15-110 CMU-Q: A (growing) reference card for Python

February 20, 2019[*][†]

Contents

Built-in data types

Data types have different properties and can be used with different operators depending on their mutable or immutable nature, and on their internal structure, that classify them as scalar (s) or non-scalar (ns).

Immutable data types:

- Numeric types:
  - `int` (s)
  - `float` (s)
  - `complex` (s)
- Boolean types:
  - `bool` (s)
- String types:
  - `str` (ns)
- Sequence types:
  - `tuple` (ns)
- `NoneType` (s)
- Function types:
  - `function` (s)

Mutable data types:

- Sequence types:
  - `list` (ns)

Notes

Assignment statements

An assignment is performed using the operator `=` in statements of the form:

`<var> = <literal> | <existing_var> | <expression>`

meaning that variable `<var>` gets assigned either the value of a given literal, or[1] the reference value of another (existing) variable, or the result of an expression (possibly involving multiple object types and operators). In all cases, `<var>` inherits the data type of the right-hand side of the assignment and gets the same value.

The effect of an assignment such as:

`<var> = <existing_var>`

changes depending on the data type of `<existing_var>`. If `<existing_var>` is a mutable object, `<var>` and

[*]Contact G. A. Di Caro for pointing out mistakes, missing information, and for suggestions (`gdicaro@cmu.edu`)

[†]Note: In the following, when the parameter of a function is optional, it is enclosed in `<>` brackets, e.g., `f(x,<y>)`. When a function returns something different than `None`, the notation `var = f()` is adopted, otherwise the function is plainly described as `f()`.

[1]The symbol `|` expresses the choice between different alternatives.

`<existing_var>` will now point to the same memory area, such as any further changes to the values of `<existing_var>` will result in changing the values of `<var>` and vice versa. In other words, `<var>` and `<existing_var>` become *aliases* (for referring to the same memory area where the numeric content is stored).

Notes

Numeric operators

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Real division: `/`; given that $x \div y = y \cdot n + r$, the operator returns $y \cdot n + r$ as a float
- Integer division: `//`; the operator returns n
- Module: `%`; the operator returns r
- Power: `**`

All numeric (arithmetic) operators, except power, apply to two operands in expressions of the form:

```
<operand1> numeric_operator <operand2>
```

The resulting type of the expression depends both on the operator and on the type of the operands.

Arithmetic operators can be used with numeric data types `int, float, complex`, as well as with `bool` object types based on the convention that `True` is automatically converted to `int` 1 and `False` to `int` 0.

Notes

`+` and `*` are also *overloaded operators* for operations on object types `str`, where `+` requires two string operands and `*` requires one string an one integer operand.

Notes

All the arithmetic operators also allow for *augmented assignment* forms to update (*in-place*) the value of an existing variable `x`, where `a` is the literal/variable value used for updating `x`:

- `x += a`
- `x -= a`
- `x *= a`
- `x /= a`
- `x //= a`
- `x %= a`
- `x **= a`

Notes

Relational operators

- x is equal to y: `x == y`
- x is not equal to y: `x != y`
- x is greater than y: `x > y`
- x is greater than or equal to y: `x >= y`
- x is less than y: `x < y`
- x is less than or equal to y: `x <= y`

where `x` and `y` are expressions that can evaluate to numbers, strings, boolean types: the relational operators are overloaded operators.

Notes

Notes

Logical operators

- `and`: `x and y`
- `or`: `x or y`
- `not`: `not x`

where `x`, `y` are expressions evaluating to booleans. The truth tables for the operators are:

`x`	`y`	`x and y`
0	0	0
1	0	0
0	1	0
1	1	1

`x`	`y`	`x or y`
0	0	0
1	0	1
0	1	1
1	1	1

`x`	`not x`
0	1
1	0

Notes

Precedence rules among operators

Priority level	Category	Operators
7 (high)	power	`**`
6	products	`* / // %`
5	sums	`+ -`
4	relational	`== != <= < >= >`
3	logical	`not`
2	logical	`and`
1(low)	logical	`or`

Useful standard library functions

- Type checking, casting and conversion:
  - `type(x)`
  - `int_var = int(x)`
  - `float_var = float(x)`
  - `bool_var = bool(x)`
  - `string_var = str(x)`
  - `list_var = list(x)`
  - `tuple_var = tuple(x)`

Notes

- Identity of an object, help on an object:
  - `addr = id(x)`
  - `help(x)`
- Input / Output:
  - `print(...)`
  - `string_var = input(msg)`

Notes

- Mathematical operations:
  - `val = min(values, <key>)`
  - `val = max(values, <key>)`
  - `val = sum(values)`
  - `seq_generator = range(from, to, step)`
  - `val = abs(v)`
  - `val = pow(x, y)`

Notes

- Operations on sequences or sets:
  - `length = len(s)`
  - `new_seq = sorted(seq, <key>, <reverse>)`
- Evaluation of strings as python expressions:
  - `expr_result = eval(expr)`
- Character to/from integer code conversions:
  - `character = chr(utf_numeric_code)`
  - `utf_numeric_code = ord(character)`

Notes

String operators

Strings are ordered sequences of characters, that can be defined using single, double, or triple quotes: `s = 'Hello'`, `s = ''Hello''`, `s = '''Hello'''`. Strings are non-scalar, immutable types.

- *Concatenation*: `+`, `+=` operators (overloaded)
- *Duplication*: `*`, `*=` operators (overloaded)
- *Comparison:* comparisons between strings using the relational operators are based on the UTF-8 encoding, that assigns an integer to each character.
- *Indexing*: `[index]` operator, where for a string of length n `i` is an integer taking values in the range from 0 to $n-1$ or from $-n$ to -1.
- *Slicing*: `[from:to]`, where for a string of length n `from` and `to` are integers taking values in the same range as above.
- *Slicing with a stride*: `[from:to:step]`, where for a string of length n `from`, `to` and `step` are integers taking values in the same range as above.
- *Membership*: `in`, `not in`, E.g., `part_of = s1 in s2`, where `s1` and `s2` are strings and `part_of` is a boolean.

Notes

String methods

- Case conversions:
  - `s_new = s.capitalize()`
  - `s_new = s.swapcase()`
  - `s_new = s.title()`
  - `s_new = s.lower()`
  - `s_new = s.upper()`

Notes

- Count, Find, and Replace:
  - `occurrences = s.count(substring, from, to)`

- `bool_var = s.endswith(suffix, from, to)`
- `bool_var = s.startswith(prefix, from, to)`
- `index = s.find(substring, from, to)`
- `index_rev = s.rfind(substring, from, to)`
- `s_new = s.replace(old, new)`
- `s_new = s.replace(old, new, max_times)`

Notes

- Manipulate strings and lists:
  - `string_joining_strings_in_list = s.join(seq)`
  - `list_of_substrings = s.split(<sep>=' ', <max_num>)`
  - `list_of_substrings = s.rsplit(<sep>=' ', <max_num>)`

Notes

- String classification by characters:
  - `bool_var = s.isalpha()`
  - `bool_var = s.isalnum()`
  - `bool_var = s.isdigit()`
  - `bool_var = s.isidentifier()`
  - `bool_var = s.islower()`
  - `bool_var = s.isupper()`
  - `bool_var = s.isprintable()`
  - `bool_var = s.isspace()`
  - `bool_var = s.istitle()`

Notes

List/tuple definition

Tuples and lists are non-scalar types representing ordered sequences of objects (any type). Tuples are *immutable*, lists are *mutable*.

- `tuple` examples: `(1,2,3); 1,'a',(2,3); (2,)`
- `list` examples: `[1,'h']; [[2,3],(2,5),'h']; [2]`
- definition of an empty tuple: `t = ()`
- definition of an empty list: `l = []`
- definition of an empty list of `n` elements all set to `0`: `l = [0]*n`
- definition of an empty list of `n` elements all set to `None`: `l = [None]*n`
- definition of an empty list of `n` (empty) lists: `l = [[]]*n`
- definition of a list of `n` lists each with one element of value 1: `l = [[1]]*n`
- definition of a new list as an alias of an existing list: `l_new = l_exist`
- definition of a new list as a clone of an existing list: `l_new = l_exist[:]`
- definition of a new list as a clone of an existing list: `l_new = l_exist.copy()`
- definition of a new list as a clone of an existing list: `l_new = []; l_new += l_exist`
- definition of a new list as a clone of an existing list: `l_new = []; l_new.extend(l_exist)`

Notes

List/tuple operators

- *Indexing*: `[index]` operator, where for a list/tuple of length n `index` is an integer taking values in the range from 0 to $n-1$ or from $-n$ to -1.

- *Slicing*: `[from:to]`, where for a list/tuple of length n `from` and `to` are integers taking values in the same range as above.

- *Slicing with a stride*: `[from:to:step]`, where for a tuple/list of length n `from`, `to` and `step` are integers taking values in the same range as above.

- *Nested indexing:* `[i][j][k]...[n]`, when a list/tuple `l` contains elements `l[i]` that are in turn lists/tuples, the j-th element of the list/tuple `l[i]` can be accessed using the notation `l[i][j]`. The notation can be iterated if there are multiple nested lists/tuples. E.g., `l[i][j][k]` accesses the k-th element of the j-th list element of the i-th list element of list `l`.

  `x = [ [1,2,3], [4,5,6] ]`; `x[0][1]` is `2`, while `x[1][1]` is `5`

  `x = [ [ [1, 'a'], [2, 'c'] ], [4,5,6] ]`; `x[0][1]` is `[2, 'c']`, `x[0][1][1]` is `'c'`, `x[1][1][1]` doesn't exist

- *Concatenation*: `+`, `+=` operators (overloaded), where for lists `+=` is an in-place operator while `+` is not

- *Duplication*: `*`, `*=` operators (overloaded), where for lists `*=` is an in-place operator while `*` is not

- *Comparison:* comparisons between two tuples/lists can be done using the relational operators; two tuples/lists are equal (`==`) if they have the same content, while they are different (`!=`) if their content is different; the comparison for greater/less than is performed between the elements at position index 0 of the two tuples/list based on their data type, that must be either the same or allow for comparison (e.g., both numeric types); if the two elements at position 0 are the same, the elements at position 1 of the two tuples/list are considered, and so on until the comparison can be precisely assessed.

  `[0,1] == [0,1]` returns `True`

  `[0,1] < [1,3,7,8]` returns `True`

  `[0,'a'] < [0,'b']` returns `True`

  `[[1,2],'a'] > [[0,1],'b']` returns `True`

- *Membership*: `in`, `not in`, where `l1 in l2`, evaluates to `True` if `l1` is a sub-list of `l2`.

  `[4,5] in [[1,2,3],[4,5]]` returns `True`

  `[4] in [1,2,3,4,5]` returns `False`

  `4 in [1,2,3,4,5]` returns `True`

Notes

List/tuple methods

- `l.append(item)`
- `l.insert(index)`
- `l.extend(existing_list)`
- `l.remove(item)`
- `removed_item = l.pop(index)`
- `occurrences = l.count(item)`
- `index_of_first_occurrence = l.index(item)`
- `l.sort(<key>, <reverse>)`
- `l.reverse()`
- `l_new = l.copy()`

Notes

Constructs for conditional decisions

-
  ```
  if boolean_expression_is_true:
     do_something
  ```

-
  ```
  if boolean_expression_is_true:
     do_something
  else:
     do_something_else
  ```

-
  ```
  if boolean_expression_1_is_true:
     do_something_1
  elif boolean_expression_2_is_true:
     do_something_else_2
  elif boolean_expression_3_is_true:
     do_something_else_3
  ...
  ```

-
  ```
  if boolean_expression_1_is_true:
     do_something_1
  elif boolean_expression_2_is_true:
     do_something_else_2
  elif boolean_expression_3_is_true:
     do_something_else_3
  else:
     do_something_else
  ```

Notes

Constructs for iteration: `for` loops

Definite loops invoked with the following syntax:

```
for variable in sequence:
   do_actions
```

-
  ```
  x = [1,4,5]
  prod = 1
  for i in x:
     prod *= i
  ```
  `prod` is 20 at the end of the loop

-
  ```
  x = ['a','b','d']
  string = ''
  for s in x:
     string += s
  ```
  `string` is 'abd' at the end of the loop

-
  ```
  add_odd = 0
  for i in range(1,12,2):
     add_odd += i
  ```
  `add_odd` is 25 at the end of the loop

-
  ```
  for i in range(1,10):
     print(''Hello!'')
  ```
  "Hello!" is output for 10 times

-
  ```
  cars = [['Cor', 2015],['Sen', 2014],['Cam', 2016]]
  years = [2015, 2016, 2018]
  count = 0
  for c in cars:
      for y in years:
         if c[1] == y:
           count += 1
  ```
  `count` is 2 at the end of the two nested loops

-
  ```
  m = [[]]*2
  m[0] = [1,2,3]
  m[1] = [4,5,6,7]
  add = 0
  for i in range(len(m)):
      for j in range(len(m[i])):
          add += m[i][j]
  ```
  `add` is 28 at the end of the two nested loops

Notes

Constructs for iteration: `while` loops

Indefinite loops invoked with the following syntax:

```
while boolean_condition_is_true:
   do_actions
```

-
  ```
  v = 5
  discounts = []
  while v > 1:
     v *= 0.95
     discounts.append(v)
  ```
  `discounts` contains 32 values between 1 and 5 at the end of the loop.

– ```
  add = 0
  while True:
     v = input(‘‘Give a positive integer number:’’)
     if v.isdigit():
        if int(v) == 0:
           break
        else
           add += int(v)
  ```
  a potentially infinite loop, it ends when input is 0.

Notes

## Statements for iteration: `continue`, `break`

- `continue`: Conditionally *skip* an iteration body

  ```
  numbers = [3, -1, 4, 0, 5]
  percent = []
  for n in numbers:
     if n <= 0:
        print(‘‘Value not allowed!’’):
        continue
     percent.append(n / 100)
     print(’’Percentage value:’’, n / 100)
  ```

  at the end of the loop `percentage` contains 3 elements, 0.03, 0.04, 0.05

- `break`: Conditionally *break out* from a loop

  ```
  numbers = [3, -1, 4, 0, 5]
  percent = []
  for n in numbers:
     if n <= 0:
        print(‘‘Error, process interrupted!’’):
        break
     percent.append(n / 100)
     print(‘‘Percentage value:’’, n / 100)
  ```

  at the end of the loop `percentage` contains one element, 0.03

Notes

Functions: definition, args, returns

A function is a *named procedure* that can be invoked anywhere in the program. It can have or not *input arguments*, and always *returns* something by means of the `return` statement. If no `return` statement is included in the function definition, the statement `return None` is executed by default. The general syntax for defining a function is:

```
def function_name(arguments):
    function_body
    return something
```

- *Variables* defined in the function body have a *local scope*, they exist in the program only for the time of execution of the function. As such, they are not accessible outside the function, as shown in the example:

  ```
  def quadratic(x):
      xx = x * x
      c = 10
      return (y + x + c)

  q = quadratic(3)
  print(q, xx, c)
  ```

  this throws an error since `xx` and `c` are not accessible in the program invoking `quadratic(3)`.

- *Function’s arguments* are also local function variables. However, if an argument variable is of a *mutable type* (e.g., a list), changes to the variable inside the function’s body affect the value of the variable also in the calling (sub)program (i.e., outside of the function body).

  ```
  def discount(base, discount, values):
      base *= discount
      for i in range(len(values)):
          values[i] *= base
      return

  v = [2,3,4]
  b = 1
  discount(b, 0.5, v)
  ```

  when `discount(b, 0.5, v)` returns, the value of the variable `v` is not anymore `[2,3,4]` but rather
  ```

[1.0, 1.5, 2.0], because v is of mutable type list and the values of v have been changed inside the function (through the alias values); instead, the value of the variable b is still 1, because this variable is of type int, which is immutable.

Notes

Functions: positional, keyword, default, arbitrary, function arguments

- *Positional arguments*: arguments are matched by *position* as given in the function definition, order matters.

  ```
  def quadratic_roots(a, b, c):
     x1 = -b / (2 * a)
     x2 = sqrt(b**2 - (4 * a * c)) / (2 * a)
     return (x1 + x2), (x1 - x2)
  ```

  calling quadratic_roots(1,3,2) produces a different effect of calling quadratic_roots(2,1,3).

- *Keyword arguments*: arguments are matched by the *name* given in the function definition.

  The call quadratic_roots(a=1, b=3, c=2) produces the same effect as the calls quadratic_roots(b=3, c=2, a=1), quadratic_roots(c=3, a=1, b=3).

- *Mixed arguments:* Positional and keyword arguments can be used together, but the positional ones must precede the keyword ones.

  quadratic_roots(1, c=3, b=2) has the same effects of quadratic_roots(c=3, a=1, b=2) and of quadratic_roots(1, 3, 2), while calling quadratic_roots(b=2, a=1, 3) is not allowed.

- *Default values for arguments:* if the value for an argument is not given explicitly when the function is called, the argument takes its *default value*; the use of default values makes an argument *optional* in practice.

  ```
  def discount(values, d = 0.5):
     for i in range(len(values)):
        values[i] *= d
     return values
  ```

  when called as discount([1,3,5]) the return list is [0.5, 1.5, 2.5], while invoking the function as discount([1,3,5], 1.0) results in [1.0, 3.0, 5.0]; if also the argument values has a default value, defined as def discount(values = [1,1,1], d = 0.5), the call discount() returns [0.5, 0.5, 0.5].

- *Arbitrary number of arguments:* An arbitrary sequence of arguments can be passed with the notation my_function(*arguments).

  ```
  def longest_len(*strings):
      max = 0
      for s in strings:
         if len(s) > max:
            max = len(s)
      return max
  ```

  the function longest_len can be called with an arbitrary number of strings as arguments, e.g., longest_len('apple', 'orange', 'peach'), longest_len('aa','abc','z','xyz','uu','oooo'); positional and keyword arguments can be also used together with arbitrary arguments, but in the definition need to be placed before the arbitrary ones.

- *Function arguments:* The reference to an *existing function* f can be passed as an argument following the regular notation.

  ```
  def parabola(x):
      return -x**2

  def cubic(x):
      return x**3

  def find_max_over_interval(f, a, b, delta):
      evaluation_points = int(abs(b-a) / delta)
      max_val = f(a)
      x_max = a
      x = a + delta
      for i in range(1, evaluation_points):
         if f(x) > max_val:
            max_val = f(x)
            x_max = x
         x += delta
      return (x_max, max_val)

  find_max_over_interval(parabola, -2, 2, 0.1)
  find_max_over_interval(cubic, -3, 1, 0.1)
  ```

  the function find_max_over_interval can be called with any existing function passed by reference as the argument f, in turn f can be used to regulary invoke the input function inside the function body.

Docstrings and inline comments

For sharing and reusing purposes, functions need to have a *specification*, which is a *string constant* placed right below

Notes

the `def` statement of the function. This string constant is called a *docstring*. It is common practice to use triple single quotes `'''` to write the string constant with the specification text since this allows to write a multi-line string.

A specification needs to include a description of the input parameters, their type and accepted values, any other actions that need to be in place before calling the function, the effects of the function, describing the returned values and possibly modifications through the parameters. A simplified example is shown below.

```
def add(x, y):
   '''Sum two objects x and y.
      The sum is returned. Type depends on x and y.
      An error is thrown if the + operator
      cannot be used between the two passed objects.'''
return x + y
```

The `help(f)` function, when invoked with the name of another function `f` as an argument, returns the description of the function and of its parameters as given by the function developer through the specification in the docstring.

In addition to the specification, it is good practice to provide *inline additional comments* to document the text. This can be done by using either `#` for single-line comments or string constants for multi-line comments.

Notes

Import statements

- `import module_name`: load the entire module, need to refer to module's functions as *methods* using the dot notation, `module_name.function_name()`.
- `import module_name as alias_name`: define a name *alias* for `module_name`, such as module's functions have to be referred to as methods using the dot notation and the alias, `alias_name.function_name()`.
- `from module_name import function_name`: instead of loading the entire module only the *specified function* from the module is being loaded (copied) into the current program, no need to use dot notation to refer to the function.
- `from module_name import f_1, f_2, f_3`: instead of loading the entire module only the *specified functions* from the module are being loaded (copied) into the current program, no need to use dot notation to refer to the functions.
- `from module_name import *`: *all* functions from the module are copied into the current program, no need to use dot notation to refer to any of the functions; possible issues with variable/function names clashing with those of the calling program.

Notes