COMP10001 Foundations of Computing
Semester 1, 2019
Python Essentials

— VERSION: 1586, DATE: MAY 24, 2019 —

Printing

```
>>> print(1 + 2 + 3)
6
>>> print("Tim")
Tim
>>> print("apples")
apples
>>> print("This is a string")
This is a string
>>> print(2, "apples")
2 apples
```

Comments

Anything after a (non-quoted) hash (#) in a line of code is ignored by Python, and can be used to comment your program/comment out code.

Variables, Literals and Types

Variable names in Python must start with a character or underscore (_), and can also include numbers (except for the initial character); they are case-sensitive and cannot be reserved words:

```
and del for is raise
assert elif from lambda return
break else global not try
class except if or while
continue exec import pass yield
def finally in print
```

Values can be assigned to variables with =, e.g.:

```
variable_name = value
>>> myname = "Tim"
>>> mynumber = 100
```

Literals are constant values of a given type:

```
>>> 2
2
>>> 3.0
3.0
>>> "apple"
'apple'
```

All variables and literals have a type:

- `int` = integer (whole number)
- `str` = string (chunk of text)
- `float` = floating point number (real number)

- `bool` = Boolean (`True` or `False`)
- `list` = sequence of values; values can be of different types, and can be modified
- `tuple` = sequence of values; values can be of different types, but can't be modified
- `dict` = dictionary (collection of keys and associated values; keys must be unique, and can't be lists or dictionaries; values can be of different types)
- `set` = set (collection of keys; keys must be unique, and can't be lists or dictionaries; basically a value-less dictionary)

To find out the type of a variable:

```
>>> type(my_name)
<type 'str'>
```

To convert to a given type, use the type name (`int`, `str`, `float`, etc.):

```
>>> int(2.0)
2
>>> float("1.8") # convert string to floating point number
1.8
```

noting that this is not always possible:

```
>>> int("two")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'two'
```

It is also possible to convert a single character into its underlying Unicode code point representation, and from an integer to a character:

```
>>> chr(59)
';'
>>> ord('A')
65
```

Arithmetic operators can be used over numeric values (`int` and `float`):

```
+ - * / % **
```

Each of these can be combined into a compound assignment operator:

```
>>> a = 2
>>> print(a)
2
>>> a += 3 # equivalent to a = a + 3
>>> print(a)
5
>>> a **= 2 # equivalent to a = a ** 2
>>> print(a)
25
```

The operators `+` and `*` can also be used over strings:

```
>>> a = "2"
>>> b = "3"
>>> print(a+b)
'23'
>>> a *= 4
>>> print(a)
'2222'
```

To check whether a string is contained within another string, use `in`:

```
>>> "a" in "abracadabra"
True
>>> "abb" in "abracadabra"
False
```

Lists

Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type.

```
>>> a = ['spam', 'eggs', 100, 1234.5]
>>> a
['spam', 'eggs', 100, 1234.5]
```

Individual values in a list can be indexed. List indices start at 0, and lists can be sliced (`[x:y]`), concatenated (+), repeated (*), etc:

```
>>> a[0]
'spam'
>>> a[3]
1234.5
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 2*a[:2] + ["fin"]
['spam', 'eggs', 'spam', 'eggs', 'fin']
```

Checking for the existence of a given value in a list:

```
>>> a = ['spam', 'eggs', 'bacon']
>>> "spam" in a
True
>>> "fin" in a
False
```

You can also calculate the length of a list via `len()`:

```
>>> len(a)
4
```

or return a sorted version of a list via `sorted()`:

```
>>> a = ['spam', 'eggs', 'bacon']
>>> print(sorted(a))
['bacon', 'eggs', 'spam']
>>> print(a) # note that the original is unchanged
['spam', 'eggs', 'bacon']
```

It is also possible to generate a list of values (e.g. to iterate over a list in a for loop) with `range()`:

```
>>> list(range(5)) # return a list from 0 (implicitly) to 5, non-inclusive
[0, 1, 2, 3, 4]
>>> list(range(1,5)) # return a list from 1 to 5, non-inclusive
[1, 2, 3, 4]
>>> list(range(5,1,-1)) # return a list from 5 to 1, non-inclusive, by steps of -1
[5, 4, 3, 2]
```

To convert a string to a list of component characters, use `list()`:

```
>>> list("ad hoc")
['a', 'd', ' ', 'h', 'o', 'c']
```

All of the above operators and functions equally apply to strings (where the “items” are the individual characters in the string) and tuples (identical to lists except that they are initialised with parentheses, and it is not possible to change the contents of a tuple). The methods described below (`append()`, `remove()` and `sort()`) are in-place and therefore do not apply to strings or tuples (as they are “immutable”).

You can append values to a list, and remove items from a list (by value or index):

```
>>> a = ['spam', 'eggs', 'bacon']
>>> a.append('sausages')
>>> print(a)
['spam', 'eggs', 'bacon', 'sausages']
>>> a.remove('eggs')
>>> print(a)
['spam', 'bacon', 'sausages']
>>> a.pop(1)
'bacon'
>>> print(a)
['spam', 'sausages']
```

Finally, you can sort a list ”in-place” using `sort()`:

```
>>> a = ['spam', 'eggs', 'bacon']
>>> a.sort()
>>> print(a) # note that the original is changed
['bacon', 'eggs', 'spam']
```

Strings

A string is a sequence of characters — alphabetic, numeric, symbols, spaces — that Python stores as a index-able sequence. Elements common to lists are covered above. Here, we cover only string-specific syntax and methods.

- Strings can be delimited by either double quotes (e.g. `"Tim"`) or single quotes (e.g. `'Tim'`).
- Multiline values can be assigned to a string by triple-quoting:

  ```
  silly = """Two
  Lines"""
  ```

- Another approach is to embed “escaped” newline characters into the string, e.g.:

  ```
  silly = "Two\nLines"
  ```

  Another common special character is `\t` for tab.

Some useful string methods are:

```
s.upper() # return s in all uppercase
```

```
s.lower() # return s in all lowercase
```

```
s.strip(STRING) # return s with all instances of characters in STRING
                # (whitespace if STRING is not supplied) removed from
                # the *ends* of s
```

```
s.split(STRING) # return a list of STRING-delimited
                # (space if STRING is not supplied) substrings in s
```

Remember that strings are immutable so to make a change “stick” you have to do, e.g.:

```
>>> s = s.upper()
```

It is also possible to generate a string with values of different types inserted into it using f-strings, e.g.:

```
>>> mystring = f"{x:2.2f} is a float and {y} is an int"
```

You can optionally stipulate the output format for different argument types by inserting a colon, a type declaration and options for that type between the colon and the type declaration. Type declarations of note are:

```
%d # integer
```

```
%f # float
```

```
%s # string
```

```
%% # prints the percentage sign
```

Getting user input

```
>>> users_name = input("Enter your name: ")
>>> print("Your name is: " + users_name)
```

Files

Data can be read in from a file by first creating a file "object":

```
>>> fp = open(FILENAME)
```

and then using the following methods:

```
fp.read() # return the entire contents of the file as a single string
```

```
fp.readlines() # return the entire contents of the file as a list of
               # strings (one per line)
```

Equivalently, these methods can be called directly over an ”anonymous” file handle:

```
>>> text = open(FILENAME).read() # store the entire contents of FILENAME in text
```

It is also possible to write data out to a file, by opening the file in ”write” mode:

```
>>> fp = open(FILENAME,'w')
```

This will write over the original contents of the file (if there were any); it is also possible to append to an existing file using `'a'` instead of `'w'`. In both cases, the following methods are used to write text to the open file object:

```
fp.write(TEXT) # write TEXT to fp, appending it to whatever is currently there
```

```
fp.close() # close fp so no more text can be written out to it
```

Conditionals

Python supports a number of conditional tests that return a Boolean truth value (True or False):

```
== <= >= > < !=
```

for example:

```
>>> 2 == 1 # test whether 1 is equal to 2
False
>>> "apple" != "banana" # test whether "apple" and "banana" are not equal
True
>>> "z" < "a"
False
>>> "z" >= "a"
True
```

These can be combined with logic operators such as `and`, `not` and `or`, e.g.:

```
>>> 2 == 1 and "apple" != "banana"
False
```

The `if:elif:else` statement allows you to test for the truth of a condition (of the type specified above), and execute a block of code depending on whether the condition/expression is `True` or `False`:

```
>>> if <condition1>:
>>>     block 1 of code
>>> elif <condition2>:
>>>     block 2 of code
>>> elif <condition3>:
>>>     block 3 of code
...
>>> else:
>>>     block n of code
```

Iteration (Loops)

The `for` statement allows you to iterate over a sequence of values:

```
>>> for [element] in [sequence]:
>>>     block of code
```

The `while` statement allows you to repeat a block of code until the stipulated logical expression evaluates to `False`:

```
>>> while [expression is True]:
>>>     block of code
```

Dictionaries

Dictionaries are like lists, but instead of indexing with a number, you index the list with a unique key. This has the advantage that you don't need to remember where a value is in the dictionary, you only need to remember what it's called.

Dictionary initialisation:

```
>>> dictionary = {}
>>> dictionary = { "capuccino" : 2.75, "chai" : 3.50 }
```

Iterating over a dictionary:

```
>>> dictionary = { "capuccino" : 2.75, "chai" : 3.50 }
>>> for key in dictionary:
...     print(key)
capuccino
chai
```

Looking up items in a dictionary:

```
>>> dictionary["capuccino"]
2.75
```

Adding items to a dictionary:

```
>>> dictionary["tea"] = 2.50
```

Checking for the existence of a given key in the dictionary:

```
>>> "tea" in dictionary # is "tea" contained in dictionary (True or False)?
```

Useful dictionary methods:

```
dictionary.pop(KEY) # remove KEY (and return its VALUE)
```

```
dictionary.keys() # return list of keys
```

```
dictionary.values() # return list of values
```

```
dictionary.items() # return list of (KEY,VALUE) tuples contained in dictionary
```

Note that the order in which keys/values are returned from the dictionary is generally not the order in which they were inserted.

Functions

A function is essentially a prewritten block of code that is often run over a list of input arguments. The main elements of a function are a name, a list of arguments (zero or more) and optionally a single return value:

```
>>> def NAME(ARGUMENTS):
>>>     statements
>>>     return VALUE
```

When lists and dictionaries are passed as an argument to a function and mutated within the function, the original list/dictionary is modified:

```
>>> dictionary = { "capuccino" : 2.75, "chai" : 3.50 }
>>> def reprice_all(dict,price):
...     for key in dict:
...         dict[key] = price
...
>>> reprice_all(dictionary,3.00)
>>> print(dictionary)
{'capuccino': 3.0, 'chai': 3.0}
```

This is not the case with other basic types (`int`, `float`, `bool`; recall that `str` and `tuple` are immutable, so the question is moot):

```
>>> myprice = 2.50
>>> def reprice(price):
...     price = 3.00
...
>>> reprice(myprice)
>>> print(myprice)
2.5
```

Libraries

It is possible to call libraries from within your code by "importing" the library, and accessing the methods it defines, e.g.:

```
>>> import math
>>> print(math.pi)
3.141592653589793
```

You can also selectively import particular methods from libraries and call them as per user-defined functions (without the LIBRARY.METHOD syntax) via:

```
>>> from math import pi
>>> print(pi)
3.141592653589793
```

The following are libraries of particular note.

collections

Library containing a number of advanced data types, notable amongst which is `defaultdict`, which initialises any call to a non-existent key with the default value for the type that is stipulated in the `defaultdict` call, e.g.:

```
>>> from collections import defaultdict
>>> word_count = defaultdict(int)
>>> print(word_count["banana"])
0
>>> word_count["durian"] += 1
>>> print(word_count["durian"])
1
```

csv

Library for parsing data from CSV files. Important methods:

```
csv.reader(FP) # return a reader object which can iteratively parse
               # each line in a CSV file accessible via FP (file object or list)
```

```
csvreader.next(READER) # return the next parsed line from READER as a list
```

For example:

```
>>> import csv
>>> csvdata = csv.reader(open("vic_visitors.csv"))
>>> header = csvdata.next()
>>> print(header)
["Victoria's Regions", '2004', '2005', '2006', '2007']
>>> for row in csvdata:
...     print(row)
['Gippsland', '63354', '47083', '51517', '54872']
['Goldfields', '42625', '36358', '30358', '36486']
['Grampians', '64092', '41773', '29102', '38058']
['Great Ocean Road', '185456', '153925', '150268', '167458']
['Melbourne', '1236417', '1263118', '1357800', '1377291']
```

Alternatively, use `DictReader` to index the data by the column headers in a dictionary:

```
>>> import csv
>>> csvdata = csv.DictReader(open("vic_visitors.csv"))
>>> for row in csvdata:
...    print(row)
...
{'2006': '51517', "Victoria's Regions": 'Gippsland', '2007': '54872', ...
{'2006': '30358', "Victoria's Regions": 'Goldfields', '2007': '36486', ...
{'2006': '29102', "Victoria's Regions": 'Grampians', '2007': '38058', ...
{'2006': '150268', "Victoria's Regions": 'Great Ocean Road', '2007': ...
{'2006': '1357800', "Victoria's Regions": 'Melbourne', '2007': '1377291', ...
```

Exceptions and Exception Handling

Whenever a syntax or run-time error occurs in Python, it takes the form of an "exception" being "raised". Python has many in-built exceptions, and the names are usually fairly self-documenting. Some of the more common ones are:

- `SyntaxError` — syntax error

  ```
  >>> a = 2 +
    File "<stdin>", line 1
  ```

```
    a = 2 +
          ^
SyntaxError: invalid syntax
```

- `NameError` — an undefined variable has been used

```
>>> b = a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

- `IndexError` — an out-of-range list or tuple index has been used

```
>>> a = [1,2,3]
>>> a[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

- `KeyError` — a non-existent dictionary key has been used

```
>>> a = {1: 2}
>>> a[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 2
```

- `ValueError` — the value of an object is invalid for that type

```
>>> a = int("a")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'a'
```

- `TypeError` — an operation has been attempted which is invalid for the type of the target object or object type combination

```
>>> a = 1 + "2"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

As can be seen in the examples above, when an exception is raised, the exception type is returned along with an error message describing the nature of the error.

One simple way of testing code is with an `assert` statement, where the code in the statement is evaluated and an exception (an `AssertionError` by default, although it is possible to specify an alternative error type) is raised if it does not evaluate to `True`:

```
>>> assert 2 == 2.0
>>> assert 2 == 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

It is possible to "catch" exceptions within Python by using a `try ... except ... finally` construction. Python will run the code block associated with the `try` clause, and if an exception is raised, rather than exiting your code with a run-time error, the block will terminate and the exception will be passed on to the `except` clause(s). Much like `elif` clauses, it is possible to have multiple `except` clauses with different exception types, in which case the block of code associated with that clause will run if and only if the raised exception is of the type stipulated in the `except` clause. You may also optionally stipulate an empty `except` clause, which will catch any exception that has been caught by preceding `except` clauses (much like `else`). The `finally` clause is executed at the very end of the code, irrespective of whether an exception was raised and handled or not. As an example:

```
while True:
    try:
        (x,y) = raw_input("Enter the X,Y coordinates of your shot:").split(",")
        (x,y) = (int(x),int(y))
        assert(0 <= x < boardsize and 0 <= y < boardsize)
        return (x,y)
    except ValueError:
        print("ERROR: indices must both be integers")
    except AssertionError:
        print("ERROR: indices must be in range [0,%d)" % boardsize)
    except:
        print("ERROR: unrecognised input")
```