

intro to Python

part 0: setup

installing Python

- Windows:
 1. download Cygwin: <https://www.cygwin.com>
 2. install it and necessary Python 3 package(s).
- Mac:
 1. download for Mac: <https://www.python.org/downloads>
 2. install it.
- Linux (Debian): run `apt install python3` if not installed.
- check Python 3 is installed:
 1. run `which python3` in a shell.
 2. check that it prints the path to `python3` executable.

interactive Python

```
$ python3 # launch Python interpreter from shell.

>>> s = "Hello world!" # define a string object.
>>> pr<tab><tab> # list candidates if there are multiple.
print(      property(
>>> pri<tab> # tab completion if there is just 1 candidate.
>>> print(s)
Hello world!
>>> help # show help.
Type help() for interactive help, or help(object) for help
about object.
>>> ^R pr<enter> # press Ctrl+R to search past lines.
>>> print(s)
Hello world!
>>> ^D # press Ctrl+D to exit interactive interpreter.
```

part 1: basics

what is Python?

Python is a:

the programmer need not worry much about the target device's details.

- high-level

usable in various applications, not a field-specific language.

- general-purpose

- programming language.

cf: human language, markup language, data serialisation language, hardware description language.

key features

- most often interpreted, but may be compiled
- dynamic but strong typing
- multi-paradigm
- emphasis on readability
- easy to get started
- expandable with use of libraries

help and references

- inside interpreter: `help( . . . )`
- online reference: <https://docs.python.org/3/>

example expressions

inside an interactive interpreter.

note: `>>>` is just a token that appears in Python's interactive interpreter and is not part of the Python language.

```
>>> print("Hello world!") # print a string literal.  
Hello world!  
>>>  
>>> a = 42 # a has value 42 (integer).  
>>> b = -7 # b has value -7 (integer).  
>>> c = (a + b)/2 # c has value 17.5 (float).  
>>> c # return the result.  
17.5
```

objects

object: an abstraction of data stored in memory:

- identity (`id`): the object's name (eg: `foo`)
- type (`type`): how the object behaves (eg: `integer`)
- value (eg: `42`)

```
>>> foo = 42      # foo refers to an integer of value 42.
>>> type(foo)
<class 'int'>
>>> foo = True    # foo now refers to a boolean of value True,
>>> type(foo)     # the old object (integer 42) is gone.
<class 'bool'>
```

naming rules

- case sensitive: "A" and "a" are different names.
- must be at least 1 character.
- may use characters [_A-Za-z0-9].
- must not begin with numeral [0-9].
- refrain from creating identifiers beginning with "_".
- should sufficiently describe its purpose.
- class/type/exception names in "CamelCase".
- variable/function/module names in "lower_case".

```
a = 42          # bad name. what is its purpose??  
num_iterations = 42 # a better name.  
3_loops = True  # illegal name beginning with numeral.
```

reserved words

- `def`, `if`, `is`, `return`, ... have special meaning in Python.
- cannot be used as a name of any object.

```
def same_id(a, b): # def: function definition.
    if a is b:      # if: conditional. is: identity testing.
        return True # return: exit function with a value.
    else:           # else: execute if prior conditions
        return False # are false. True and False are also
                    # reserved words.

numbers = [2**i for i in range(7)] # for: iterative loop,
while len(numbers) > 0:           # in: membership test.
    num = numbers.pop()           # while: conditional loop.
    if num % 2 == 1:
        pass # pass: end execution of current loop iteration.
```

types and literals

- type: determines how an object shall be interpreted.
- literal: an explicitly expressed value.

TYPE	EXAMPLE LITERALS			
none	None			
boolean	True	False		
integer	42	0x2a	0o377	0b10111010
float/real	42.	17.5	1e100	
complex	1+1j	42-7j	0j	
string	"Hello world!"			
list	[]	[42, -7]		

lines and whitespace

- mandatory indentation for function/class definitions, loops, conditionals, ... (recommendation: use tabs)
- explicit line continuation using \ at end of line.

```
#!/usr/bin/python3

    # blank lines.

a = 42 # statement.
b = -7 # another statement.
c = (a + b) \
    / 2 # explicit line
        # continuation "\".
```

```
d = [42, -7,
     33, 0] # implicit line
            # continuation
            # because of
            # brackets.

def return_true():
    if True: # mandatory
        return True # indenting.
```

operators and assignments

- operator: **returns** a value from 1 or more operands.
- assignment: binds a name to an object, **no return**.

```
>>> a = 42 # bind 42 to a.
>>> a # return 42.
42
>>> -a # unary negation.
-42
>>> b = -7
>>> a + b # add operator.
35
>>> c = a
>>> c += b # add b to c.
```

```
>>> d = 0b10010101
>>> e = 0b10101100
>>> d ^ e # xor operator.
57
>>> bin(d ^ e) # as base 2.
'0b111001'
>>> f = 0b11001100
>>> f >>= 3 # shift f right
>>> bin(f) # by 3 bits.
'0b11001'
```

list of operators

- **green**: return value is boolean.
- **yellow**: return value is one of the operands, not boolean, in a way that satisfies truthiness.
- operator meanings described below usually assume a and b are numeric (eg: "+" is described as "addition").
- however, other types may use these operator symbols (eg: "a" + "b" returns "ab", [42] + [-7] returns [42, -7]).

unary operators:

+	a	no-op	~ a	bitwise NOT
-	a	negation	not a	logical NOT

ternary operators:

a	if	c	else	b	conditional return
---	----	---	------	---	--------------------

binary operators:

a + b	addition	a << n	left bit shift	a == b	equal to
a - b	subtraction	a >> n	right bit shift	a != b	not equal to
a * b	multiplication	a & b	bitwise AND	a < b	less than
a ** b	exponentiation	a b	bitwise OR	a > b	greater than
a / b	true division	a ^ b	bitwise XOR	a <= b	less than or equal
a // b	floored division	a and b	logical AND	a >= b	greater than or equal
a % b	modulus	a or b	logical OR		
a in c	positive membership	a is b	positive object identity		
a not in c	negative membership	a is not b	negative object identity		

list of assignment delimiters

<code>a = b</code>	assign b to a	<code>a **= b</code>	raise a to the power of b
<code>a += b</code>	add b to a	<code>a &= b</code>	AND a with b
<code>a -= b</code>	subtract b from a	<code>a = b</code>	OR a with b
<code>a *= b</code>	multiple a by b	<code>a ^= b</code>	XOR a with b
<code>a /= b</code>	true divide a by b	<code>a <<= n</code>	shift bits of a left by n bits
<code>a //= b</code>	floor divide a by b	<code>a >>= n</code>	shift bits of a right by n bits
<code>a %= b</code>	take a modulo b	<code>a := b</code>	assign b to a and return its value

conditional execution (1)

- often we need to execute code conditionally.
- this is achieved using keywords `if`, `elif`, `else`.
- conditions A and B may be almost any expression (suggestion: make them return explicit `True` or `False`).
- `else` is often used to handle invalid cases.

```
if A:
    foo() # executes if A is True.
elif B:
    bar() # executes if A is False and B is True.
else:
    qux() # executes if A is False and B is False.
baz()    # always executes after foo(), bar() or baz().
```

conditional execution (2)

- **note: requires Python 3.10 or later.**
- code can also be conditionally executed based on multiple cases, instead of just binary **True** and **False**.
- **default** is often used to handle invalid cases.

```
match value:
    case A:
        foo() # executes if value matches A.
    case B:
        bar() # executes if value matches B.
    default:
        qux() # executes if value matches none of the above.
```

repeated execution

- often we need to execute code repeatedly.
- `for A in B` : execute loop for each element in B.
- `while c`: execute loop while C evaluates to true.
- iterated code must be indented after `for`, `while`.

```
for n1 in range(8): # n1 takes values from 0 to 7.
    if n1 > 6:
        break # ends the loop when n1 is 7.
    print(n1)

n2 = 0
while n2 <= 6: # each iteration is executed if this
    print(n2)  # condition evaluates to true.
    n2 += 1    # care is needed to avoid infinite loops.
```

importing libraries

- often we need to use external libraries.
- importing is done using keyword `import`.

```
>>> import math # import all of math into its own namespace.
>>> math.log(2.)
0.6931471805599453
>>> from math import log, sin, pi # import specific objects
>>> log(2.)                        # into global namespace.
0.6931471805599453
>>> from math import tan as Tan # specify the alias to
>>> Tan(pi)                      # import tan under.
-1.2246467991473532e-16
>>> import math as m # import math under alias m.
>>> m.pi
```

part 2: data types

none

- represents an absence.
- useful for missing data, arguments, return value, etc.
- only 1 possible value: `None`

```
>>> def no_return():  
...     x = 42  
...     # no return value.  
...  
>>> None # does nothing.  
>>> print(no_return())  
None
```

```
>>> def foo(x, lst=None):  
...     # lst=[] introduces a  
...     # bug, hence the None.  
...     if lst is None:  
...         lst = []  
...     lst.append(x)  
...     return lst  
...  
>>> (foo(42), foo(-7))  
([42], [-7])  
>>> import re  
>>> match = \  
...     re.search("a", "foo")  
>>> if match is None:  
...     print("no match found")  
...  
no match found
```

boolean

- true and false values, used in logical operations.
- 2 possible values: `True` and `False`
- see "truthiness" for how other types behave logically.

```
>>> bool(None)
False
>>> bool([0]) # non-empty.
True
>>> False or 42 # non-bool.
42
>>> bool(False or 42)
True
>>> bool(0.0 and 42)
False
```

```
>>> not 0.0 # returns bool.
True
>>> not False or 42
True
>>> not (False or 42)
False
>>> if False or 42:
...     print("42 is true")
...
42 is true
```

numeric

- integer: arbitrarily-sized integers (eg: 42, -7).
- floating point: 64-bit real numbers (eg: 42., -7e33).
- complex: a pair of floats (eg: 42-7j, 1e100j).

```
>>> # big integers.  
>>> 2 ** 76  
75557863725914323419136  
>>> # 64-bit floats.  
>>> 2. ** 76  
7.555786372591432e+22  
>>> # complex numbers.  
>>> import cmath  
>>> cmath.exp(math.pi * 1j)  
(-1+1.2246467991473532e-16j)
```

```
>>> # division.  
>>> 42 / -7 # float.  
-6.0  
>>> 42 // -7 # integer.  
-6  
>>> # mixed-type compares.  
>>> 42 == 42.  
True  
>>> True == 1+0j  
True
```

beyond none, boolean and numeric

- **range**: iteratively outputs integers of a sequence.
- **string**: immutable text sequence of Unicode characters.
- **bytes**: immutable binary data sequence of bytes.
- **tuple**: immutable sequence of (often heterogeneous) elements.
- **list**: mutable ordered sequence of (often homogeneous) elements.
- **set**: mutable unordered elements that obey set theory.
- **dictionary**: mutable unordered key--value pairs (associative array).

```
>>> newtuple = (42, -7)
>>> set(range(8)) # form a set using a range.
{0, 1, 2, 3, 4, 5, 6, 7}
>>> newlist = [42, -7] + [33, 1] # list concatenation.
>>> newlist[1:3] # sublist from 1 inclusive to 3 exclusive.
[-7, 33]
```

ranges

- an object that generates a sequence of integers.
- commonly used in `for` loops, eg: `for i in range(42).`

```
# from 0 (inclusive) to
#   stop (exclusive).
range(stop)

# from start (inclusive)
#   to stop (exclusive).
range(start, stop)

# from start to stop in
#   increments of step.
range(start, stop, step)
```

```
>>> list(range(8))
[0, 1, 2, 3, 4, 5, 6, 7]
>>>
>>> list(range(3, 8))
[3, 4, 5, 6, 7]
>>>
>>> list(range(3, 8, 2))
[3, 5, 7]
>>>
>>> list(range(8, 3, -1))
[8, 7, 6, 5, 4]
```

strings

- immutable ordered sequence of Unicode characters.
- commonly used to process text.
- Python 3 natively supports UTF-8 encoded text.

```
>>> s = " あいうえお "
>>> s[1:] # indexable.
' いうえお '
>>> len(s)
5
>>>
>>> for c in s:
...     # iterable.
...     print(c)
```

```
>>> s = "never give up"
>>> s.capitalize() + \
...     " {}!!".format("Jane")
'Never give up Jane!!'
>>> s.split(" ")
['never', 'give', 'up']
>>> print(s.upper()) # to stdout.
NEVER GIVE UP
>>> s.index("g")
6
```

bytes

- immutable ordered sequence of raw bytes.
- commonly used to process binary data.
- byte literals: ASCII characters (eg: `b"foo"`), escaped characters (eg: `b"\0\1\n\"`), hexadecimal (eg: `b"\xff"`).

```
>>> s = b"\xe3\x81\x82"
>>> s[1:] # indexable.
b'\x81\x82'
>>> len(s)
3
>>> s.decode() # UTF-8
' あ '
```

```
>>> for b in b"\x2a\x21":
...     # iterable, prints unsigned
...     # 8-bit integer value of
...     # each byte.
...     print(b)
42
33
```

tuples

- immutable ordered collection of possibly repeated elements of any type (including other collections).
- often used to make a function return multiple values.
- a tuple is hashable iff all its elements are hashable.

```
>>> a = ()          # an empty tuple.
>>> b = (42,)       # a single-element tuple.
>>> c = tuple(range(8)) # explicit tuple creation.
>>> (x, y, z) = cartesian(lat, lon, radius)
>>> (42, None, "foo")[1:] # sub-tuple via indexing.
(None, 'foo')
>>> (42, None, "foo")[2] # indexing for a single element.
'foo'
```

lists

- mutable ordered collection of possibly repeated elements of any type.
- common operations: append, pop, sort, find, ...

```
>>> a = [6] + [7]           # concatenation.
>>> b = list(range(8))      # explicit list creation.
>>> b.append(42)            # append 42 onto the end of b.
>>> b.extend(a)             # concatenate a onto the end of b.
>>> b.pop()                 # pop b (return the last element).
7
>>> b.pop()                 # pop b again.
6
>>> len(b)                  # length of b: [0, 1, ..., 7, 42].
9
```

sets

- mutable unordered de-duplicated collection of hashable objects (none, string, number, tuple, ...).

```
>>> a = {6, 7, 7}           # a new set (7 is de-duplicated).
>>> a.add(4)                # add 4 to a.
>>> b = set(range(8))       # explicit set creation.
>>> b.difference(a)         # return set difference.
{0, 1, 2, 3, 5}
>>> a.intersection(b)      # return set intersection.
{4, 6, 7}
>>> a.union(b)              # return set union.
{0, 1, 2, 3, 4, 5, 6, 7}
>>> a.issubset(b)           # a is a subset of b.
True
```

dictionaries

- mutable unordered collection of key--value pairs.
- the keys form a set (eg: number, string, tuple), but each value may be of any type.

```
>>> a = {"Jane": 42, "Bob": -7}
>>> a["Jane"] # indexing is done by keys, not integers.
42
>>> a["Craig"] = 33 # add "Craig".
>>> len(a)
3
>>> b = {"Azusa": 1, "Yu": 10, "Jane": 444}
>>> a.update(b) # add "Azusa" and "Yu", modify "Jane".
>>> a
{'Jane': 444, 'Bob': -7, 'Craig': 33, 'Azusa': 1, 'Yu': 10}
```

list/set/dict comprehension

- to quickly create a collection of objects, use "comprehension".
- this is achieved using a "for loop"-like syntax.
- elements may be conditionally included using `if`.

```
>>> {2**i for i in range(8)} # set (note there is no ':').
{32, 1, 2, 64, 4, 128, 8, 16} # the set may print unsorted
>>>                                     # because it is.
>>> [2**i for i in range(8) if i % 2 == 0] # list.
[1, 4, 16, 64] # odd powers of 2 are excluded.
>>>
>>> s = "abcd"; {k: v for (k, v) in zip(s, range(len(s)))}
{'a': 0, 'b': 1, 'c': 2, 'd': 3} # dict (note the ':').
```

truthiness

- expressions that expect boolean (true/false) input (eg: `if`, `while`, `and`, `or`, `not`) test input value "truthiness".
- can be confusing: is `[None]` "true" or "false"?
- to check an object's truthiness, use `bool(...)`.

TYPE	VALUE	RESULT	EXAMPLES
none	None	False	
bool/numeric	zero	False	False 0 0.000 0+0j
bool/numeric	non-zero	True	True 42 1e-100 1j
container	empty	False	" " range(0) () []
container	non-empty	True	"foo" range(42) (None,) [42]

mutability

- whether or not an object's value can be modified.
 - mutable: set, list, dictionary, ...
 - immutable: none, boolean, number, string, tuple, ...
- a tuple (immutable) may contain mutable elements.

```
>>> # example 1: list.
>>> a = [42, -7]
>>> a.append(33) # a gains
>>> a           # new 33.
[42, -7, 33]
>>> a[0] = 1 # 0th element
>>> a       # replaced.
[1, -7, 33]
```

```
>>> # example 2: tuple.
>>> b = (42, -7)
>>> b[0] = 1
(error)
>>> id(b)
132716370646592
>>> b = (1, -7) # b is new
>>> id(b)      # object.
132716370645888
```

hashability

- a hash is a reproducible integer value computed from a hashable object (none, boolean, number, ...).
- it may be unstable between sessions or interpreters.
- some collections require hashable elements.

```
>>> hash(None)
591726
>>> hash(0)
0
>>> hash(1)
1
>>> hash("foo")
829405355636723524
```

```
>>> hash({42, -7}) # can't hash sets.
TypeError: unhashable type: 'set'
>>> hash((42, -7)) # hashable tuple.
3483934736066029577
>>> {(42, -7): "foo"}
{(42, -7): 'foo'} # tuple as a key.
>>> {(42, [-7]): "foo"}
TypeError: unhashable type: 'list'
```

containment

- some objects contain other objects:
 - string/bytes: sub-sequences (eg: `"f" in "foo"`).
 - range/tuple/list/set: elements (eg: `1 in range(42)`).

```
>>> # range.  
>>> 1 in range(42)  
True  
>>> 42 in range(42)  
False  
>>> # string.  
>>> "fo" in "foo"  
True  
>>> "fob" in "foo"  
False
```

```
>>> # set.  
>>> "fo" in {"foo", "bar"}  
False  
>>> "foo" in {"foo", "bar"}  
True  
>>> # dictionary keys.  
>>> 42 in {"foo": 42}  
False  
>>> "foo" in {"foo": 42}  
True
```

basic summary of built-in types

TYPE NAME	PROPERTIES				EXAMPLE LITERALS	
	MUTABLE	ITERABLE	COMPARE	CONTAINMENT	"FALSE"	"TRUE"
none	no	no	none	none	None	none
boolean	no	no	numeric	none	False	True
integer	no	no	numeric	none	0	42
float	no	no	numeric	none	0.	42.0 -7e33
complex	no	no	none	none	0j	(42-7j)
range	no	yes	none	element	range(0)	range(42)
string	no	yes	lexical	sub-sequence	" "	"foo" 'foo'
bytes	no	yes	lexical	sub-sequence	b" "	b"foo" b'foo'
tuple	no	yes	lexical	element	()	(42, -7)
list	yes	yes	lexical	element	[]	[42, -7]
set	yes	yes	inclusive	element	set()	{42, -7}
dictionary	yes	yes (key)	none	element (key)	{}	{"foo": 42}

- in this summary, all immutable types are hashable, except tuples, which are hashable only if all its elements are hashable.
- any two objects can be compared for equality (`==`, `!=`), but mixed-type objects (except numerics) compare as `False`.
- numeric types (but not complex) can be mixed with each other when comparing `<`, `<=`, `...`, but other types generally cannot.
- when boolean values are involved in comparison, `False` has numeric value 0 and `True` has numeric value 1.
- sorting: numeric (eg: `True < 1.1`). lexical (eg: `[0, 1] < [1]`). inclusive (subset/superset, eg: `{1, 2} < {1, 2, 4}`).
- dunder functions related to properties: `__id__`; `__iter__`; `__lt__`, `__gt__`, `__le__`, `__ge__`; `__contains__`

part 3: functions

overview of functions

- a block of code that is executed when it is called.
- often accepts arguments when called.
- often returns a value before it exits.
- commonly used to hide code and reduce repetition.

```
def cartesian(lat, lon, radius=R): # 3 arguments.  
    x = radius * cos(lon) * cos(lat)  
    y = radius * sin(lon) * cos(lat)  
    z = radius * sin(lat)  
    return (x, y, z) # return multiple values using a tuple.  
  
for p in points:  
    (x, y, z) = cartesian(p["lat"], p["lon"]) # many calls  
    print("{}\t{}\t{}".format(x, y, z))      # to cartesian.
```

defining and calling functions

- define a function using keyword `def`.
- return a value and stop execution using `return`.
- call a function using its identifier.
- give the function call its arguments in parentheses.

```
def cartesian(lat, lon, radius=R): # start of definition.
    x = radius * cos(lon) * cos(lat) # function body
    y = radius * sin(lon) * cos(lat) # (indented).
    z = radius * sin(lat)
    return (x, y, z) # return (function exits here).

for p in points:
    (x, y, z) = cartesian(p["lat"], p["lon"]) # 2 arguments
    print("{}\t{}\t{}".format(x, y, z)) # given here.
```

scope

- an object is only available in the region it is created.
- most defined objects have global scope.
- objects defined inside `def` or `class` = local scope.

```
>>> x = 42      # global scope.
>>> def foo():
...     x = 1    # function scope.
...     print(x)
...
>>> foo()       # prints x in foo.
1
>>> print(x)    # prints global x.
42
```

```
# name, x, foo, bar
# have class scope.
class MyClass:
    name = ""
    x = 0
    def foo(self, name):
        self.name = name
    def bar(self):
        self.x += 1
```

side effects

- subject to scope rules, functions can have effects other than return a value, ie: **side effects**.
- arguments are passed to functions by **reference**, so functions can modify outside mutable objects.
- functions can also read and write files, print, etc.

```
>>> def foo(c, x=0):  
...     c.append(x)    # append x as element to list c.  
...  
>>> x = [42, -7]  
>>> foo(x, 33)        # foo receives list x by reference and  
>>> x                 # modifies x without returning a value.  
[42, -7, 33]
```

function arguments

- functions can have various **arguments**:
 - positional (eg: a), keyword (eg: c) or either (eg: b);
 - mandatory (eg: a) or optional (eg: b=42, c=-7).
- left of /: positional-only, right of *: keyword-only.
- default values can be assigned using = (eg: b=42).

```
>>> def foo(a, /, b=42, *, c=-7): # / and * constrain args.
...     print(a, b, c)
...
>>> foo(0, 1) # a is mandatory, b set to 1 (positional).
0 1 -7
>>> foo(0, c=2) # b is default value, c set to 2 (keyword).
0 42 2
```

variadic arguments

- functions can also have **variadic** arguments (a variable number of arguments):
 - `*args` : positional arguments (tuple).
 - `**kwargs` : keyword arguments (dictionary).
- note: names `args` and `kwargs` are convention only.

```
>>> def foo(*args, **kwargs):  
...     print(args, kwargs)  
...  
>>> foo(0, 1, 2, x=3, y=4)  
(0, 1, 2) {'x': 3, 'y': 4}
```

```
>>> def bar(a, /, **kwargs):  
...     print(a, kwargs)  
...  
>>> bar(0, a=42) # allowed.  
0 {'a': 42}
```

lambda expressions

- function literals (anonymous functions) can be expressed using keyword `lambda`.

```
>>> x = lambda a, b: a + b # a function with two arguments.
>>> (lambda a, b: a + b)(42, -7) # same function as x.
35
>>>
>>> fruits = {"banana", "strawberry", "apple", "pear"}
>>> sorted(fruits, key=(lambda x: x[0])) # sort by 0th char.
['apple', 'banana', 'pear', 'strawberry']
>>> sorted(fruits, key=(lambda x: x[1])) # sort by 1st char.
['banana', 'pear', 'apple', 'strawberry']
>>> sorted(fruits, key=(lambda x: len(x))) # sort by length.
['pear', 'apple', 'banana', 'strawberry']
```