1 Language reference

A comment starts with a hash character #

1.1 Conditionals

```
if expression:
  commands
elif expression:
  commands:
else:
  commands
```

1.2 Loops

```
while expression:    for element in list:
  commands             commands
else:                else:
  commands             commands
```

Note: A `break` statement executed in the first suite terminates the loop without executing the `else` clause's suite.

To terminate immediately the nearest enclosing loop, use `break`. To skip to the next iteration instead, use `continue`.

Loopnig on dictionaries and sets loops on keys.

2 Built-in types

2.1 Booleans

There are three Boolean operations: `or`, `and` and `not`. There are eight comparison operations, which can be chained and have higher priority than the Boolean operations: <, <=, >, >=, ==, `!=`, `is`, `is not`.

2.2 Numeric types

Python supports three numeric types: integers, floats and complex numbers. Integers have unlimited precision. Complex numbers have real (`z.real`) and imaginary part (`z.imag`), both are floats.

Numeric types support the following operations, sorted by ascending priority:

1. +, -, *, /,
2. // (floored quotient), % (remainder),
3. `abs(x)`,
4. `int(x)`, `float(x)`, `complex(re, im)`,
5. `c.conjugate`,
6. `divmod` (quotient and remanider),
7. `pow(x, y) == x**y`.

Bitwise operations, | (or), ^ (xor), & (and), <<, >> (shifts) and ~ (inversion) only make sense for integers.

2.3 Iterator types

To-do.

2.4 Sequence types

There are three basic sequence types: lists, tuples, and range objects. Common sequence operations include:

1. `x in s` and `x not in s`,
2. `s + t`, the concatenation,
3. `s * n`, adding s to itself n times,
4. `s[i]`, *i*th item of s and `s[i:j:k]`, slicing,
5. `len(s)`, `min(s)`, `max(s)`,
6. `s.index(x)`, index of the first occurrence,
7. `s.count(x)`, total number of occurrences.

Mutable sequence types support following operations:

1. `del s[i:j:k]` removes the elements,
2. `s.append(x)`,
3. `s.clear()` removes all elements,
4. `s.copy()` creates a shallow copy,
5. `s.extend(t)`,
6. `s *= n`,
7. `s.insert(i, x)`,
8. `s.pop([i])` removes an item, retrieves it,
9. `s.remove(x)`,
10. `s.reverse()` reverses the items in place.

Lists are mutable and typically store collections of homogeneous items. They provide the method `sort`, which sorts the list in place. May be constructed using:

1. a pair of square brackets: `[]`, `[1, 2, 3]`,
2. a comprehension: `[x for x in iter]`,
3. the type constructor: `list(iter)`.

Tuples are immutable and typically store collections of heterogeneous data. May be constructed using:

1. a pair of parantheses: `()`, `(1,)`,
2. the `tuple(iterable)` built-in.

Ranges are immutable sequence of numbers. May be constructed using: `range(stop)` or the longer version, `range(start, stop, step)`.

Useful:

```
for index, value in enumerate(list):
    commands
```

2.5 Text sequence type

Strings are immutable sequences of Unicode code points surrounded by single, double or triple quotes.

1. Case conversion:
   (a) `.capitalize()`,
   (b) `.casefold()` (aggresive lowercase),
   (c) `.lower(), lstrip()`,
   (d) `.swapcase()`,
   (e) `.title()`,
   (f) `.upper()`.
2. Justification:
   (a) `.center(width, fillchair)`,
   (b) `.ljust(width, fillchair)`,
   (c) `.rjust(width, fillchar)`,
   (d) `.zfill()` left fills with zeroes.
3. Search and/or replace:
   (a) `.count(sub, start, end)`,
   (b) `.find(sub, start, end)`,
   (c) `.rfind(sub, start, end)`,
   (d) `.index(sub)` like `find`, raises error when text is not found (not −1),
   (e) `.rindex(sub, start, end)`,
   (f) `.replace(old, new, count)`,
   (g) `.endswith(suffix)`,
   (h) `.startswith(prefix)`,
4. Joining and splitting:
   (a) `.join(iterable)`,
   (b) `.partition(sep)` returns 3-tuple,
   (c) `.rpartition(sep)` from right,
   (d) `.split()` splits into the same type,
   (e) `.rsplit()` splits from right,
   (f) `.splitlines()`,
5. `.format(*args, **kwargs)`,
6. `.strip()` removes leading/trailing chars,
7. `.rstrip()` removes trailing chars only.

2.6 Set types

A set object is an unordered collection of distinct hashable objects. The `set` is mutable, `frozenset` is not. Sets support following operations.

1. `len(s)`, `x in s`, `x not in s`,
2. `copy()`: returns a shallow copy,
3. `.isdisjoint(other)`,
4. `.issubset(other)`: `set1 <= set2`,
5. `.issuperset(other)`: `set1 >= set2`,
6. `set1 < set2`: being proper subset,
7. `set1 > set2`: being proper superset,
8. `union(*sets)`: `set1 | set2 | ...`,
9. `intersection(*sets)`: `set1 & set2 ...`,
10. `difference(*sets)`: `set1 - set2 - ...`,
11. `symmetric_difference(other)`: `set1 ^ set2`.

Mutable sets support following operations as well:

1. `.add()`,
2. `.remove()` raises errors,
3. `.discard()` doesn't raise errors,
4. `.pop()` removes and returns an element,
5. `.clear()` removes all elements.

2.7 Mapping types (`dict`)

A (mutable) dictionary maps hashable values to arbitrary objects.

These are the operations that dictionaries support:

1. `len(d)`, `d[key]`, `key in d`, `key not in d`,
2. `del d[key]` removes `d[key]` from `d`.
3. `iter(d)` returns an iterator over keys,
4. `clear()` removes all items,
5. `copy()` returns a shallow copy,
6. `pop(key)`, `popitem()`,
7. `items()` returns keys and values,
8. `keys()` returns keys,
9. `values`,
10. `update()`.

3 Text processing services

The modules described in this chapter provide a wide range of string manipulation operations and other text processing services.

3.1 string

Common string operations

3.2 Regular expressions (`re`)

- `search(pattern, string)` scans through a string, looking for any place where the regex matches.
- `match(pattern, string)` checks whether the regex matches at the beginning of the string.
- `findall(pattern, string)` returns all substrings where the regex matches as a list
- `finditer(pattern, string)` returns all substrings where the regex matches as an iterator.

3.3 difflib

Helpers for computing deltas

3.4 textwrap

Text wrapping and filling

3.5 unicodedata

Unicode Database

3.6 stringprep

Internet String Preparation

3.7 readline

GNU readline interface

3.8 rlcompleter

Completion function for GNU readline