- NOTE: just go through built-in docs at top all modules can be reviewed as necessary

# Python Docs

## Built-in Types

### Truth Value Testing

- any object in if or loop conditional
- falsey types:
  - None
  - False
  - 0, 0L, 0.0, 0j
  - empty sequence `'', (), []`
  - empty mapping {}
  - user-defined class with `__nonzero__` or `__len__` that return integer 0 or false

### Boolean Operations - and, or, not (in ascending priority)

- or
- and
- not

### Comparisons

- <, <=, >, >=, ==, !=, is, is not
- can override `__lt__` etc, but can also override `__cmp__`

### Numeric Types - int, float, long, complex

- booleans are a subtype of plain integers
- ints are always C longs (sys.maxint)
- floats are usually doubles (sys.float_info)
- complex - z.imag, z.real
- fractions - hold rationals
- decimal - floats with user-defined precision
- can use ctors float(), int(), long(), complex() etc
- general numeric operators with
  - .conjugate()

    - divmod()
    - pow() or `**`
- all numbers.Real types (int, long, and float) also include
    - math.trunc(x)
    - round(x, n)
    - math.floor(x)
    - math.ceil(x)

## Bitwise Operations on Integer Types

## Additional Methods on Integer Types

- int.bit_length()
- long.bit_length()

## Additional Methods on Float Types

- float.as_integer_ratio() - returns pair
- float.is_integer()
- float.hex()
- float.fromhex()

## Iterator Types

- `container.__iter__()`
- `iterator.__iter__()` - return self
- `iterator.next()` - raises StopIteration on end

## Sequence Types - str, unicode, list, tuple, bytearray, buffer, xrange

- string literals - 'fdsa' or "fksdla"
- unicode strings - u'sjkdf' or u"dfsjakl"
- lists - `[0, 1, 2]`
- tuple - elem1, elem2, elem3 or (elem1, elem2, elem3) always () for an empty tuple and (elem1,) for a single item
- bytearray - built-in bytearray() function - mutable sequence of integers from [0, 256)
- buffer - not directly supported by Python syntax
- xrange - similar to buffer
- supports
    - x in s
    - x not in s
    - s + t - concatenation

    - s * n - add s to itself n times
    - `s[i]` - 0 indexed ith item
    - `s[i:j]` - slice
    - `s[i:j:k]` - slice with step
    - len(s)
    - min(s)
    - max(s)
    - s.index(x) - index of first occurrence
    - s.count(x) - number of x in s

## String Methods

- .capitalize()
- .center(width, fill character)
- .count(substring, start, end) - num of non-overlapping occurrences
- .decode(encoding, errors)
- .encode(encoding, errors)
- .endswith(suffix, start, end)
- .expandtabs(tabsize)
- .find(sub, start, end)
- .format() - new standard for string formatting
- .index(sub, start, end) - like find but raises ValueError
- .isalnum()
- .isalpha()
- .isdigit()
- .islower()
- .isspace()
- .istitle()
- .isupper()
- .join(iterable) - concatenate elements of iterable with src string as separator
- .ljust(width, fill character)
- .lower()
- `.lstrip([chars])`
- .partition(sep)
- .replace(old, new, count)
- .rfind(sub, start, end)
- .rindex(sub, start, end)
- .rjust(width, fill character)
- .rpartition(sep)

- .rsplit(sep, maxsplit)
- .rstrip([chars])
- .split(sep, maxsplit)
- .splitlines(sep, maxsplit)
- `.splitlines([keepends])` - keeps
- .startswith(prefix, start, end)
- .strip([chars])
- .swapcase()
- .title() - title cased
- .translate(table, delete characters
- .upper()
- .zfill(width) - filled with 0's
- unicode.isnumeric()
- unicode.isdecimal()

## String Formatting Operations

## XRange Type

## Mutable Sequence Types

## Set Types - set, frozenset

## Mapping Types - dict

## File Objects

## memoryview type

## Context Manager Types

## Other Built-in Types

- Modules
- Classes and Class instances
- Functions
- Methods
- Code Objects
- Type Objects
- The Null Object
- The Ellipsis Object

- The NotImplemented Object
- Boolean values
    - bool()
- Internal Objects

#### Special Attributes

- `__dict__`
- `__methods__`
- `__members__`
- `__class__`
- `__bases__`
- `__name__`
- attributes for new-style classes (inherits from object, required for `__slots__` and `__getattribute__` )
    - `__mro__`
    - .mro()
    - `__subclasses__`

# Note review the built-in functions, specifically chr

# HOWTOs

# Descriptors

- an object attribute with "binding behavior" -> one whose attribute access has been overridden by methods in the descriptor protocol
- `__get__()` , `__set__()` , `__delete__()`
- if any are defined for an object, it is a descriptor
- default behavior for attribute access is to get, set, or delete the attribute from an object's dictionary
- a.x lookup chain -> `a.__dict__['x']` -> `type(a).__dict__['x']` -> then base classes of type(a) excluding metaclasses
- mechanism behind properties, methods, static methods, class methods, and super()
- simplify underlying C-code

## Descriptor Protocol

```
descr.__get__(self, obj, type=None) --> value
descr.__set__(self, obj, value) --> None
descr.__delete__(self, obj) --> None
```

- overriding any of these makes an object a descriptor
- get & set -> data descriptor
- get -> non-data descriptors
- get, set causes AttributeError -> read-only data descriptor

## Invocation

- different for obj vs. class (object is an instance class is the type)
- object (using `object.__getattribute__()` ):

```
b.x -> type(b).__dict__['x'].__get__(b, type(b))
```

```
- priority:
    - data descriptors
    - instance variables
    - non-data descriptors
    - `__getattr__()` if provided
- see PyObject_GenericGetAttr() in Objects/object.c
```

- class (using `type.__getattribute__()` ):

```
B.x -> B.__dict__['x'].__get__(None, B)
```

- descriptors are invoked by the `__getattribute__()` method
- overriding `__getattribute__()` prevents automatic descriptor calls
- `object.__getattribute__()` and `type.__getattribute__()` make different calls to **get** ().
- data descriptors always override instance dictionaries.
- non-data descriptors may be overridden by instance dictionaries
- calls to super return objects with custom `__getattribute__()` for invoking descriptors

## Properties

- can call property() to build a data descriptor that triggers function calls upon access to an attribute

```
property(fget=None, fset=None, fdel=None, doc=None) -> property attribute
```

# Functional Programming

## Iterators

- built on iterators
- must implement `__next__()` and throw StopIteration when called after end
- built-in iter() tries to return an iterator from an object

## Generator expressions and list comprehensions

```
line_list = ['  line 1\n', 'line 2  \n', ...]

# Generator expression -- returns iterator
stripped_iter = (line.strip() for line in line_list)

# List comprehension -- returns list
stripped_list = [line.strip() for line in line_list]
```

- generator expressions are always in parentheses but can be inside function call parens

```
obj_total = sum(obj.count for obj in list_all_objects())
```

## Generators

```
def generate_ints(N):
    for i in range(N):
        yield i
```

- just requires the yield keyword, which is detected by Python's bytecode compiler and compiles the function differently as a result
- returns generator object that supports iterator protocol

## Built-in functions (using iterators)

- map
- filter
- enumerate
- sorted
- any
- all
- zip

## The itertools module

- functions that create a new iterator based on an existing iterator
- functions for treating an iterator’s elements as function arguments
- functions for selecting portions of an iterator’s output
- a function for grouping an iterator’s output
- count
- cycle
- repeat
- chain
- islice
- tee
- starmap
- filterfalse
- dropwhile
- compress
- combinations
- permutations
- combinations_with_replacement
- groupby

## Calling functions on elements

- operator module ->
  - implements operators as functions

## The functools module

- partial -> binds an existing function to arguments
- reduce

## Small functions and the lambda expression

```
adder = lambda x, y: x + y
```

- creates an anonymous function that returns the value of the expression

# MRO