displayed lines beginning with “…” indicate required indentation of such lines, according to Python language syntax.*

*Actual indentation depth depends on the block level of such lines (more on source code indentation soon).*

**Editing and Running Python/simpy source code**

The idle command that launches an idle command window also invokes a command shell. “>>>” is the shell command prompt. The shell’s File menu item can be used to create a new source file, or/and, to open (for editing) an existing source file. In a source file editing session, selecting the Run menu item

Run - - > Run Module (or F5) translates (if needed) the source code in the current edit window, and if no translation errors, executes this source. By default, results are displayed in the idle window (we do not cover run results re-direction).

**Source code module structure**

It is optional, depending on Python/simpy source code purposes/deployment, to have or not have, a “main” method

(main is required in various programming languages). In this document, a main routine will usually NOT be used.

We will code Python/simpy command scripts; a script execution is done by running the source code script file a) via Run .

**Variables and values**

As in many languages, a variable is assigned a value using the “=” operator. But a Python variable’s type is not declared before assignment (Python implements dynamic (i.e., duck) typing). The type of a variable is inferred from its value, as in:

>>> variableName = value *< == Note: press Enter to terminate a shell statement*

and the current value of (the variable) variableName is displayed in the idle shell by

>>> variableName < -- Note that script source code line variableName does not display the variable’s value

**Constant and variable naming conventions**

Variable name must be a character sequence (with no embedded spaces)

Variable names consist of only letters, base-10 digits, and underscore (\_)

Variable names cannot begin with a base-10 digit.

Variable names  are case-sensitive, meaning that my\_int, MY\_INT, My\_Int, and mY\_iNt are all distinct variables.

There are no language-enforced naming standards: my\_int is considered conventional naming, whereas myInt is non-conventional camelCase naming. Consistency in naming across a project/system is best practice (less confusion for all).

MY\_CONSTANT = 1380 < - - All caps denotes a constant, but is not enforced by the language translator.

This document covers enough basic Python for developing moderate complexity simpy simulations.

**Short comments**

Comments within one source line are denoted with the pound/hash mark (#). When # is the first character of a line of code, that entire line is a comment. The # can also appear anywhere in a source line, and the rest of that line is a comment.

>>> # This entire line is a comment

VS

>>> foo = 1 # Inner line comment: assign int 1 to a variable named “foo” ; foo is a reference to a location containing int value 1

**Long comments**

Text of any length that is enclosed with triple double (or single) quotes is a comment. Special character sequences, such as

\n (new line) can be part of the comment. Also, triple quotes syntax has a role in retrieving source code documentation (illustrated later).

>>> x = """ long < - - “long” only in the sense of spanning source lines

comment """

>>> x

' long\ncomment ' < -- x value display retains the control newline characters

>>> print(x) < -- printing suppresses display of control characters but implements those controls

long

comment

*Note – In old (version2 Python, print was not a function, rather, an operator, and its use was as in: print x*

There is a choice in delimiter quotes: single triple quotes or double triple quotes:

s1 = '''This string contains """ so use triple-single-quotes.'''

s2 = """This string contains ''' so use triple-double-quotes."""

Example

>>> s1 # Display the value of s1

'This string contains """ so use triple-single-quotes.'

If a string contains both triple-single-quotes *and* triple-double-quotes, then you should escape one of them (a rare situation).

Example

>>> st = ' """ single triple quotes \'\'\' and double triple quotes """ some space """ '

>>> st

' """ single triple quotes \'\'\' and double triple quotes """ some space """ ' < == Printing does not display the escaped characters

>>> print("Python and %s are number %d" % ('Django', foo)) # Printing strings using old C-style formatted string

Python and Django are number 1 *< = Recall that foo, from above, is a reference to the integer 1*

**Built-in Types**

Some fundamental built-in types are **numerics**, **sequences**, **mappings**, **classes**, **instances** and **exceptions**.

A sequence is an implementation of an ordered set. *(Note - a set is unordered, thus not a sequence)*. Sequence examples include strings and lists. Sequence elements are indexable using [ ] notation, and the minimum index value defaults to 0.

A dictionary, i.e., hash container (with type ‘dict’) is not a sequence because items are accessed using keys, not indexes.

**More on Variables, Typing, and Assignment of value**

Variables can be thought of as names that refer to otherwise anonymous objects containing the actual values;

a variable can have both its type and value altered:

Each Python object also has an internal “id”, illustrated by: objectName - - - - > object

>>> w=3 object reference object’s value

>>> id(w)

1397994624

>>> foo = 'bar' # Assignment of a string value (could also code: foo = “bar” using double quotes

>>> foo

'bar'

>>> type(foo) # The built-in function named type returns the type of an object

<class 'str'>

>>> y = foo # y’s value is now the same as foo’s value, that is, y and foo are names for the same object

>>> foo = 1 # foo’s type (previously str) and value (‘bar’) are dynamically changed; the original variable name foo is destroyed

>>> foo

1 (foo is currently in class ‘int’, regardless of the string type it had before the value 1 was assigned, but y still has value ‘bar’)

>>>y

‘bar’

Variable foo (above) was mapped to string object, 'bar', but is then re-mapped to an integer object, 1. Note - the string foo previously referred to disappears, unless other variables are also referring to it (as shown by y in above code).

Thus, the remap of a variable’s type is not a conversion/cast because previous variable foo referenced the string that has been destroyed, and the new variable foo references the integer 1

>>> foo=1

>>> print(Foo) *< -- Review of naming rules*

Traceback (most recent call last): *< --- Error messages are thrown in the idle command window in Red*

File "<pyshell#12>", line 1, in <module>

print(Foo)

NameError: name 'Foo' is not defined *< == Thus, Foo is not the same identifier as foo because identifiers are case sensitive*

>>> id(foo) *< == All objects have a unique id within their lifetime*

1994372160

>>> Foo=1

>>> id(Foo) *< == Foo and foo now reference the same object (in the CPython implementation covered in this document)*

1994372160

>>> Foo=2

>>> id(Foo) < == foo and Foo now reference (i.e., name) different objects

1994372192

**Notes about None**

None (capitalized) is a special object in Python that represents absence of a value or, equivalently, a “null” value. It has a purpose similar to Java’s null. (For those with SQL language knowledge, None is **not** the same concept as SQL “null” which is typeless and is incomparable to non-null values).

None is an object having its own datatype, and None is comparable to any object.

>>> print(None, " has type ", type(None))

None has type <class 'NoneType'>

>>> print(none)

Traceback (most recent call last):

File "<pyshell#1>", line 1, in <module>

print(none)

NameError: name 'none' is not defined

And, unlike the null of SQL (that is incomparable to everything, including itself), variable values that are None compare equal to one another: >>> None == 3 has result: False whereas :>>> None == None has result: True

Also, None does not imply False, 0, or an empty list or string.

**Operators**

Python supports many of the same operators seen in other languages.

These include arithmetic operators, such as +, -, and \*, and so on, and this includes their corresponding augmented assignment operators, +=, -=, \*=, and so forth. So x = x + 1 can be coded as x += 1. Absent are the increment/decrement operators (++ and --) of other languages. The standard comparison operators, such as <, >=, ==, !=, and so on, are also available, and clauses can be grouped with Boolean AND and OR with “and” and “or”, respectively:

>>> (3 \* 4 > 10) and (5 + 5 >= 10)

True

A note on equality operators

The **==** operator tests whether two objects have the same value.

By contrast, the **is** operator tests whether operands x, y reference the same object. “is” is actually an abbreviation for id(x) == id(y)

*Always consult Python documentation for details that invariably arise during coding.*

*For example, unlike C, all comparison operations in Python have the same priority, which is lower than that of any arithmetic, shifting or bitwise operation. Also, unlike C, expressions like a < b < c mean the same as in mathematics.*

*Thus, this last expression is evaluated in Python as: a < b and b < c.*

*However, the assignment operator does not behave this way:*

>>> ww=xx

Traceback (most recent call last):

File "<pyshell#11>", line 1, in <module>

ww=xx

NameError: name 'xx' is not defined

>>> ww=xx=[] *< - - xx references an empty list ( a list container uses [ ] as its delimiter )*

>>> print("ww: ",ww, " and its id: ", id(ww), " and xx: ",xx, " and its id: ", id(xx))

ww: [] and its id: 2566491464904 and xx: [] and its id: 2566491464904

The results are because successive assignment results are associated (that is, applied) from right to left.

**Confirming/checking the type of an object**

>>> isinstance(Foo, int) *< == Assuming that Foo has value 2; the second argument can be any type, as demoed next*

True

isinstance works in general – *Example of applying isinstance to a do-nothing user-defined class definition instance*

>>> class Exclass:

""" Class documentation string (aka docstring) for a do-nothing class """

pass < == The statement that terminates the syntax for dummy class and function definitions

>>> ec = Exclass() *# An instance of class Exclass* *< == A null argument is passed to the default class instance creator*

>>> isinstance(ec,Exclass)

True

**Characters**

Interestingly, there is no data type “character”, nor “char”. Instead, individual characters are strings of length 1, and those characters are accessed and referenced as elements in strings using index access with [ ]. Such a string can be a name

(i.e., reference) for a string or a literal:

>>> x = 'abc' # x references a literal object of type string

>>> x[1] # Indexing a variable that references a string

'b'

>>> 'abc'[1] # Indexing a literal string is also valid

'b'

**An object’s Boolean Value**

Like various other languages, there are exactly two Boolean values: True and False (Note: they are capitalized).

All Python objects have a Boolean value (obtained by calling function bool) for any type of object.

Boolean is a subtype of type int. Any numeric type equal to zero has Boolean value False, and nonzero numeric values have Boolean value True.

The Boolean values a) of an empty container and b) of a nonempty container (such as a list with at least one element)

are False and True, respectively.

True and False can be explicitly assigned as a variable’s value. Note that Booleans, like numbers are literals.

>>> download\_complete = False

>>> bool(download\_complete)

False

More examples: >>> bool(-1.23) >>>bool(0.0) >>>bool(“”) >>>bool(None)

True False False False

Here is a more advanced example: a list with 2 elements

>>> bool([None, 0])

True

Explanation: A nonempty list has a True value. The “truthfulness” of objects arises when they are used in conditionals in

if and while statements where the execution path depends on object Boolean values (of either True or False).

Results of successive binary operators can depend on association (grouping) order, from left to right, as in:

>>> 1<(1<1) # These two examples show that an expression having a Boolean value also has an associated int value of 0 or 1

False

>>> (1<1)<1

True

The “not” operator negates (i.e., reverses) the Boolean value of a comparison:

>>> myval = 0

>>> nyvalue = not myval

>>> nyvalue

True

Also, the Boolean literals are case-sensitive:

>>> show\_output = true

Traceback (most recent call last):

File "<pyshell#1>", line 1, in <module>

show\_output = true

NameError: name 'true' is not defined

>>> show\_output = True # This assignment is legal; case matters in the truth value (“true” is, by default, undefined

**Source code blocking**

A block is a sequence of statements in a (source) program or script.

Python code blocks are delimited by indentation rather than curly braces or bracket pairings such as begin/end.

The statements in a given block are all indented exactly the same amount. The default is one tab, or 4 spaces, per indent level. However, indent character(s) used must be consistent in each source file – **NEVER MIX tabs and spaces in blocks**.

**Indentation is not a cosmetics recommendation, it is a syntax rule enforced by the language.**

Indent makes it easy to identify where blocks of code belong, and to take it one step further, it is impossible to have a

“dangling-else” problem, simply because an else belongs to one if or the other, namely the if at the same indent level.

When the last (non-comment) character of a source line is one “:” one or more following lines must indent another level.

All 3 code sections below, as displayed in idle, compute the same thing. Style wise, the first section is preferred. Following each of the sections with two successive returns will execute that section.

>>> if 1 + 1 == 2:  
...      print("foo") *< -- The “…” mean each of the 3 statements must be indented one level, that is, one tab (the default) to the right of the if*  
...      print ("bar")  
...      x = 42

**OR**  
>>> if 1 + 1 == 2:  
...      print ("foo"); print ("bar"); x = 42 (a “;” separates statements)

**OR**

>>> if 1 + 1 == 2: print ("foo"); print ("bar"); x = 42

(Each of the 3 sections above give the following results):

foo

bar

>>> x (Then, after the 3 statements are executed, display the value of the int scalar 42)

42

More condition statement demos

In this example the code is shown as it would be coded in a source file, not in an idle session (*notice the 1 tab indentation of the print() function call*)

Notice the consistent indentation of one or more statements that are treated as a group of statements. Each such statements group is executed as specified by source code logic flow. For example, the two print statements following “if 3==4” are executed when that test result is true.

-------------------------------------

Indentation denotes that the else is associated with the if 3==4 clause, thus “Statement3” is printed

if 'b' == 'b':

print('Statement0')

if 3==4:

print('Statement1')

print('Statement2')

else: < -- “:” after else is followed by >=1 indented statement

print('Statement3')

*has the output*

Statement0

Statement3

Python generally has an absence of symbols. Not only are there no delimiting braces, but also no trailing semicolon (;) for terminating statements, no dollar signs ($), and no required parentheses (( )) for conditional statements, as illustrated in the preceding if examples.

Occasional “at” (@) signs for decorators (a function construct not covered here) and underscores (\_) can occur.

**Python Standard Types**

Standard types include scalars or literals (such as numbers (integers & floats) and strings), as well as containers,

i.e., structures for naming and grouping together multiple Python objects into a container object referenced by one name.

**Object type review**

Return the Python type of someObject using the function call type(someObject)

Test an object’s type using function call isinstance(object,type); for example: >>> isinstance(8,int) has result: True

Python has two primary numeric types: **int** (for integer) and float (for floating point numbers).

In following its principle of keeping it simple,

Python has one integer type, int, as opposed to other languages with multiple integer types. (Python previously had another integer type called long; its functionality has been merged into the version 3.x int implementations).

One int type is accomplished by avoiding integer hardware sizes/preferences. A portable indefinite/infinite bits representation is now used, so that, for example, -5 is treated by bitwise operators as "...1111111111111111111011".

Specifically, negative int values are represented as a variant of 2’s complement, displaying as unlimited left sign extension.

In addition to normal base-10 notation, integers can be represented in hexadecimal (base 16) and octal (base 8).

Floats are double-precision floating-point real numbers familiar from other languages. Examples of int and float:

>>> 1.25 + 2.5

3.75

>>> -9 - 4

-13

>>> 1.1

1.1000000000000001 *< -- To be amplified on later; the reason for the trailing “1” in the representation of 1.1 is that the base 10 fraction 1/10*

*cannot be represented exactly in base 2.. The display of such a value is an approximation formed from the result of rounding up/down*

Regarding the last example, floats have a large range; however, they are not very accurate in terms of representing rational numbers with a repeating fraction. In addition, Python type float has limited precision (means significant digits) compared with unlimited type int precision, as illustrated below:

myInt = 28700000000000000000000000000000000000000000000000000035400000000999264839020876555555

myInt += 1

myInt # Indefinite precision is also implemented for arithmetic on large type int values

28700000000000000000000000000000000000000000000000000035400000000999264839020876555556

myFloat = 1.28700000000000000000000000000000000000000000000000000035400000000999264839020876555556

myFloat += 1

myFloat # A float value has limited precision, because it uses the machine’s floating point hardware instructions

2.287

There is another floating-point type called Decimal—which is not a built-in type and must be accessed (i.e. imported) via the decimal module. Decimal has a smaller value range, but better precision. There is also built-in complex number support.

NumPy (a mnemonic for Numerical processing in Python) is open source support for large multi-dimensional arrays and matrices, and math functions such as distributions – *We will make use of some numpy-based pdf distributions*

A trailing 'L' representing long integers is seen in old code and documentation, such as:

1L, -42L, 99999999999999999L, and so on.

**Table - Built-in Numeric Types**

**Type Description Examples**

int Signed Integers (no size limit) -1, 0, 0xE8C6, 0377, 42 (the second example is hexadecimal (base 16))

float Double-precision Floating-Point Numbers 1.25, 4.3e+2, -5., -9.3e, 0.375

complex Complex (Real+Imaginary) Numbers 2+2j, .3-j, -10.3e+5-60j

Machine dependent info for numeric types is obtained from calls to sys; this display is for a Dell Latitude 5570, core i5:

>>> print(sys.int\_info)

sys.int\_info(bits\_per\_digit=30, sizeof\_digit=4)

>>> print(sys.float\_info)

sys.float\_info(max=1.7976931348623157e+308, max\_exp=1024, max\_10\_exp=308, min=2.2250738585072014e-308,

min\_exp=-1021, min\_10\_exp=-307, dig=15, mant\_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)

**Numeric Operators**

Numbers support basic arithmetic operations: addition (+), subtraction (-), multiplication (\*), division (/ and //), modulus (%), and exponentiation (\*\*). The division operator / represents “classic division”, meaning truncation when both operands are integers (floor division) but “true division” for floats. Python also features an explicit “floor division” operator that always returns an integer result (meaning a fraction part that is discarded regardless of its operand types:

>>> 1 / 2 # true division (int operands, but float is result type)

0.5

>>> 1.0 / 2.0 # true division (float operands)

0.5

>>> 1 // 2 # floor division (// operator)

0

>>> 1.0 // 2.0 # floor division (// operator)

0.0

>>> type(1.0)

<class 'float'>

>>> type(1.0//2.0) # Thus, // with type float operands returns type float, and the numeric value has no fraction part.

<class 'float'>

Python integers have bitwise operators for binary AND (&), OR (|), XOR (^), bit inversion (~), and left and right shift (<< and >>), as well as their augmented assignment equivalents, such as &=, <<=, and so forth. Python supports bit processing for integers interpreted as bit sequences.

>>> 7 | 5 # Inclusive or

7

>>> 7 ^ 5 # Exclusive or

2

>>> w=7

>>> ~w # The “complement” (bit flips) of w is the bit representation of -w-1 = (-7-1 = -8). Similarly, w= -5 gives ~w= -(-5)-1 = 4 = … 0000000000100

**Numeric Built-in and Factory Functions**

Each numeric type has a factory function that enables users to convert from one numeric type to another. Some languages use the terminology “conversion” and/or “casting” but this is not how Python does type conversion. Python does not change the type of an existing object. Instead, a new object is created (*hence the term “factory”*):

int(12.34) creates a new int object with value 12 (based on fraction truncation), whereas float(12) returns 12.0.

Finally, we have complex and bool. Python also features a handful of operational built-in functions that apply to numbers

such as rounding floats to a specified number of digits or abs for the absolute value of a number. Built-ins examples:

>>> int('123')

123

>>> type(int('123')) # Confirm the type of the resulting new integer

<class 'int'>

>>> int(45.67)

45

>>> round(1.15, 1)

1.2

>>> float(10)

10.0

Float values

Many numeric float values cannot be represented exactly on binary-arithmetic based computers. This means that much numeric fractional scaling and approximation is involved with applications involving float values/expressions. Here is a simple demo of type float approximation issues:

>>> .1 + .1 == .2

True

>>> .1 + .1 + .1 == .3 *< -- A surprise to those unfamiliar with hardware number representations*

False

Some factory functions applied to strings

>>> ord('a') # The ord() function returns an integer representing the Unicode code point for the given Unicode character.

97

>>> chr(65) # Returns the string representing a character whose Unicode code point is the integer argument (chr is inverse of ord)

'A'

>>> ord('A')

65

>>> chr(ord('A'))

'A'

>>> chr(8364) < - - A more obscure Unicode example

'€’

**About containers**

Def - A container is any object that supports testing for membership using the “in” operation.

For example, >>> '4' in '1342' result is: True. Other containers include: lists, dictionaries, and tuples (covered soon).

>>> x = 'abc'

>>> import collections

>>> isinstance(x, collections.Container) # Is x a container?

True

>>> isinstance(5, collections.Container) # Is 5 a container?

False

“in” is not a recursive operator, illustrated by following shell commands:

>>>lis = [1,2,[2,3]] - -> >>>3 in lis - -> False - -> >>>2 in lis --> True

**Immutable objects**

Def - immutable object cannot have its value changed/modified Examples: a literal string, number, Boolean.

A divmod result, is another kind of immutable called a **tuple** that can have several components.

divmod

<built-in function divmod>

>>> (divmod(15,6)) has result (2, 3) first element is whole number division result, the second element is the division’s remainder

>>> type( (2, 3) ) # See that divmod result is a tuple

<class 'tuple'>

>>> len((2,3)) # (2,3) has a length; compare with: len(5), that throws an error (because a scalar has no length property)

2

>>> (2,3)(1) # Indexing accessed tuple components, not function call notation as with “()”.

Traceback (most recent call last):

File "<pyshell#22>", line 1, in <module>

(2,3)(1)

TypeError: 'tuple' object is not callable *< -- This error generally means x(someArg) is a function (but, here, x is not a function)*

>>> (2,3)[1] # For any sequence type, including tuples, the indexing operation is “[ ]”

3

Also, an immutable that is a member of a list can also be accessed by list indexing.

Given >>> x=(2,3) and >>> xlist = [x]

>>> xlist[0] result is (2, 3)

>>> xlist[0][1] *< -- Get the second (indexed by 1, not 2) element of the tuple x that is contained in the list named xlist*

3

In summary, a tuple is an ordered and immutable collection. Unlike a set, a tuple allows duplicate members**.**

**Sequence data types, and Iterables**

Many programming languages have arrays for grouping like objects, accessible by index position. Python’s sequence types support various array-like features; and the importable NumPy package does implement arrays

A sequence represents a finite ordered collection indexed by int values Out-of-bounds int values cause ‘out of range’ errors. Moreover, for any sequence y, y[.5] will produce the error:

Traceback (most recent call last):

File "<pyshell#20>", line 1, in <module>

y[.5]

TypeError: list indices must be integers or slices, not float

Two commonly used Python containers are **lists** (Ex: 3 element list: [ 1,2,[‘xxx’] ]) and **strings** (Ex: string: 'Python').

They are part of a broader set of data structures called sequences (examples & use below).

A sequence is an example of a type that is an **iterable**: a data structure you can “traverse” or “iterate”, one element at a time. The basic idea behind an iterable is you can continually ask it for the next object, and it continues to “read off” its internal collection of objects, per object. To test if object x is iterable, iter(x) throws an error if x is not iterable

Python sequences are not only iterators in this way, but also support random access - capability to retrieve an object at a specific position. *Python uses iterables more extensively than most languages, internally & at great advantage in coding.*

In total, Python has six sequence/sequential data types: strings; byte sequences; byte arrays; lists; tuples; range objects.

**Sets**

Objects of type set are an example of a type that is not a sequence. A Python set behaves much like a set in mathematics.

>>> p = {1,2} < -- Notice the curly braces notation

>>> 2 in p # Demo set membership and subset testing

True

In mathematics, a set is not ordered. Thus, indexing the elements is meaningless, as shown here for Python sets.

>>> p[1]

Traceback (most recent call last):

File "<pyshell#95>", line 1, in <module>

p[1]

TypeError: 'set' object does not support indexing

>>> isinstance(p,set)

True

>>> x = {1,2}

>>> y = {4,1,2}

>>> x.issubset(y)

True

A [set](https://www.w3schools.com/python/python_sets.asp)’s elements are unordered, immutable, and cannot be indexed. A set cannot have duplicate members.

**Sequence Slicing**

The following example operate on a string. Unlike many other languages, a string can be treated both as a discrete object or/and as a list of individual characters.

>>> s = 'Python'

>>> s[0] # The first element of s

'P'

>>> s[4]

'o'

>>> s[-1] # The last element of s

'n'

Python supports negative indices. It is common in many languages to code something like

data[len(data)-1] or data[data.length-1] to get the last element of an array.

As the s[-1] example above shows, user does not need to code a function call, because the simple index -1 suffices.

Indexing multiple elements of a sequence is called slicing in Python.

Slicing is specified by a pair of indices, say i and j, delimited by a single colon (:). When a slice is requested, the interpreter takes the subset of elements beginning at the first index i, up to but *not including* the second index j.

>>> s = 'Python'

>>> s[1:4]

'yth'

>>> s[2:4]

'th'

>>> s[:4] # Unspecified first index means first position

'Pyth'

>>> s[3:]

'hon'

>>> s[3:-1] # From 4th position to (position to the left of the last position)

'ho' # ‘n’ is not included in result because the 2nd slice parameter position (here, the last char) is not included in the result

>>> s[:] # Operator closure: the result of a slice of a sequence is another sequence of the same type

'Python'

>>> str(s)

'Python'

The absence of an index means either from the beginning or through to the end, depending on which slice parameter is missing. An improper slice (meaning to return a copy of the entire sequence) can be designated with [:].

The slicing syntax is also applicable to lists and all other sequence types.

**Other Sequence Operators**

We saw the slicing operation in the previous section using the [ ] and [:] operators. Other operations on sequences include concatenation (+), repetition/duplication (\*), and membership (in) or non-membership (not in).

Strings were used in examples above, but these operations apply to other sequences as well.

>>> 'Python and' + 'Django are cool!'

'Python andDjango are cool!'

>>> 'Python and' + ' ' + 'Django are cool!'

'Python and Django are cool!'

>>> '+’\*10 # Duplication example

'++++++++++'

>>> 'an' in 'Django'

True

>>> 'xyz' not in 'Django'

True

**Alternatives to + operator for Concatenating sequences**

Using the + operator with sequences should generally be avoided, especially in advanced coding work.

When you’re new to the language, + does solve the problem of putting a pair of strings together; however, it’s not a solution that provides the best performance. (*The reasons involve Python’s C underpinnings* *& the data copying by operations*)

When we say “copy,” we mean a copy of the references and not of the objects themselves that is called a shallow copy.

Next is an outline of various copy approaches.

Instead of 'foo'+'bar', you can use string format operator (%) discussed in the following strings section, as in

'%s%s' % ('foo', 'bar'). Another way of putting strings together, especially given a list of strings to merge together, is the string join method, such as ‘’.join(['foo', 'bar']). The leading ‘’ can contain any empty (as here) or non-empty string to separate the concatenated result of ‘foo’ and ‘bar’. Join is the preferred concatenation, especially for large strings:

>>> '&&'.join(['foo', 'bar','bolt'])

'foo&&bar&&bolt'

extend adds the contents of another list to the current list (vs. list1 += list2); list1.extend(list2) is similarly preferred.

>>> l1, l2 = [123], [45,89] # **Unpacking** (i.e., assigning) right-side values (2 lists) into left side variables l1 and l2

Works for any right-side of assignment that is iterable

>>> l1.extend(l2)

>>> l1

[123, 45, 89]

**memoryview sequence type**

Just brief mention, applications a) doing large-scale amounts of string processing or b) frequently references large storage areas can reduce memory copying in such applications by using memoryview (was known as “buffer” in Python2).

**Lists in more detail**

The Python type most like arrays of other languages is lists. A list is a mutable, resizable sequence of type instances, each of whose locations can hold an object of any type.

Explanations of each of the following statements are given below the code.

>>> book = ['Python', 'Development', 8] # 1) create list, and notice the mixed element object types

>>> book.append(2008) # 2) append an element

>>> book.insert(1, 'Web') # 3) insert an element at a specified list index position

>>> book

['Python', 'Web', 'Development', 8, 2008]

>>> book[:3] # 4) slice is first 3 elements, at indexes 0, 1, and 2

['Python', 'Web', 'Development']

>>> 'Django' in book # 5) is string ‘Django’ an element in the book list?

False

>>> book.remove(8) # 6) remove element identified by its literal value

>>> book.pop(-1) # 7) remove last element; pop(-2) removes 2nd to last element, if it exists (else index range error thrown)

2008

>>> book ['Python', 'Web', 'Development']

>>> book \* 2 # 8) repetition/duplication, but book is not modified

['Python', 'Web', 'Development', 'Python', 'Web', 'Development']

>>> book.extend(['with', 'Django']) # 9) merge extend’s argument list into list named book

>>> book

['Python', 'Web', 'Development', 'with', 'Django']

Explanations for the previous example:

1. Create list initially with a pair of strings and an integer.

2. Add another int to the end of the list.

3. Insert a string into the second position (at index 1).

4. Pull out a slice of the first three elements.

5. Membership check (Is an item in the list?)

6. Remove an item regardless of its location in the list.

7. Remove (and return) an item by its location (index).

8. Demonstrate the repetition/duplication operator \*.

9. Extend this list with another one.

**A first script/program example**

"""

demoLists.py WJM  *Each script/function/class body starts with a docstring consisting of text description surrounded by “”” “””*

Demo simple list processing AND the effects of a list append function

"""

lis1 = [1,2]

print("lis1's internal id ",id(lis1))

print("First element in list lis1 ",lis1[0]) # List indexing starts at 0, not 1

"""

The following assignment does NOT do what Python beginners think it does:

This is an example of the Python coding convention of a “docstring” at the top of each module, class, and function.

Lengthy, multi-line comments anywhere in code should also be delimited by a beginning and ending “””

Execute this script using the Run menu in a source file window –--------------------------------------

The preamble (1-line) program description following the filename and author/date documentation should be expanded when the source program is a simpy source file because DES modules must also document a) Usage (how to run the model) b) run termination criteria, and

c) model time unit (t.u.).

(*All the above is the same kind of documentation that was expected in gpssW source code*)

The script’s code is executed from top to bottom.

One of the things this means is that a function call such as h(x,y) cannot precede the def … section that defines h.

**A main() function is usually unnecessary in homework scripts**

1) The append function adds the list argument to lis1 2) the append function returns

the value None; in other words, append has a side-effect, and 3) variable lis2 is a

reference to the result returned by append()

"""

lis2 = lis1.append([3,4]) # A 2-element list becomes lis1's element with largest index

print("List lis1's modified contents: ",lis1)

"""

*Assignment to lis2 was NOT a copy of the enlarged list lis1.*

*Surprisingly, lis2 was assigned None, because append() returns the value None*

"""

print("lis2 type and value are ",type(lis2)," and ",lis2)

**Script execution Results:**

lis1's internal id 2144140278600

First element in list lis1 1

List lis1's modified contents: [1, 2, [3, 4]]

lis2 type and value are <class 'NoneType'> and None

**List Methods**

Reset the list named book back to its value in the middle of the previous set of examples.

We then sort the list, with some follow-up discussion afterward.

>>> book = ['Python', 'Web', 'Development', 8, 2008]

>>> book.sort() # NOTE: in-place sort ; that is, the list is sorted, but results of the sort are not displayed

>>> book

[8, 2008, 'Development', 'Python', 'Web']

A “sort” on mixed types is, technically, undefined. How can you compare objects (for example strings versus numbers) that have no relationship? The algorithm Python uses is a “best guess” as the “right thing to do”: Sort all the numeric values first (smallest to largest) followed by a lexicographic sort of the strings. Sorting files and class instances makes results less defined.

List built-in methods such as sort, append, and insert modify the object directly and (as some above examples show) do not have a return value. Newcomers to Python might find it strange that sort does not return a sorted copy of the list, so beware.

In contrast, string method upper returns a string copy of the original string, but in UPPERCASE – see Strings section.

Unlike lists, strings are not mutable. That is why upper returned a (modified) copy. But the original string still exists.

Python3 has built-in functions sorted and reversed, which take a list as an argument and return a sorted or order-reversed copy.

In summary, a list is collection that is ordered and mutable; a list allows duplicate members.

**Dictionaries**

A dictionary is a container having <key, value> pairs, where each value is accessed by a corresponding key.

A dictionary object is a Python container having **type 'dict'**. dict type implements the familiar hash structure and operations.

thisdict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
print(thisdict)

print(thisdict["brand"])

==================

{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

Ford

Caution: Each dictionary value is accessed by the syntax dictName[key]. This is not conventional “indexing” via int indexes.

>>> d = { 1: 'abc', 2: [1,2,3] }

>>> d

{1: 'abc', 2: [1, 2, 3]}

>>> d[1]

'abc'

>>> d[0] < -- If d was indexable, then d[x] would return the value at dict position x; instead, d[x] returns the value for the key x

Traceback (most recent call last):

File "<pyshell#118>", line 1, in <module>

d[0]

KeyError: 0

A dictionary cannot have two items with the same key:

If 1: ‘xyz’ is added to dict d above, then value ‘abc’ is removed from d and replaced by new value ‘xyz’:

>>> d[1] = 'xyz'

>>> d

{1: 'xyz', 2: [1, 2, 3]}

To add an item to a Python dictionary, you should **assign a value to a new index key in your dictionary**. Unlike lists and tuples, there is no add() , insert() , or append() method available for adding items to a dict type object..

The keys in a dictionary (aka “dict”) can be from different types:

>>> stuff = {1: 'apple','b': 'grape'} - -> >>> stuff['b']

'grape'

To determine how many items a dictionary has, use the len() function.

As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

= > in Python 3.6:

>>> d[0] = 45

>>> d

{1: 'xyz', 2: [1, 2, 3], 0: 45} # Notice that entry 0: 45 was not placed at the beginning of d.

When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.

Unordered means that the items do not have a defined order, so indexed access is not possible = > dict is not a subtype of sequence.

Whenever access to each item in a container is only by a key (i.e. unique retrieval key) that container object should be type dict.

>>> isinstance(d, collections.Container) # Assuming the previous import collections statement is still in effect

True

A **list comprehension** specifies a result as a list [ ] such that the list items are obtained by iterating over the elements in some object, and calculating something.

(Notice the similarity of list comprehension syntax to that of a relational database SQL query).

**Examples Combining dict and list processing in a script**

Ex1: add the values of the type int items in a dictionary

inventory = {'rope': 1, 'torch': 6, 'money': '$1000',’strategy’: [‘planA’,’planB’]}

# Sum the dict values that are type int

print( "Sum of all int values in inventory is ", sum([ x for x in inventory.values() if isinstance(x,int) ]) )

# Even simpler code than the above sum depends on the sum function recognizing that list comprehension creates a list

print( "Sum of all int values in inventory is ", sum( x for x in inventory.values() if isinstance(x,int) ) )

Here are the results of the above two print() statements

=================================== RESTART: C:/Users/mitchell/148\_f23/dictProcessingExamples.py ==================================

Sum of all int values in inventory is 7

Sum of all int values in inventory is 7

The list comprehension examples above simply constructed a new list by collecting items from a source object (here, from a dict).

However, list comprehension can also construct a new list from the results of function calls/expressions/etc, as illustrated next:

Using a slightly larger dict from above examples, the list comprehension below computes a list of expression values (in this case, the expression is the built-in len function:

inventory = {'rope': 1, 'torch': 6, 'money': '$1000','strategy': ['planA','planB'], 5:500}

[len(z) for z in inventory.values() if type(z) != int]

[5, 2]

Explanation of results

The non-int type key values are ‘money’ and ‘strategy’. The corresponding length of the values of each of these keys is

len(‘$1000’) which is 5 and len(‘strategy’) which is 2 (since it is a list containing two string values).

In summary, no duplicate members allowed.

That is, duplicate keys are dis-allowed, but two distinct keys can have the same value.

A dictionary is a collection which is ordered *(Python 3.7 and newer*) and mutable

= => Application requirements determine where and why lists VS dictionaries VS Numpy arrays, etc. are used.

**Range sequence objects and their use**

Coding often needs a sequence of integers starting from a specified start integer to (but NOT including) a stop integer.

The built-in range function returns a range object consisting of integers. A common use is loop iteration.

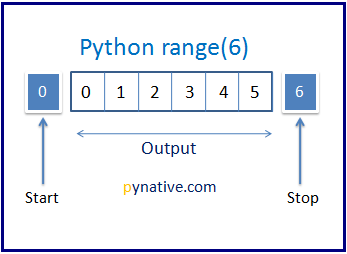
# Generate numbers between 0 to (but not including) 6; 6 is referred to as the “stop” value. 0 is the default start value.

>>> for i in range(6):

print(i, end=" ") *< == The end=”someString” print argument separates result values on the same output line*

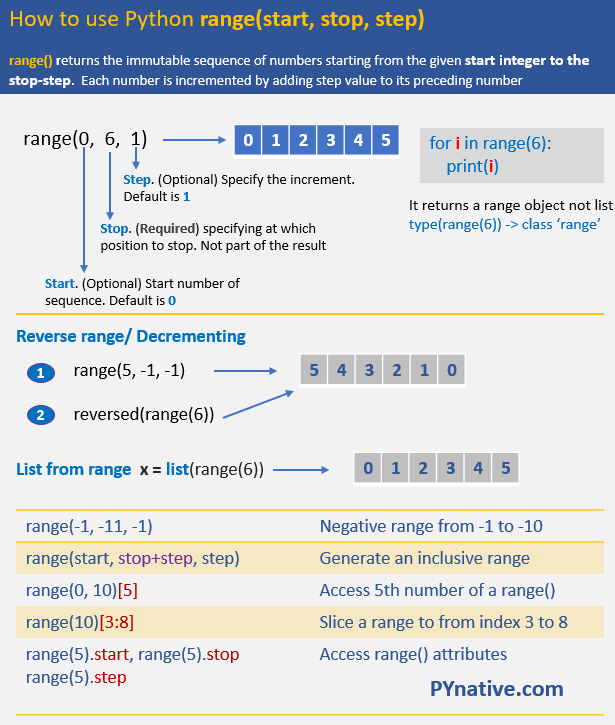
=======================

0 1 2 3 4 5



The figure below details some of the optional parameters for a range object.

**Source**: https://pynative.com/python-range-function/#h-range-over-character-or-alphabet



The range() function only works with integers ==> all arguments must be integers. You cannot use float

numbers or any other data type as a start, stop, and step value.

Any of the three range arguments can be positive or negative. The step value must not be zero.

**range() example using all 3 arguments**

A for loop executes a block of code repeatedly for a specified number of times. We can iterate over a sequence of numbers produced by the range() function using a for loop.

for i in range(1, 10, 2):

… print("Current value of i is:", i)

=============================

Results:

Current value of i is: 1

Current value of i is: 3 < -- Notice that the first range value is start, and successive range values are (previous range value + step)

Current value of i is: 5

Current value of i is: 7

Current value of i is: 9

**Concatenating the result of two range() object (how to build a custom range object)**

Consider “adding” range(5) and range(10,15) to get the concatenated range like [0, 1, 2, 3, 4, 10, 11, 12, 13, 14].

Add/merge the result of multiple range() functions using itertools.chain().

""" Demo the concat/union of two range objects"""

from itertools import chain # itertools provides functionality for processing iterable ojects

# Concatenate ranges

new\_range = chain(range(5), range(10,15))

for num in new\_range:

print(num, end=' ')

=========================

0 1 2 3 4 10 11 12 13 14

Another example – range elements in a chain does NOT need to be in any order

nc = chain(range(2),range(-3,0))

for x in nc:

print(x,end= ' ')

=========================

0 1 -3 -2 -1

**Iterate a container - several variations of the for loop**

You can iterate Python sequence types such as lists, strings, dict keys, etc. by choosing among for loop variations.

Examples of variations in for loop statements. Assume given the list values = ["a", "b", "c"].

Note - *for any variable x having a string value print(x), displays only the string’s characters but not the delimiter quotes.*

**C ) Almost as good as we can expect; eliminated loop index manipulations**

for value in values:

print(value)

a

b

c

Prints the list values only

**A ) FIRST APPROACH –**

**Index babysitting**

index = 0

for value in values:

print(index, value)

index += 1

0 a

1 b

2 c

Prints each list element value and its list position using variable index.

**B ) Better than A), but at cost of coding a len() function call and indexing access**

for index in range(len(values)):

value = values[index]

print(index, value)

0 a

1 b

2 c

The len function provides the range iterator’s STOP value, but the code must explicitly call the len() function

An alternative that avoids all tedious (and error-prone) code of A),B), and C) uses **enumerate**.

Because a list is an iterable, enumerate accesses the list’s <position, item value> pairs, one pair at a time.

**D ) A more sophisticated version of C ) that does more than C ) and uses only 2 lines of code**

for count, value in **enumerate**(values): *Variables count and value are not special names with enumerate, meaning that*

print(count, value) *count and value can be any two variables. In addition, the first print argument*

0 a can be replaced by count+1, for example, if desired, resulting in 1,2,3 replacing 0,1,2

1 b (Moreover count+1 could be any expression based on variable count) 2 c

Little known, even by many Python programmers, is the else clause of a for statement. The body of an else clause executes whenever a loop terminates normally, that is, with no break statement having aborted the loop. *An else clause avoids the need to create and manage a “flag” variable in the loop body.*

Example 1:

intsLessThanTen = 0

for x in [1,2,3,7]:

if x > 9:

break # break terminates loop execution, and resumes at the statement following the loop

intsLessThanTen += 1 # Found another small int

else: # Here because the loop exited normally (without doing break)

# The things to do when the loop terminated normally

print(intsLessThanTen," integers less than 10 were found")

================================ RESTART: C:/Users/mitchell/148\_f22/forStatementElseClause.py ================================

4 integers less than 10 were found

Example 2: Looping over the items in a container and updating container values in that same loop is error prone.

Instead, looping over a copy can be less error-prone.

y = [1, 3, -7]

y.copy() # A list has the copy function feature

[1, 3, -7]

for index,value in enumerate(y.copy()):

if value < 0:

y[index] = 1000

>>> y # See that y has been modified

[1, 3, 1000]

**While statement**

Most languages have a loop statement (such as while) with which control and termination are controlled by app-specific conditions.

Also, the break statement (in any form of loop) terminates loop execution (but not program/script execution):

“””Starting with 0, build a list of non-negative int values until the list has length 3”””

mylist, nextValue = [ ], 0 # **Unpack** the right-side sequence of two objects and assign their values to the left-side variables

while True:

mylist.append(nextValue)

if len(mylist)>2: # List now has length 3

break

else:

nextValue += 1

print('Final list ',mylist)

====================

Final list [0, 1, 2]

**First use of Python** **Exceptions**

Each statement execution could potentially involve exceptions. The except clause in a try block is a (pre-defined by Python) example:

x=1

try:

y=x/0

except:

print("An exception occurred")

Results of executing the above 5 lines of code in idle is

An exception occurred

*except is a generic mechanism whose body executes automatically when an exception occurs. Getting the detailed reason, that is, the cause for the exception requires additional coding.*

**Closing notes about this document**

If you are new to Python, what will strike you is the streamlined language elements:

statement blocking specified by indent, not by {} or () or begin/end and such. No need for “;” terminators

absence (meaning unavailable in the language) of little-used and error prone constructs such as ++, case