0. INTRODUCCIÓN A PYTHON

Docente
M.C. Hugo Armando Guillén Ramírez
hguillen@uabc.edu.mx / hugoagr@gmail.com

2018-1

LITERATURA RECOMENDADA

- Sweigart, A. (2015). *Automate the Boring Stuff with Python: Practical Programming for Total Beginners*. No Starch Press.
- Lutz, M. (2014). *Python Pocket Reference: Python In Your Pocket*. " O'Reilly Media, Inc.".
- McKinney, W. (2012). *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. " O'Reilly Media, Inc.".

Table 1-3: Valid and Invalid Variable Names

Valid variable names	Invalid variable names
`balance`	`current-balance` (hyphens are not allowed)
`currentBalance`	`current balance` (spaces are not allowed)
`current_balance`	`4account` (can't begin with a number)
`_spam`	`42` (can't begin with a number)
`SPAM`	`total_$um` (special characters like `$` are not allowed)
`account4`	`'hello'` (special characters like `'` are not allowed)

Table 12. Python 3.X reserved words

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

OPERADORES

Table 6. Numeric operations (all number types)

Operation	Description	Class method
`X + Y, X - Y`	Add, subtract	`__add__, __sub__`
`X * Y, X / Y, X // Y, X % Y`	Multiply, divide, floor divide, remainder	`__mul__, __truediv__`[a], `__floordiv__, __mod__`
`-X, +X`	Negative, identity	`__neg__, __pos__`
`X | Y, X & Y, X ^ Y`	Bitwise OR, AND, exclusive OR (integers)	`__or__, __and__, __xor__`
`X << N, X >> N`	Bitwise left-shift, right-shift (integers)	`__lshift__, __rshift__`
`~X`	Bitwise invert (integers)	`__invert__`
`X ** Y`	*X* to the power *Y*	`__pow__`
`abs(X)`	Absolute value	`__abs__`
`int(X)`	Convert to integer[b]	`__int__`
`float(X)`	Convert to float	`__float__`
`complex(X), complex(re,im)`	Make a complex value	`__complex__`
`divmod(X, Y)`	Tuple: `(X / Y, X % Y)`	`__divmod__`
`pow(X, Y[,Z])`	Raise to a power	`__pow__`

Table 2. Comparisons and Boolean operations

Operator	Description
`X < Y`	Strictly less than[a]
`X <= Y`	Less than or equal to
`X > Y`	Strictly greater than
`X >= Y`	Greater than or equal to
`X == Y`	Equal to (same value)
`X != Y`	Not equal to (same as `X<>Y` in Python 2.X only)[b]
`X is Y`	Same object
`X is not Y`	Negated object identity
`X < Y < Z`	Chained comparisons
`not X`	If `X` is false then `True`; else, `False`
`X or Y`	If `X` is false then `Y`; else, `X`
`X and Y`	If `X` is false then `X`; else, `Y`

Table 3. Sequence operations (strings, lists, tuples, bytes, bytearray)

Operation	Description	Class method
`X in S` `X not in S`	Membership tests	`__contains__`, `__iter__`, `__getitem__`[a]
`S1 + S2`	Concatenation	`__add__`
`S * N, N * S`	Repetition	`__mul__`
`S[i]`	Index by offset	`__getitem__`
`S[i:j], S[i:j:k]`	Slicing: items in `S` from offset `i` through `j`–1 by optional stride `k`	`__getitem__`[b]
`len(S)`	Length	`__len__`
`min(S), max(S)`	Minimum, maximum item	`__iter__`, `__getitem__`
`iter(S)`	Iteration protocol	`__iter__`
`for X in S:`, `[expr for X in S]`, `map(func, S)`, etc.	Iteration (all contexts)	`__iter__`, `__getitem__`

Table 4. Mutable sequence operations (lists, bytearray)

Operation	Description	Class method
`S[i] = X`	Index assignment: change item at existing offset `i` to reference `X`	`__setitem__`
`S[i:j] = I`, `S[i:j:k] = I`	Slice assignment: `S` from `i` through `j`–1 with optional stride `k` (possibly empty) is replaced by all items in iterable `I`	`__setitem__`
`del S[i]`	Index deletion	`__delitem__`
`del S[i:j]`, `del S[i:j:k]`	Slice deletion	`__delitem__`

Table 5. Mapping operations (dictionaries)

Operation	Description	Class method
`D[k]`	Index by key	`__getitem__`
`D[k] = X`	Key assignment: change or create entry for key *k* to reference *X*	`__setitem__`
`del D[k]`	Delete item by key	`__delitem__`
`len(D)`	Length (number of keys)	`__len__`
`k in D`	Key membership test[a]	Same as in Table 3
`k not in D`	Converse of `k in D`	Same as in Table 3
`iter(D)`	Iterator object for *D*'s keys	Same as in Table 3
`for k in D:`, etc.	Iterate through keys in *D* (all iteration contexts)	Same as in Table 3

STRINGS

String formatting

In both Python 3.X and 2.X (as of 3.0 and 2.6), normal `str` strings support two different flavors of string formatting—operations that format objects according to format description strings:

- The original expression (all Python versions), coded with the `%` operator: *`fmt`*`%` `(`*`values`*`)`
- The newer method (3.0, 2.6, and later), coded with call syntax: *`fmt`*`.format(`*`values`*`)`

Table 8. % string formatting type codes

Code	Meaning	Code	Meaning
`s`	String (or any object, uses `str()`)	`X`	`x` with uppercase
`r`	`s`, but uses `repr()`, not `str()`	`e`	Floating-point exponent
`c`	Character (int or str)	`E`	`e` with uppercase
`d`	Decimal (base 10 integer)	`f`	Floating-point decimal
`i`	Integer	`F`	`f` with uppercase
`u`	Same as d (obsolete)	`g`	Floating-point `e` or `f`
`o`	Octal (base 8 integer)	`G`	Floating-point `E` or `F`
`x`	Hex (base 16 integer)	`%`	Literal '%' (coded as %%)

Table 7. String constant escape codes

Escape	Meaning	Escape	Meaning
`\`*`newline`*	Ignored continuation	`\t`	Horizontal tab
`\\`	Backslash (\)	`\v`	Vertical tab
`\'`	Single quote (')	`\N{`*`id`*`}`	Unicode dbase id
`\"`	Double quote (")	`\u`*`hhhh`*	Unicode 16-bit hex
`\a`	Bell	`\U`*`hhhhhhhh`*	Unicode 32-bit hex[a]
`\b`	Backspace	`\x`*`hh`*	Hex (at most 2 digits)
`\f`	Formfeed	`\`*`ooo`*	Octal (up to 3 digits)
`\n`	Line feed	`\0`	Null (not end of string)
`\r`	Carriage return	`\`*`other`*	Not an escape

Splitting and joining methods

`S.split([sep [, maxsplit]])`

Returns a list of the words in the string `S`, using *`sep`* as the delimiter string. If *`maxsplit`* is given, at most *`maxsplit`* splits are done. If *`sep`* is not specified or is `None`, any whitespace string is a separator. `'a*b'.split('*')` yields `['a','b']`. Hint: use `list(S)` to convert a string to a list of characters (e.g., `['a','*','b']`).

`S.join(iterable)`

Concatenates an iterable (e.g., list or tuple) of strings into a single string, with `S` added between each item. `S` can be '' (an empty string) to convert an iterable of characters to a string (`'*'.join(['a','b'])` yields `'a*b'`).

`S.replace(old, new [, count])`

Returns a copy of string `S` with all occurrences of substring *`old`* replaced by *`new`*. If *`count`* is passed, the first *`count`* occurrences are replaced. This works like a combination of `x=S.split(old)` and *`new`*`.join(x)`.

`S.splitlines([keepends])`

Splits string `S` on line breaks, returning lines list. The result does not retain line break characters unless *`keepends`* is true.

Table 9. Python 3.X string method calls

S.capitalize()

S.casefold() (as of Python 3.3)

S.center(*width*, [, *fill*])

S.count(*sub* [, *start* [, *end*]])

S.encode([*encoding* [, *errors*]])

S.endswith(*suffix* [, *start* [, *end*]])

S.expandtabs([*tabsize*])

S.find(*sub* [, *start* [, *end*]])

S.format(**args*, ***kwargs*)

S.format_map(*mapping*) (as of Python 3.2)

S.index(*sub* [, *start* [, *end*]])

S.isalnum()

S.isalpha()

S.isdecimal()

S.isdigit()

S.isidentifier()

S.islower()

S.isnumeric()

S.isprintable()

S.isspace()

S.istitle()

S.isupper()

S.join(*iterable*)

S.ljust(*width* [, *fill*])

S.lower()

S.lstrip([*chars*])

S.maketrans(*x* [, *y* [, *z*]])

S.partition(*sep*)

S.replace(*old*, *new* [, *count*])

S.rfind(*sub* [, *start* [, *end*]])

S.rindex(*sub* [, *start* [, *end*]])

S.rjust(*width* [, *fill*])

S.rpartition(*sep*)

S.rsplit([*sep* [, *maxsplit*]])

S.rstrip([*chars*])

S.split([*sep* [, *maxsplit*]])

S.splitlines([*keepends*])

S.startswith(*prefix* [, *start* [, *end*]])

S.strip([*chars*])

S.swapcase()

S.title()

S.translate(*map*)

S.upper()

S.zfill(*width*)

LISTAS

LISTAS

`[]`
An empty list.

`[0, 1, 2, 3]`
A four-item list: indexes `0` through `3`.

`L = ['spam', [42, 3.1415], 1.23, {}]`
Nested sublists: `L[1][0]` fetches 42.

`L = list('spam')`
Creates a list of all items in any iterable, by calling the type constructor function.

`L = [x ** 2 for x in range(9)]`
Creates a list by collecting expression results during iteration (list comprehension).

DICCIONARIOS

DICCIONARIOS

`{}`

An empty dictionary (not a set).

`{'spam': 2, 'eggs': 3}`

A two-item dictionary: keys `'spam'` and `'eggs'`, values `2` and `3`.

`D = {'info': {42: 1, type(''): 2}, 'spam': []}`

Nested dictionaries: `D['info'][42]` fetches `1`.

`D = dict(name='Bob', age=45, job=('mgr', 'dev'))`

Creates a dictionary by passing keyword arguments to the type constructor.

`D = dict(zip('abc', [1, 2, 3]))`

Creates a dictionary by passing key/value tuple pairs to the type constructor.

`D = dict([['a', 1], ['b', 2], ['c', 3]])`

Same effect as prior line: accepts any iterable of keys and values.

`D = {c.upper(): ord(c) for c in 'spam'}`

Dictionary comprehension expression (in Python 3.X and 2.7). See "List comprehension expressions" for full syntax.

TUPLAS

TUPLAS

`()`
An empty tuple.

`(0,)`
A one-item tuple (not a simple expression).

`(0, 1, 2, 3)`
A four-item tuple.

`0, 1, 2, 3`
Another four-item tuple (same as prior line); not valid where comma or parentheses are otherwise significant (e.g., function arguments, 2.X prints).

`T = ('spam', (42, 'eggs'))`
Nested tuples: `T[1][1]` fetches `'eggs'`.

`T = tuple('spam')`
Creates a tuple of all items in any iterable, by calling the type constructor function.

ARCHIVOS

ARCHIVOS DE ENTRADA

infile = open(*filename*, 'r')
Creates input file object, connected to the named external file. *filename* is normally a string (e.g., 'data.txt'), and maps to the current working directory unless it includes a directory path prefix (e.g., r'c:\dir\data.txt'). Argument two gives file *mode*: 'r' reads text, 'rb' reads binary with no line-break translation. Mode is optional and defaults to 'r'. Python 3.X's open() also accepts an optional Unicode encoding name in text mode; 2.X's codecs.open() has similar tools.

infile.read()
Reads entire file, returning its contents as a single string. In text mode ('r'), line-ends are translated to '\n' by default. In binary mode ('rb'), the result string can contain nonprintable characters (e.g., '\0'). In Python 3.X, text mode *decodes* Unicode text into a str string, and binary mode returns unaltered content in a bytes string.

infile.readline()
Reads next line (through end-of-line marker); empty at end-of-file.

infile.readlines()
Reads entire file into a list of line strings. See also the file object's line iterator alternative, discussed in the next list item.

for *line* in *infile*
Uses the *line iterator* of file object *infile* to step through lines in the file automatically. Available in all iteration contexts, including for loops, map(), and comprehensions (e.g., [line.rstrip() for line in open('*filename*')]). The iteration for *line* in *infile* has an effect similar to for *line* in *infile*.readlines(), but the line iterator version fetches lines on demand instead of loading the entire file into memory, and so is more space-efficient.

ARCHIVOS DE SALIDA

outfile = open(*filename*, 'w')

Creates output file object, connected to external file named by *filename* (defined in the preceding section). Mode 'w' writes text, 'wb' writes binary data with no line-break translation. Python 3.X's open() also accepts an optional Unicode encoding name in text mode; 2.X's codecs.open() has similar tools.

outfile.write(*S*)

Writes all content in string *S* onto file, with no formatting applied. In text mode, '\n' is translated to the platform-specific line-end marker sequence by default. In binary mode, the string can contain nonprintable bytes (e.g., use 'a\0b\0c' to write a string of five bytes, two of which are binary zero). In Python 3.X, text mode requires str Unicode strings and *encodes* them when written, and binary mode expects and writes bytes strings unaltered.

outfile.writelines(*I*)

Writes all strings in iterable *I* onto file, not adding any line-end characters automatically.

OPERACIONES CON SECUENCIAS

Indexing: S[i]

- Fetches components at offsets (first item is at offset `0`).
- Negative indexes count backward from the end (last item is at offset `-1`).
- `S[0]` fetches the first item; `S[1]` fetches the second item.
- `S[-2]` fetches the second-to-last item (same as `S[len(S) - 2]`).

Slicing: S[i:j]

- Extracts contiguous sections of a sequence, from *i* through *j*–1.
- Slice boundaries *i* and *j* default to `0` and sequence length `len(S)`.
- `S[1:3]` fetches from offsets `1` up to, but not including, `3`.
- `S[1:]` fetches from offsets `1` through the end (`len(S)-1`).
- `S[:-1]` fetches from offsets `0` up to, but not including, the last item.
- `S[:]` makes a top-level (shallow) copy of sequence object `S`.

Extended slicing: S[i:j:k]

- The third item *k* is a stride (default 1), added to the offset of each item extracted.
- `S[::2]` is every other item in entire sequence `S`.
- `S[::-1]` is sequence `S` reversed.
- `S[4:1:-1]` fetches from offsets 4 up to, but not including, 1, reversed.

Slice assignment: S[i:j:k] = I

- Slice assignment is similar to deleting and then inserting where deleted.
- Iterables assigned to basic slices `S[i:j]` need not match in size.
- Iterables assigned to extended slices `S[i:j:k]` must match in size.

Other

- Concatenation, repetition, and slicing return new objects (though not always for tuples).

ESTRUCTURAS DE CONTROL

IF

```
import random
r = random.random()          # random number in [0,1)
if 0 <= r < 0.25:
   # move north
   y = y + d
elif 0.25 <= r < 0.5:
   # move east
   x = x + d
elif 0.5 <= r < 0.75:
   # move south
   y = y - d
else:
   # move west
   x = x - d
```

FOR

```
for i in [3, 4, 5, 6, 7]:
    print i**2
```

```
for i in range(3, 8, 1):
    print i**2
```

```
grocery = ['bread', 'milk', 'butter']

for item in enumerate(grocery):
  print(item)

print('\n')
for count, item in enumerate(grocery):
  print(count, item)
```

WHILE

```
while True:
    print('Who are you?')
    name = input()
    if name != 'Joe':
        continue
    print('Hello, Joe. What is the password? (It is a fish.)')
    password = input()
    if password == 'swordfish':
        break
print('Access granted.')
```

FUNCIONES

```
def y(t):
    v0 = 5                      # Initial velocity
    g = 9.81                    # Acceleration of gravity
    return v0*t - 0.5*g*t**2
```