

Virtual Environment setup

```
sudo pip install virtualenv # sudo may be optional

mkdir testenv

cd testenv

virtualenv -p $( which python3.4 ) .lpvenv

source .lpvenv v/b in/ activate
```

This set of commands is for a unix-like environment, but a similar setup can be done on Windows.

Core types

Integers: unlimited	cast: <code>int(value)</code>
Boolean: True/False	cast: <code>bool(value)</code>
	any non-zero is True
Real: 64 bits	
Decimal: arbitrary precision	<code>from decimal import Decimal</code>
Complex: $1 + 2j$	methods: <code>real</code> , <code>imag</code>

Strings

<code>" "</code>	empty string
<code>len(value)</code>	built-in to get the length
<code>s[start:stop:step]</code>	slicing, start/stop/step are optional, stop is non-inclusive
<code>s.encode('utf-8')</code>	utf-8 encoded version of s, bytes object
<code>b"string"</code>	bytes object

Tuples

<code>()</code>	empty tuple
<code>(val,)</code>	1-element tuple, comma is required
<code>(a,b,c)</code>	generic syntax for tuples
<code>a,b,c = 1,2,3</code>	implicit on assignment

Tuples are immutable

Lists

<code>[]</code> or <code>list()</code>	empty list
<code>list((create, from, a, tuple) _)</code>	
<code>[x + 5 for x in (1,2,3)]</code>	definition by comprehension
<code>+</code>	concatenate lists

Lists (cont)

* repeat list

Notable methods: `append(x)`, `count(x)`, `extend(list)`, `index(x)`, `insert(pos,value)`, `pop()`, `pop(pos)`, `remove(val)`, `reverse()`, `sort()`, `clear()`

Notable functions: `min(list)`, `max(list)`, `sum(list)`, `len(list)`

Dictionaries

<code>dict()</code> or <code>{}</code>	empty dictionary
Equivalent definitions	<code>dict(A=1, Z=-1)</code> <code>{'A': 1, 'Z': -1}</code> <code>dict(zip(['A', 'Z'], [1, -1]))</code> <code>dict([('A', 1), ('Z', -1)])</code> <code>dict({'Z': -1, 'A': 1})</code>

Notable operations: `len`, `del`, `in`

Notable methods: `clear()`, `keys()`, `values()`, `items()`, `popitem()`, `pop(key)`, `update({key, value})`, `update(key=value)`, `get(key)`, `setdefault`

Sets

<code>set()</code>	empty set
<code>value in set</code>	test for presence

Mutable, for immutable, use `frozenset`.

Notable methods: `add(x)`, `remove(x)`

Conditions

```
if condition:
    stuff

else:
    stuff

if condition:
    stuff

elif condition2:
    stuff

elif condition3:
    stuff

else:
    stuff

ternary = a if condition else b
```

Iteration

```
for i in range(start, stop, step):
    stuff
for value in [sequence]:
    stuff
for position, value in enumerate(sequence):
    stuff
for a, b in zip(first, second):
    stuff
for ###
else
    stuff to do at end of loop (usually
exception when breaking in loop)
while condition:
    stuff
else
    stuff to do when condition is false
break # breaks the loop
continue # continue at the start of the loop body
```

module `itertools` provides lots of interesting tools for iteration

Exceptions

```
try:
    # do something
except ExceptionName as e:
    # do something
except (ExceptionName, OtherException) as e:
    # do something
else:
    # do something when no exception
finally:
    # do something anyway, exception or not
```

Functions

```
def function_name(args):
    body
```

`nonlocal variable_name` access to nearest enclosing scope for variable

`global variable_name` access global scope variable

`def func(a, b, c):` standard positional args

`func(c=1, b=2, a=3)` call using keywords

Functions (cont)

`def func(a, b=55, c=85):` default values for missing args, !! be wary of mutable defaults

`def func(*n):` variable arg list, addressed as an array

`func(*n)` calls `func` unpacking array argument

`def func(**kwargs):` variable keyword arguments, addressed as a dict

`func(**n)` calls `func` unpacking dict argument

`pass` empty body method

`return a, b, c` return multiple values

`lambda [parameters]: expression` for small snippets

```
def func(n):
    """documentation, possibly multiline"""
    pass
```

Special attributes: `__doc__`, `__name__`, `__qualname__`, `__module__`, `__defaults__`, `__code__`, `__globals__`, `__dict__`, `__closure__`, `__annotations__`, `__kwdefaults__`

builtins: `abs` `all` `any` `ascii` `bin` `callable` `chr` `compile` `delattr` `dir` `divmod` `eval` `exec` `format` `getattr` `globals` `hasattr` `hash` `hex` `id` `input` `isinstance` `issubclass` `iter` `len` `locals` `max` `min` `next` `oct` `open` `ord` `pow` `print` `repr` `round` `setattr` `sorted` `sum` `vars`

Utilities

```
map(function, iterable, optional_iterable, ...)
```

```
zip(*iterables)
```

```
filter(function, iterable)
```



By **Kalamar**
cheatography.com/kalamar/

Published 14th February, 2018.
Last updated 14th February, 2018.
Page 2 of 5.

Sponsored by **Readable.com**
Measure your website readability!
<https://readable.com>

Comprehensions

list [expression for iterable clause if optional clause]

nested lists [expression for iterable clause_1 for iterable clause_2 if optional clause filter]

dict { key_expression: value_expression for iterable clause if optional clause filter }

set { value_expression for iterable clause if optional clause filter }

Generator functions

yield x returns the value x, suspends and stores the internal state

next(function) resumes execution (or start the first time)

until loop ends with no value or StopIteration is raised

generator.send(x) resumes and makes x as the return value of yield

usage example: for in in generator_function(**some_params)

yield from list_comprehension advanced pattern

Generator expression

like a list comprehension but () instead of []

returns 1 item at a time

easier to read than map+filter

Decorators

```
def wrap(func):
    def wrapper(*args, **kwargs):
        # do something about func
        func(*args, **kwargs)
        # do something about func
    return wrapper

# Apply decorator
def to_decorator(...):
    # body
    to_decorator = wrap(to_decorator)
    # More idiomatic
    @wrap
    def to_decorator(...):
```

Decorators (cont)

```
>#body
from functools import wraps
@wraps(func)
def wrapper(...) # to keep the name and doc from the wrapped function
    # Decorator with args: make a decorator factory
def decorator_factory(factory_args):
    def decorator(func):
        def wrapper(*args, **kwargs):
            # do something about func
            func(*args, **kwargs)
            # do something about func
        return wrapper
    return decorator
@decorator_factory(1,2,...)
def to_decorate(...):
```

Multiple decorators: the closer the decorator to the function, the sooner it is applied.

Classes

class Child(Parent): is-a inheritance

class Child(Multiple, Parent): Child.mro() returns Method Resolution Order

def __init__(self, args): initializer

super().__init__(self, some_args) call parent initializer

def instance_method(self, args): self is a convention for the first argument, the implicit instance

instance = ClassName(params) create instance

@staticmethod annotation for static method, no special argument (like self)

Classes (cont)

`@classmethod` annotation for static method, special argument is the class object itself (convention: `cls`). Usage: factory methods or extracting sub-methods from `@staticmethod` methods (to avoid having a hard-coded class name when calling them)

`_attribute` pseudo private attribute (convention)

`__some_attr_or_method` almost private through name mangling

`@property` for getter

`@property.setter` for setter

a class is a subclass of itself; attributes can be added in declaration (class attributes) or dynamically to instance or class

Custom Iteration

iterable must define `__iter__` or `__getitem__`

iterator must define `__iter__` (returning self) and `__next__` (raise `StopIteration` at the end)

usage: `for i in iterable:`



By **Kalamar**
cheatography.com/kalamar/

Published 14th February, 2018.
 Last updated 14th February, 2018.
 Page 4 of 5.

Sponsored by **Readable.com**
 Measure your website readability!
<https://readable.com>