Learn Python in Y minutes

Source: learnpython

1. 1. Primitive Datatypes and Operators
2. 2. Variables and Collections
3. 3. Control Flow
4. 4. Functions
5. 5. Classes
6. 6. Modules
7. 7. Advanced

Single line comments start with a # symbol.

```
""" Multiline strings can be written
    using three "s, and are often used
    as comments
"""
```

1. Primitive Datatypes and Operators

You have numbers

```
3   # => 3
```

Math is what you would expect

```
1 + 1  # => 2
8 - 1  # => 7
10 * 2  # => 20
35 / 5  # => 7
```

Division is a bit tricky. It is integer division and floors the results automatically.

```
5 / 2  # => 2
```

To fix division we need to learn about floats.

```
2.0  # This is a float
11.0 / 4.0  # => 2.75 ahhh...much better
```

Result of integer division truncated down both for positive and negative.

```
5 // 3  # => 1
5.0 // 3.0  # => 1.0 # works on floats too
-5 // 3  # => -2
-5.0 // 3.0  # => -2.0
```

Note that we can also import division module

to carry out normal division with just one /

```
from __future__ import division

11 / 4  # => 2.75  ...normal division
11 // 4  # => 2 ...floored division
```

Modulo operation

```
7 % 3  # => 1
```

Exponentiation (x to the yth power)

```
2 ** 4  # => 16
```

Enforce precedence with parentheses

```
(1 + 3) * 2  # => 8
```

Boolean Operators: Note “and” and “or” are case-sensitive

```
True and False  # => False
False or True  # => True
```

Note using Bool operators with ints

```python
0 and 2  # => 0
-5 or 0  # => -5
0 == False  # => True
2 == True  # => False
1 == True  # => True
```

negate with not

```
not True  # => False
not False  # => True
```

Equality is ==

```
1 == 1  # => True
2 == 1  # => False
```

Inequality is !=

```
1 != 1  # => False
2 != 1  # => True
```

More comparisons

```
1 < 10   # => True
1 > 10   # => False
2 <= 2   # => True
2 >= 2   # => True
```

Comparisons can be chained!

```
1 < 2 < 3  # => True
2 < 3 < 2  # => False
```

Strings are created with " or '

```
"This is a string."
'This is also a string.'
```

Strings can be added too!

```
"Hello " + "world!"  # => "Hello world!"
```

Strings can be added without using ‘+’

```
"Hello " "world!"  # => "Hello world!"
```

. . . or multiplied

```
"Hello" * 3  # => "HelloHelloHello"
```

A string can be treated like a list of characters

```
"This is a string"[0]  # => 'T'
```

You can find the length of a string

```
len("This is a string")  # => 16
```

String formatting with %

Even though the % string operator will be deprecated on Python 3.1 and removed

later at some time, it may still be good to know how it works.

```
x = 'apple'
y = 'lemon'
z = "The items in the basket are %s and %s" % (x, y)

# z => 'The items in the basket are apple and lemon'
```

A newer way to format strings is the format method.

```
 This method is the preferred way

"{} is a {}".format("This", "placeholder")

"{0} can be {1}".format("strings", "formatted")

You can use keywords if you don't want to count.
"{name} wants to eat {food}".format(name="Bob", food="lasagna")
```

None is an object

```
None  # => None
```

Don't use the equality "==" symbol to compare objects to None

Use "is" instead

```
"etc" is None  # => False
None is None  # => True
```

The ‘is’ operator tests for object identity.

This isn’t very useful when dealing with primitive values, but is very useful when dealing with objects.

- Any object can be used in a Boolean context.

The ‘is’ operator tests for object identity.

This isn't very useful when dealing with primitive values, but is very useful when dealing with objects.

- Any object can be used in a Boolean context.
- The following values are considered `False`:

The ‘is’ operator tests for object identity.

This isn’t very useful when dealing with primitive values, but is very useful when dealing with objects.

- Any object can be used in a Boolean context.
- The following values are considered `False`:
    - None

The ‘is’ operator tests for object identity.

This isn't very useful when dealing with primitive values, but is very useful when dealing with objects.

- Any object can be used in a Boolean context.
- The following values are considered `False`:
  - None
  - zero of any numeric type (e.g., `0`, `0L`, `0.0`, `0j`)

The ‘is’ operator tests for object identity.

This isn't very useful when dealing with primitive values, but is very useful when dealing with objects.

- Any object can be used in a Boolean context.
- The following values are considered `False`:
  - None
  - zero of any numeric type (e.g., `0`, `0L`, `0.0`, `0j`)
  - empty sequences (e.g., `''`, `()`, `[]`)

The ‘is’ operator tests for object identity.

This isn’t very useful when dealing with primitive values, but is very useful when dealing with objects.

- Any object can be used in a Boolean context.
- The following values are considered `False`:
  - None
  - zero of any numeric type (e.g., `0`, `0L`, `0.0`, `0j`)
  - empty sequences (e.g., `''`, `()`, `[]`)
  - empty containers (e.g., `{}`, `set()`)

The ‘is’ operator tests for object identity.

This isn't very useful when dealing with primitive values, but is very useful when dealing with objects.

- Any object can be used in a Boolean context.
- The following values are considered `False`:
  - None
  - zero of any numeric type (e.g., `0`, `0L`, `0.0`, `0j`)
  - empty sequences (e.g., `''`, `()`, `[]`)
  - empty containers (e.g., `{}`, `set()`)
  - instances of user-defined classes meeting certain conditions see: https://docs.python.org/2/reference/datamodel.html#object.nonzero

All other values are truthy (using the `bool()` function on them returns `True`).

```
bool(0)  # => False
bool("")  # => False
```

2. Variables and Collections

Python has a print statement

```
print "I'm Python. Nice to meet you!"
# => I'm Python. Nice to meet you!
```

Simple way to get input data from console

```
input_string_var = raw_input(
    "Enter some data: ")  # Returns the data as a string

input_var = input("Enter some data: ")  # Evaluates the data
## Warning: Caution is recommended for input() method usage
```

Note: In python 3, input() is deprecated and raw_input() is renamed to input()

No need to declare variables before assigning to them.

```
some_var = 5

# Convention is to use lower_case_with_underscores
some_var  # => 5
```

Accessing a previously unassigned variable is an exception.

See Control Flow to learn more about exception handling.

```
some_other_var  # Raises a name error
```

- if can be used as an expression

Accessing a previously unassigned variable is an exception.

See Control Flow to learn more about exception handling.

```
some_other_var  # Raises a name error
```

- if can be used as an expression
- Equivalent of C's '?:' ternary operator

```
"yahoo!" if 3 > 2 else 2  # => "yahoo!"
```

Lists store sequences

```
li = []

#You can start with a prefilled list
other_li = [4, 5, 6]
```

Add stuff to the end of a list with append

```
li.append(1)  # li is now [1]
li.append(2)  # li is now [1, 2]
li.append(4)  # li is now [1, 2, 4]
li.append(3)  # li is now [1, 2, 4, 3]

# Remove from the end with pop
li.pop()  # => 3 and li is now [1, 2, 4]

# Let's put it back
li.append(3)  # li is now [1, 2, 4, 3] again.
```

Access a list like you would any array

```
li[0]  # => 1

# Assign new values to indexes that have already
# been initialized with =

li[0] = 42
li[0]  # => 42
li[0] = 1  # Note: setting it back to the original value
# Look at the last element
li[-1]  # => 3
```

Looking out of bounds is an IndexError

```
li[4]  # Raises an IndexError
```

You can look at ranges with slice syntax.

It's a closed/open range for you mathy types.

```
li[1:3]  # => [2, 4]
# Omit the beginning
li[2:]  # => [4, 3]
# Omit the end
li[:3]  # => [1, 2, 4]
# Select every second entry
li[::2]  # =>[1, 4]
# Reverse a copy of the list
li[::-1]  # => [3, 4, 2, 1]
# Use any combination of these to make advanced slices
# li[start:end:step]
```

Remove arbitrary elements from a list with “del”

```
del li[2]  # li is now [1, 2, 3]
```

You can add lists

```
li + other_li  # => [1, 2, 3, 4, 5, 6]
## Note: values for li and for other_li are not modified.
```

Concatenate lists with “extend()”

```
li.extend(other_li)  # Now li is [1, 2, 3, 4, 5, 6]
```

Remove first occurrence of a value

```
li.remove(2)  # li is now [1, 3, 4, 5, 6]
li.remove(2)  # Raises a ValueError as 2 is not in the list
```

Insert an element at a specific index

```
li.insert(1, 2)  # li is now [1, 2, 3, 4, 5, 6] again
```

Get the index of the first item found

```
li.index(2)  # => 1
li.index(7)  # Raises a ValueError as 7 is not in the list
```

Check for existence in a list with “in”

```python
1 in li  # => True
```

Examine the length with “len()”

```
len(li)  # => 6
```

Tuples are like lists but are immutable.

```
tup = (1, 2, 3)
tup[0]  # => 1
tup[0] = 3  # Raises a TypeError
```

You can do all those list thingies on tuples too

```
len(tup)  # => 3
tup + (4, 5, 6)  # => (1, 2, 3, 4, 5, 6)
tup[:2]  # => (1, 2)
2 in tup  # => True
```

You can unpack tuples (or lists) into variables

```
a, b, c = (1, 2, 3)  # a is now 1, b is now 2 and c is now 3
d, e, f = 4, 5, 6  # you can leave out the parentheses
```

Tuples are created by default if you leave out the parentheses

```
g = 4, 5, 6  # => (4, 5, 6)
```

Now look how easy it is to swap two values

```
e, d = d, e  # d is now 5 and e is now 4
```

Dictionaries store mappings

```
empty_dict = {}

## Here is a prefilled dictionary
filled_dict = {"one": 1, "two": 2, "three": 3}
```

Source: learnpython

Look up values with [ ]

```
filled_dict["one"]  # => 1
```

Get all keys as a list with “keys()”

```
filled_dict.keys()  # => ["three", "two", "one"]

## Note - Dictionary key ordering is not guaranteed.
## Your results might not match this exactly.
```

Get all values as a list with “values()”

```
filled_dict.values()  # => [3, 2, 1]

## Note - Same as above regarding key ordering.
```

Get all key-value pairs as a list of tuples with “items()”

```
filled_dict.items()

# => [("one", 1), ("two", 2), ("three", 3)]
```

Check for existence of keys in a dictionary with “in”

```
"one" in filled_dict  # => True
1 in filled_dict  # => False
```

Looking up a non-existing key is a KeyError

```
filled_dict["four"]  # KeyError
```

Use “get()” method to avoid the KeyError

```
filled_dict.get("one")  # => 1
filled_dict.get("four")  # => None

## The get method supports a default argument when the value
filled_dict.get("one", 4)  # => 1
filled_dict.get("four", 4)  # => 4

## note that filled_dict.get("four") is still => None
## (get doesn't set the value in the dictionary)
```

set the value of a key with a syntax similar to lists

```
filled_dict["four"] = 4

# now, filled_dict["four"] => 4
```

“setdefault()” inserts into a dictionary only if the given key isn’t present

```
filled_dict.setdefault("five", 5)
# filled_dict["five"] is set to 5

filled_dict.setdefault("five", 6)
# filled_dict["five"] is still 5
```

Sets store . . . well sets (which are like lists but can contain no duplicates)

```
empty_set = set()

## Initialize a "set()" with a bunch of values
some_set = set([1, 2, 2, 3, 4])
# some_set is now set([1, 2, 3, 4])

## order is not guaranteed,
## even though it may sometimes look sorted

another_set = set([4, 3, 2, 2, 1])
# another_set is now set([1, 2, 3, 4])
```

Since Python 2.7, {} can be used to declare a set

```
filled_set = {1, 2, 2, 3, 4}  # => {1, 2, 3, 4}

## Add more items to a set
filled_set.add(5)
# filled_set is now {1, 2, 3, 4, 5}
```

Do set intersection with &

```
other_set = {3, 4, 5, 6}
filled_set & other_set  # => {3, 4, 5}
```

Do set union with |

```
filled_set | other_set  # => {1, 2, 3, 4, 5, 6}
```

Do set difference with -

```
{1, 2, 3, 4} - {2, 3, 5}  # => {1, 4}
```

Do set symmetric difference with ^

```
{1, 2, 3, 4} ^ {2, 3, 5}  # => {1, 4, 5}
```

Check if set on the left is a superset of set on the right

```
{1, 2} >= {1, 2, 3}  # => False
```

Check if set on the left is a subset of set on the right

```
{1, 2} <= {1, 2, 3}  # => True
```

Check for existence in a set with in

```
2 in filled_set  # => True
10 in filled_set  # => False
```

3. Control Flow

Let’s just make a variable

some_var = 5

if statement. Indentation is significant in python!

```
## prints "some_var is smaller than 10"
if some_var > 10:
    print "some_var is totally bigger than 10."
elif some_var < 10:  # This elif clause is optional.
    print "some_var is smaller than 10."
else:  # This is optional too.
    print "some_var is indeed 10."
```

For loops iterate over lists

```
for animal in ["dog", "cat", "mouse"]:
    # You can use {0} to interpolate formatted strings.
    # (See above.)
    print "{0} is a mammal".format(animal)

# prints:
#    dog is a mammal
#    cat is a mammal
#    mouse is a mammal
```

“range(number)” returns a list of numbers

from zero to the given number

```
"""
prints:
    0
    1
    2
    3
"""
for i in range(4):
    print i
```

“range(lower, upper)” returns a list of numbers

from the lower number to the upper number

```
"""
prints:
    4
    5
    6
    7
"""
for i in range(4, 8):
    print i
```

While loops go until a condition is no longer met.

```
"""
prints:
    0
    1
    2
    3
"""
x = 0
while x < 4:
    print x
    x += 1  # Shorthand for x = x + 1
```

Handle exceptions with a try/except block

Works on Python 2.6 and up:

```
try:
    # Use "raise" to raise an error
    raise IndexError("This is an index error")
except IndexError as e:
    pass  # Pass is just a no-op.
          # Usually you would do recovery here.
except (TypeError, NameError):
    pass  # Multiple exceptions can be handled together,
          # if required.
else:  # Optional clause to the try/except block. Must follow all e
    print "All good!"
          # Runs only if the code in try raises no exceptions
finally:  # Execute under all circumstances
    print "We can clean up resources here"
```

Instead of try/finally to cleanup resources you can use a with statement

```
with open("myfile.txt") as f:
    for line in f:
        print line
```

4. Functions

Use “def” to create new functions

```
def add(x, y):
    print "x is {0} and y is {1}".format(x, y)
    return x + y  # Return values with a return statement
```

Calling functions with parameters

```
add(5, 6)
# => prints out "x is 5 and y is 6" and returns 11
```

Another way to call functions is with keyword arguments

Keyword arguments can arrive in any order.

```
add(y=6, x=5)
```

Functions with a variable number of positional args will be interpreted as a tuple by using *

```
def varargs(*args):
    return args

varargs(1, 2, 3)  # => (1, 2, 3)
```

Functions may take a variable number of keyword args,

they will be interpreted as a dict by using **

```
def keyword_args(**kwargs):
    return kwargs

## Let's call it to see what happens
keyword_args(big="foot", loch="ness")
# => {"big": "foot", "loch": "ness"}
```

You can do both at once, if you like

```
def all_the_args(*args, **kwargs):
    print args
    print kwargs

"""
all_the_args(1, 2, a=3, b=4) prints:
    (1, 2)
    {"a": 3, "b": 4}
"""
```

When calling functions, you can do the opposite of args/kwargs!

Use * to expand positional args and use ** to expand keyword args

```
args = (1, 2, 3, 4)
kwargs = {"a": 3, "b": 4}

all_the_args(*args)
# equivalent to foo(1, 2, 3, 4)

all_the_args(**kwargs)
# equivalent to foo(a=3, b=4)

all_the_args(*args, **kwargs)
# equivalent to foo(1, 2, 3, 4, a=3, b=4)
```

you can pass args and kwargs along to other functions that take args/kwargs

by expanding them with * and ** respectively

```
def pass_all_the_args(*args, **kwargs):
    all_the_args(*args, **kwargs)
    print varargs(*args)
    print keyword_args(**kwargs)
```

Function Scope

```
x = 5

def set_x(num):
    # Local var x not the same as global variable x
    x = num  # => 43
    print x  # => 43

def set_global_x(num):
    global x
    print x  # => 5
    x = num  # global var x is now set to 6
    print x  # => 6

set_x(43)
set_global_x(6)
```

Python has first class functions

```
def create_adder(x):
    def adder(y):
        return x + y

    return adder

add_10 = create_adder(10)
add_10(3)  # => 13
```

There are also anonymous functions

```
(lambda x: x > 2)(3)  # => True
(lambda x, y: x ** 2 + y ** 2)(2, 1)  # => 5
```

There are built-in higher order functions

```
map(add_10, [1, 2, 3])  # => [11, 12, 13]
map(max, [1, 2, 3], [4, 2, 1])  # => [4, 2, 3]

filter(lambda x: x > 5, [3, 4, 5, 6, 7])  # => [6, 7]
```

We can use list comprehensions for nice maps and filters

```
[add_10(i) for i in [1, 2, 3]]  # => [11, 12, 13]
[x for x in [3, 4, 5, 6, 7] if x > 5]  # => [6, 7]
```

You can construct set and dict comprehensions as well.

```
{x for x in 'abcddeef' if x in 'abc'}
# => {'a', 'b', 'c'}

{x: x ** 2 for x in range(5)}
# => {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

5. Classes

We subclass from object to get a class (1/3)

```
class Human(object):
    # A class attribute. It is shared by all instances of this class
    species = "H. sapiens"

    # Basic initializer, this is called when this class is instantiated.
    # Note that the double leading and trailing underscores denote objects
    # or attributes that are used by python but that live in user-controlle
    # namespaces. You should not invent such names on your own.
    def __init__(self, name):
        # Assign the argument to the instance's name attribute
        self.name = name

        # Initialize property
        self.age = 0
```

Continue (2/3)

```python
# An instance method. All methods take "self" as the first argument
def say(self, msg):
    return "{0}: {1}".format(self.name, msg)

# A class method is shared among all instances
# They are called with the calling class as the first argument
@classmethod
def get_species(cls):
    return cls.species

# A static method is called without a class or instance reference
@staticmethod
def grunt():
    return "*grunt*"
```

Continue (3/3)

```
# A property is just like a getter.
# It turns the method age() into an read-only attribute
# of the same name.
@property
def age(self):
    return self._age

# This allows the property to be set
@age.setter
def age(self, age):
    self._age = age

# This allows the property to be deleted
@age.deleter
def age(self):
    del self._age
```

Instantiate a class

```
i = Human(name="Ian")
print i.say("hi")
# prints out "Ian: hi"

j = Human("Joel")
print j.say("hello")
# prints out "Joel: hello"
```

Call our class method

```
i.get_species()  # => "H. sapiens"
```

Change the shared attribute

```
Human.species = "H. neanderthalensis"
i.get_species()  # => "H. neanderthalensis"
j.get_species()  # => "H. neanderthalensis"
```

Call the static method

```
Human.grunt()   # => "*grunt*"
```

Update the property

```
i.age = 42
```

Get the property

```
i.age  # => 42
```

Delete the property

```
del i.age
i.age  # => raises an AttributeError
```

6. Modules

You can import modules

```
import math

print math.sqrt(16)  # => 4
```

You can get specific functions from a module

```
from math import ceil, floor

print ceil(3.7)  # => 4.0
print floor(3.7)  # => 3.0
```

You can import all functions from a module.

```
## Warning: this is not recommended
from math import *
```

You can shorten module names

```
import math as m

math.sqrt(16) == m.sqrt(16)  # => True

## you can also test that the functions are equivalent
from math import sqrt

math.sqrt == m.sqrt == sqrt  # => True
```

Python modules are just ordinary python files.

```
## You can write your own, and import them. The name of the
## module is the same as the name of the file.

## You can find out which functions and attributes
## defines a module.
import math

dir(math)

## If you have a Python script named math.py in the same
## folder as your current script, the file math.py will
## be loaded instead of the built-in Python module.
## This happens because the local folder has priority
## over Python's built-in libraries.
```

7. Advanced

Generators (1/4)

```
## A generator "generates" values as they are requested instead of storing
## everything up front

## The following method (*NOT* a generator) will double all values and
## store it in `double_arr`. For large size of iterables,  might get huge!
def double_numbers(iterable):
    double_arr = []
    for i in iterable:
        double_arr.append(i + i)
    return double_arr

## Running the following would mean we'll double all values first and
## return all of them back to be checked by our condition
for value in double_numbers(range(1000000)):  # `test_non_generator`
    print value
    if value > 5:
        break
```

Generators (2/4)

```
## We could instead use a generator to "generate" the doubled value
## as the item is being requested

def double_numbers_generator(iterable):
    for i in iterable:
        yield i + i

## Running the same code as before, but with a generator, now allows us
## to iterate over the values and doubling them one by one as they are
## being consumed by our logic. Hence as soon as we see a value > 5, we
## break out of the loop and don't need to double most of the values
## sent in

## MUCH FASTER!

for value in double_numbers_generator(xrange(1000000)):  # `test_generator`
    print value
    if value > 5:
        break
```

Generators (3/4)

```
## BTW: did you notice the use of `range` in `test_non_generator`
## and `xrange` in `test_generator`?

## Just as `double_numbers_generator` is the generator version of
## `double_numbers` We have `xrange` as the generator version of `range`
## `range` would return back and array with 1000000 values for us to use
## `xrange` would generate 1000000 values for us as we request / iterate
## over those items

## Just as you can create a list comprehension, you can create generator
## comprehensions as well.

values = (-x for x in [1, 2, 3, 4, 5])
for x in values:
    print(x)  # prints -1 -2 -3 -4 -5 to console/terminal
```

Generators (4/4)

```
## You can also cast a generator comprehension directly to a list.

values = (-x for x in [1, 2, 3, 4, 5])
gen_to_list = list(values)

print(gen_to_list)  # => [-1, -2, -3, -4, -5]
```

Decorators (1/4)

```
## A decorator is a higher order function, which accepts and returns
## a function.

## Simple usage example - add_apples decorator will add 'Apple' element
## into fruits list returned by get_fruits target function.

def add_apples(func):
    def get_fruits():
        fruits = func()
        fruits.append('Apple')
        return fruits
    return get_fruits
```

Decorators (2/4)

```
@add_apples
def get_fruits():
    return ['Banana', 'Mango', 'Orange']

## Prints out the list of fruits with 'Apple' element in it:
## Banana, Mango, Orange, Apple
print ', '.join(get_fruits())
```

Decorators (3/4)

```python
## in this example beg wraps say
## Beg will call say. If say_please is True
## then it will change the returned message

from functools import wraps

def beg(target_function):
    @wraps(target_function)
    def wrapper(*args, **kwargs):
        msg, say_please = target_function(*args, **kwargs)
        if say_please:
            return "{} {}".format(msg, "Please! I am poor :(")
        return msg

    return wrapper
```

Decorators (4/4)

```
@beg
def say(say_please=False):
    msg = "Can you buy me a beer?"
    return msg, say_please


print say()
# Can you buy me a beer?

print say(say_please=True)
# Can you buy me a beer? Please! I am poor :(
```