1 Intro

Useful resource: `https://tenthousandmeters.com/tag/python-behind-the-scenes/`

this document is about learning python. The following is the hello world program:

```
print("Hello World");
```

2 Variables

Each variable is **connected** to a value.

Uppercase letters in variable names have special meaning (later)

Internally, variables are **references** to values in memory.

2.0.1 Strings

You can use both " and ' to delimit them.

Concat them with + to write them over multiple lines, and write
n to write newline.

The internal representation of strings in python is actually not that simple.

The string has 2 possible states: **compact** and **legacy**, in which compact representation basically is a list of UTF-8 characters and is used only *maximum character and size are known at creation time* (eg for string literals).

Otherwise, it will revert to the legacy representation, which, depending on the content of the string, can be of 3 *kinds*

- Latin-1
- UCS-2
- UCS-4

Reported here is the actual struct used in CPython as of PEP393

```c
typedef struct {
  PyObject_HEAD
  Py_ssize_t length;
  Py_hash_t hash;
  struct {
      unsigned int interned:2;
      unsigned int kind:2;
      unsigned int compact:1;
      unsigned int ascii:1;
      unsigned int ready:1;
  } state;
  wchar_t *wstr;
} PyASCIIObject;

typedef struct {
  PyASCIIObject _base;
  Py_ssize_t utf8_length;
  char *utf8;
  Py_ssize_t wstr_length;
} PyCompactUnicodeObject;

typedef struct {
  PyCompactUnicodeObject _base;
  union {
      void *any;
      Py_UCS1 *latin1;
      Py_UCS2 *ucs2;
      Py_UCS4 *ucs4;
  } data;
} PyUnicodeObject;
```

link to the documentation: `https://peps.python.org/pep-0393/#string-creation`

We have methods to manipulate the string, like `strip`, `find`, (index), `split`, `join`, we can **use all comparisons operations lexicograhycal**,

We can also query for membership like

```python
'a' in 'apple' == True
```

3 Numbers

There are 3 types of number in python: **integers**, **floating-point numbers** and **complex numbers**. The standard library also gives us `decimal.Decimal` and `fractions.Fraction`.

To create a complex number, just append the 'j' to a numeric literal

```
inum = -32432
fnum = 3.32423
cnum = 3.14 - 1j
```

Integers in python are **arbitrary-precision integers**.

```
typedef struct {
    PyObject ob_base;
    Py_ssize_t ob_size; /* Number of items in variable part */
} PyVarObject;

struct _longobject {
    PyVarObject ob_base; // expansion of PyObject_VAR_HEAD macro
    digit ob_digit[1];
};
```

the `ob_digit` member is a pointer to an array of digits. More information on this bignum arithmetic implementation `https://tenthousandmeters.com/blog/python-behind-the-scenes-8-how-python-integers-work/`

This comes with performance implications for each integer operation and the memory consumption of each integer, which is proportional to the number itself. For reference **small numbers take 28 bytes**. You can verify that by calling the `bit_length` method on an integer

- a reference count `ob_refcnt`: 8 bytes
- a type `ob_type`: 8 bytes
- an object's size `ob_size`: 8 bytes
- `ob_digit`: 4 bytes.

Floating numbers are instead double precision floatin point numbers, stored in a `PyObject` type, which is a reference counted object.

```
typedef struct {
    PyObject_HEAD
    double ob_fval;
} PyFloatObject
```

Complex numbers are basically a pair of floating point numbers (double precision)

```
typedef struct {
    PyObject_HEAD
    double cval_real;    // Real part
    double cval_imag;    // Imaginary part
} PyComplexObject;
```

For each of the number types the the following operations are defined Furthermore, integers also feature **bitwise operations**, which are `|` (or),^ (xor), `&` (and), `<<` (left shift), `>>` (right shift), (not)

Table 1: Built-in python numbers operations

`Operation`	`Description`
`x + y`	Sum of x and y
`x - y`	Difference of x and y
`x * y`	Product of x and y
`x / y`	Quotient of x and y
`x // y`	Floored quotient of x and y
`x % y`	Remainder of x/y
`-x`	x negated
`+x`	x unchanged
`abs(x)`	Absolute value or magnitude of x
`int(x)`	x converted to integer
`float(x)`	x converted to floating point
`complex(re, im)`	A complex number with real part re, imaginary part im (defaults to zero)
`c.conjugate()`	Conjugate of the complex number c
`divmod(x, y)`	The pair $(x/y, x\%y)$
`pow(x, y)`	x to the power y
`x ** y`	x to the power y

4 Boolean

The `bool` type has 2 possible values: `True` and `False`, with a constructor

5 Containers Intro

Python supports 3 types of containers: **sequences** and **set types** and **mapping objects**, each with its interface, which is meant to be implemented if you want to write your custom container.

A method which is shared by all of them is the one that returns an interator over the container (part of the **iterable** interface)

```
container.__iter__()
```

While an **iterator** implmenets

```
iterator.__iter__()
iterator.__next__() # raise StopIteration exception at end
```

Example of a first custom container (with immutable elements, hence useless, for the sake of using iterators)

```
def ordinalStr(num):
  match num:
    case 1:
      return '1st'
```

```
    case 2:
      return '2nd'
    case 3:
      return '3rd'
    case _:
      return str(num) + 'th'

# trying out iterators for myself
class Thing:
  # private method starts with __

  # private variable
  _numbers = [ordinalStr(x) for x in range(1, 10)]

  # constructor
  def __init__(self, x):
    # another variable
    self._firstStr = self._numbers[0]
    self._arg = x

  # iterable interface
  def __iter__(self):
    return self._numbers.__iter__()

  # to print it directly
  def __repr__(self):
    className = type(self).__name__
    return f'{className}()'

  def getArg(self):
    return self._arg;
```

Starting to see **sequences**

6 Sequence Types

There are 3: **`list`**, **`tuple`**, **`range`**

All sequences have common operations, eg. membership, indexing, concatenation, repetition, minmax, count.

Of course, these operators work if you defined the interface, in particular in min/max, you need to define the **comparison** operations (`__lt__`, `__le__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`) If you give a **negative index** `i`, then the actual formula used to index the object is `len(s) + i`

Note that `str` objects are immutable (like strings in java) and the equivalent of string builder is `io.StringIO` (or use `str.join()` on an *(*iterable))

Table 2: Sequence operations

Operation	Outcome
`x in s`	`True` if an item of s is equal to x, else `False`
`x not in x`	`False` if an item of s is equal to x, else `True`
`s + t`	the concatenation of s and t
`s * n` or `n * s`	equivalent to adding s to itself n times **shallow copy**
`s[i]`	ith element of s
`s[i:j]`	slice of s from i to j
`s[i:j:k]`	slice of s from i to j with step
`len(s)`	length of s
`min(s)`	smallest element of s
`max(s)`	largest element of s
`s.index(x[, i[, j]])`	find first occurrence of x, optionally within a subsequence from i to j
`s.count(x)`	number of occurrences of x in s

Table 3: Mutable Sequence Operations

Operation	Outcome
`s[i] = x`	item of `s` at `i` replaced by `x`
`s[i:j] = t`	slice of s from i to j replaced by iterable t
`del s[i:j]`	same as `s[i:j] = []`
`s[i:j:k] = t`	elements of `s[i:j:k]` are replaced by iterable t
`s.append(x)`	appends `x` at the end of the sequence, same as `s[len(s):len(s)] = [x]`
`s.clear()`	same as `del s[:]`
`s.copy()`	creates a shallow copy of `s`, same as `s[:]`
`s.extend(t)`	extends `s` with the contents of `t`, similiar, but **not** equal to `s[len(s):len(s)] = t`
`s += t`	same as `s.extend(t)`

`tuple` and `range` are immutable, hence **hashable**, which means that they can be used as key types in an mapping object like `dict`.

Mutable sequence types like `list` support additional operations, i.e. assignment, deletion, insertion, extension.