Is Python strongly typed?

Asked 12 years, 7 months ago Modified 2 years, 9 months ago Viewed 263k times

371

I've come across links that say Python is a strongly typed language.

However, I thought in strongly typed languages you couldn't do this:

```
bob = 1
bob = "bob"
```

I thought a strongly typed language didn't accept type-changing at run-time. Maybe I've got a wrong (or too simplistic) definition of strong/weak types.

So, is Python a strongly or weakly typed language?

python strong-typing weak-typing

Share Improve this question

Follow

edited Jun 15, 2019 at 2:07

Tom Zych

13.6k ● 9 ● 37 ● 54

asked Jul 4, 2012 at 12:16

Pacane

21.5k ● 18 ● 65 ● 101

13 Answers

Sorted by: Highest score (default)

570

Python is strongly, dynamically typed.

- **Strong** typing means that the type of a value doesn't change in unexpected ways. A string containing only digits doesn't magically become a number, as may happen in Perl. Every change of type requires an explicit conversion.
- **Dynamic** typing means that runtime objects (values) have a type, as opposed to static typing where variables have a type.

As for your example

```
bob = 1
bob = "bob"
```

This works because the variable does not have a type; it can name any object. After `bob=1`, you'll find that `type(bob)` returns `int`, but after `bob="bob"`, it returns `str`. (Note that `type` is a regular function, so it evaluates its argument, then returns the type of the value.)

Contrast this with older dialects of C, which were weakly, statically typed, so that pointers and integers were pretty much interchangeable. (Modern ISO C requires conversions in many cases, but my compiler is still lenient about this by default.)

I must add that the strong vs. weak typing is more of a continuum than a boolean choice. C++ has stronger typing than C (more conversions required), but the type system can be subverted by using pointer casts.

The strength of the type system in a dynamic language such as Python is really determined by how its primitives and library functions respond to different types. E.g., `+` is overloaded so that it works on two numbers *or* two strings, but not a string and an number. This is a design choice made when `+` was implemented, but not really a necessity following from the language's semantics. In fact, when you overload `+` on a custom type, you can make it implicitly convert anything to a number:

```
def to_number(x):
    """Try to convert function argument to float-type object."""
    try:
        return float(x)
    except (TypeError, ValueError):
        return 0

class Foo:
    def __init__(self, number):
        self.number = number

    def __add__(self, other):
        return self.number + to_number(other)
```

Instance of class `Foo` can be added to other objects:

```
>>> a = Foo(42)
>>> a + "1"
43.0
>>> a + Foo
42
>>> a + 1
43.0
>>> a + None
42
```

Observe that even though strongly typed Python is completely fine with adding objects of type `int` and `float` and returns an object of type `float` (e.g., `int(42) + float(1)` returns `43.0`). On the other hand, due to the mismatch between types Haskell would complain if one tries the following `(42 :: Integer) + (1 :: Float)`. This makes Haskell a strictly typed language, where types are entirely disjoint and only a controlled form of overloading is possible via type classes.

Share Improve this answer Follow edited Apr 28, 2020 at 7:40 community wiki 8 revs, 3 users 57% Fred Foo

28 One example I don't see very often but I think is important to show that Python is not completely strongly typed, is all the things that evaluate to boolean: docs.python.org/release/2.5.2/lib/truth.html – gsgx Apr 4, 2013 at 19:53

37 Not so sure if this is a counter example: Things can evaluate to a boolean, but they don't suddenly "become" a boolean. It's almost as if someone implicitly called something like as_boolean(<value>), which is not the same as the type of the object itself changing, right? – jbrendel Jan 7, 2014 at 20:33

24 Being truthy in boolean context is not a counterexample, because nothing is actually being converted to `True` or `False`. But what about number promotion? `1.0 + 2` works just as well in Python as it does in Perl or C, even though `"1.0" + 2` doesn't. I agree with @jbrendel that this isn't really an implicit conversion, it's just overloading—but in the same sense, Perl isn't doing any implicit conversion either. If functions don't have declared parameter types, there's nowhere for implicit conversions to happen. – abarnert Aug 10, 2014 at 18:18

28 A better way to think about *strong* typing is that type matters when performing operations on a variable. If the type is not as expected, a language that complains is **strongly typed** (python/java) and one that isn't is **weakly typed** (javascript) **Dynamically typed** languages(python) are those which allow the type of a variable to change at runtime whereas **statically typed** languages(java) do not allow this once a variable is declared. – keios Sep 26, 2015 at 19:07

5 @gsingh2011 Truthiness is useful and not *weak typing* on its own, but an accidental `if isValid(value) - 1` can leak. The boolean is coerced into integer, which is then evaluated as a truthy value. `False - 1` becomes truthy and `True - 1` becomes falsy, leading to an embarrassingly difficult two-layered off-by-one error to debug. In this sense, python is *mostly* strongly typed; type coercions don't usually cause logical errors. – Aaron3468 Aug 27, 2016 at 6:34

93

There are some important issues that I think all of the existing answers have missed.

Weak typing means allowing access to the underlying representation. In C, I can create a pointer to characters, then tell the compiler I want to use it as a pointer to integers:

```
char sz[] = "abcdefg";
int *i = (int *)sz;
```

On a little-endian platform with 32-bit integers, this makes `i` into an array of the numbers `0x64636261` and `0x00676665`. In fact, you can even cast pointers themselves to integers (of the appropriate size):

```
intptr_t i = (intptr_t)&sz;
```

And of course this means I can overwrite memory anywhere in the system.*

```
char *spam = (char *)0x12345678
spam[0] = 0;
```

* Of course modern OS's use virtual memory and page protection so I can only overwrite my own process's memory, but there's nothing about C itself that offers such protection, as anyone who ever coded on, say, Classic Mac OS or Win16 can tell you.

Traditional Lisp allowed similar kinds of hackery; on some platforms, double-word floats and cons cells were the same type, and you could just pass one to a function expecting the other and it would "work".

Most languages today aren't quite as weak as C and Lisp were, but many of them are still somewhat leaky. For example, any OO language that has an unchecked "downcast",* that's a type leak: you're essentially telling the compiler "I know I didn't give you enough information to know this is safe, but I'm pretty sure it is," when the whole point of a type system is that the compiler always has enough information to know what's safe.

* A checked downcast doesn't make the language's type system any weaker just because it moves the check to runtime. If it did, then subtype polymorphism (aka virtual or fully-dynamic function calls) would be the same violation of the type system, and I don't think anyone wants to say that.

Very few "scripting" languages are weak in this sense. Even in Perl or Tcl, you can't take a string and just interpret its bytes as an integer.* But it's worth noting that in CPython (and similarly for many other interpreters for many languages), if you're really persistent, you can use `ctypes` to load up `libpython`, cast an object's `id` to a `POINTER(Py_Object)`, and force the type system to leak. Whether this makes the type system weak or not depends on your use cases—if you're trying to implement an in-language restricted execution sandbox to ensure security, you do have to deal with these kinds of escapes…

* You can use a function like `struct.unpack` to read the bytes and build a new int out of "how C would represent these bytes", but that's obviously not leaky; even Haskell allows that.

Meanwhile, implicit conversion is really a different thing from a weak or leaky type system.

Every language, even Haskell, has functions to, say, convert an integer to a string or a float. But some languages will do some of those conversions for you automatically—e.g., in C, if you call a function that wants a `float`, and you pass it in `int`, it gets converted for you. This can definitely lead to bugs with, e.g., unexpected overflows, but they're not the same kinds of bugs you get from a weak type system. And C isn't really being any weaker here; you can add an int and a float in Haskell, or even concatenate a float to a string, you just have to do it more explicitly.

And with dynamic languages, this is pretty murky. There's no such thing as "a function that wants a float" in Python or Perl. But there are overloaded functions that do different things with different types, and there's a strong intuitive sense that, e.g., adding a string to something else is "a function that wants a string". In that sense, Perl, Tcl, and JavaScript appear to do a lot of implicit conversions (`"a" + 1` gives you `"a1"`), while Python does a lot fewer (`"a" + 1` raises an exception, but `1.0 + 1` does give you `2.0`*). It's just hard to put that sense into formal terms—why shouldn't there be a `+` that takes a string and an int, when there are obviously other functions, like indexing, that do?

* Actually, in modern Python, that can be explained in terms of OO subtyping, since `isinstance(2, numbers.Real)` is true. I don't think there's any sense in which `2` is an instance of the string type in Perl or JavaScript… although in Tcl, it actually is, since *everything* is an instance of string.

Finally, there's another, completely orthogonal, definition of "strong" vs. "weak" typing, where "strong" means powerful/flexible/expressive.

For example, Haskell lets you define a type that's a number, a string, a list of this type, or a map from strings to this type, which is a perfectly way to represent anything that can be decoded from JSON. There's no way to define such a type in Java. But at least Java has parametric (generic) types, so you can write a function that takes a List of T and know that the elements are of type T; other languages, like early Java, forced you to use a List of Object and downcast. But at least Java lets you create new types with their own methods; C only lets you create structures. And BCPL didn't even have that. And so on down to assembly, where the only types are different bit lengths.

So, in that sense, Haskell's type system is stronger than modern Java's, which is stronger than earlier Java's, which is stronger than C's, which is stronger than BCPL's.

So, where does Python fit into that spectrum? That's a bit tricky. In many cases, duck typing allows you to simulate everything you can do in Haskell, and even some things you can't; sure, errors are caught at runtime instead of compile time, but they're still caught. However, there are cases where duck typing isn't sufficient. For example, in Haskell, you can tell that an empty list of ints is a list of ints, so you can decide that reducing `+` over that list should return 0*; in Python, an empty list is an empty list; there's no type information to help you decide what reducing `+` over it should do.

* In fact, Haskell doesn't let you do this; if you call the reduce function that doesn't take a start value on an empty list, you get an error. But its type system is powerful enough that you *could* make this work, and Python's isn't.

Share Improve this answer Follow edited Aug 10, 2014 at 18:22 answered Aug 10, 2014 at 18:15

abarnert
366k ● 54 ● 622 ● 690

4 This answer is brilliant! A shame it's languished for so long at the bottom of the list. – LeoR May 28, 2015 at 12:59

4 Just a little comment to your C example: `char sz[]` is not a pointer to char, it is array of char, and in the assignment it decays into pointer. – majkel.mk Feb 17, 2017 at 18:54

In 2021, this one is still a brilliant answer! – Michael Lee Nov 23, 2021 at 16:39

90

TLDR;

Python typing is **Dynamic** so you can *change* a string variable to an int (in a **Static** language you can't)

```
x = 'somestring'
x = 50
```

Python typing is **Strong** so you can't merge types:

```
'foo' + 3 --> TypeError: cannot concatenate 'str' and 'int' objects
```

In weakly-typed Javascript this happens...

```
 'foo'+3 = 'foo3'
```

Regarding Type Inference

Some languages like Java force you to explicitly declare your object types

```
int x = 50;
```

Others like Kotlin simply infer it's an `int` from the value itself

```
x = 50
```

But because both languages use *static* types, `x` can't be changed from an `int`. Neither language would allow a *dynamic* change like

```
x = 50
x = 'now a string'
```

Share Improve this answer Follow edited Apr 3, 2022 at 14:33 answered Jul 27, 2017 at 19:01

Adam Hughes
16.3k ● 14 ● 94 ● 139

I don't know the details of Javascript but `'x' + 3` may be `operator+` overloading and doing the type conversion behind the scene? – NeoZoom.lua May 8, 2019 at 13:18

21 Anyway, your answer is actually more concise and easy to understand than the above ones. – NeoZoom.lua May 8, 2019 at 13:34

3 JavaScript developer here, best explanation ever! – Wenfang Du Jan 26, 2021 at 1:58

1 It could be argued that the existence of `int / int = float` (and in general, the fact an int can be used almost anywhere a float can) makes Python3 *slightly* weak in its typing. – o11c Oct 3, 2022 at 5:04

What about `<int> + <float>`? Python here does implicit conversion, does it? So why strongly typed? – Iskander14yo Apr 12, 2024 at 17:30

48

You are confusing 'strongly typed' with 'dynamically typed'.

I cannot change the type of `1` by adding the string `'12'`, but I can choose what types I store in a variable and change that during the program's run time.

The opposite of dynamic typing is static typing; the *declaration of variable types* doesn't change during the lifetime of a program. The opposite of strong typing is weak typing; the

type of *values* can change during the lifetime of a program.

Share Improve this answer Follow edited Jul 4, 2012 at 12:23 answered Jul 4, 2012 at 12:18

Martijn Pieters
1.1m ● 320 ● 4.2k ● 3.4k

1 The description in the link strongly typed: "Generally, a strongly typed language has stricter typing rules at compile time, which implies that errors and exceptions are more likely to happen during compilation." implies Python is a weakly typed language..., is wiki wrong? – NeoZoom.lua May 8, 2019 at 13:12

1 @secret that's not implied at all. Python has strict typing rules at compile time, each object created has just one type. And 'generally' doesn't imply anything, it just means that Python is an exception to that. – Martijn Pieters May 8, 2019 at 13:19

23

According to this wiki Python article Python is both dynamically and strongly typed (provides a good explanation too).

Perhaps you are thinking about *statically typed* languages where types can not change during program execution and type checking occurs during compile time to detect possible errors.

This SO question might be of interest: Dynamic type languages versus static type languages and this Wikipedia article on Type Systems provides more information

Share Improve this answer Follow edited May 23, 2017 at 12:34

Community Bot
1 ● 1

answered Jul 4, 2012 at 12:18

Levon
143k ● 35 ● 203 ● 193

11

It's already been answered a few times, but Python is a strongly typed language:

```
>>> x = 3
>>> y = '4'
>>> print(x+y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The following in JavaScript:

```
var x = 3
var y = '4'
alert(x + y) //Produces "34"
```

That's the difference between weak typing and strong typing. Weak types automatically try to convert from one type to another, depending on context (e.g. Perl). Strong types *never* convert implicitly.

Your confusion lies in a misunderstanding of how Python binds values to names (commonly referred to as variables).

In Python, names have no types, so you can do things like:

```
bob = 1
bob = "bob"
bob = "An Ex-Parrot!"
```

And names can be bound to anything:

```
>>> def spam():
...     print("Spam, spam, spam, spam")
...
>>> spam_on_eggs = spam
>>> spam_on_eggs()
Spam, spam, spam, spam
```

For further reading:

https://en.wikipedia.org/wiki/Dynamic_dispatch

and the slightly related but more advanced:

http://effbot.org/zone/call-by-object.htm

Share Improve this answer Follow

answered Jul 4, 2012 at 12:34

Wayne Werner
51.8k ● 34 ● 209 ● 301

1 Several years later - another useful and relevant resource: youtu.be/_AEJHKGk9ns – Wayne Werner May 1, 2015 at 23:19

Strong vs weak typing has nothing to do with the result type of expressions like 3+'4'. JavaScript is just as strong as Python for this example. – qznc Sep 26, 2015 at 16:40

3 @oneloop that's not necessarily true, it's just that the behavior for combining floats and ints is well-defined, and results in a float. You can do `"3"*4` in python, too. The result of course, is `"3333"`. You wouldn't say that it's converting either thing. Of course that could be just arguing semantics. – Wayne Werner May 31, 2016 at 4:12

1 @oneloop It's not necessarily true that because Python produces `float` out of the combination of `float` and `int` that it's converting the type implicitly. There is a natural relationship between float and int, and indeed, the type heirarchy spells that out. I suppose that you could argue Javascript considers `'3'+4` and `'e'+4` to both be well-defined operations in the same way that Python considers `3.0 + 4` to be well-defined, but at that point then there's really no such thing as strong or weak types, just (un)defined operations. – Wayne Werner May 31, 2016 at 14:07

2 At that point, I think the line I would draw is that any language with a `TypeError` (or equivalent) is a strongly typed language. – Wayne Werner May 31, 2016 at 14:08

The term "strong typing" does not have a definite definition.

11

Therefore, the use of the term depends on with whom you're speaking.

I do not consider any language, in which the type of a variable is not either explicitly declared, or statically typed to be strongly typed.

Strong typing doesn't just preclude conversion (for example, "automatically" converting from an integer to a string). It precludes assignment (i.e., changing the type of a variable).

If the following code compiles (interprets), the language is not strong-typed:

Foo = 1 Foo = "1"

In a strongly typed language, a programmer can "count on" a type.

For example, if a programmer sees the declaration,

UINT64 kZarkCount;

and he or she knows that 20 lines later, kZarkCount is still a UINT64 (as long as it occurs in the same block) - without having to examine intervening code.

Share Improve this answer Follow

answered Sep 13, 2015 at 5:20

user5330045
111 ● 1 ● 2

9

A Python variable stores an untyped reference to the target object that represent the value.

Any assignment operation means assigning the untyped reference to the assigned object -- i.e. the object is shared via the original and the new (counted) references.

The value type is bound to the target object, not to the reference value. The (strong) type checking is done when an operation with the value is performed (run time).

In other words, variables (technically) have no type -- it does not make sense to think in terms of a variable type if one wants to be exact. But references are automatically dereferenced and we actually think in terms of the type of the target object.

Share Improve this answer Follow

answered Jul 4, 2012 at 13:15

pepr
20.8k ● 15 ● 82 ● 144

3

I just discovered a superb concise way to memorize it:

> Dynamic/static typed expression; strongly/weakly typed value.

Share Improve this answer Follow edited Nov 10, 2021 at 12:52 answered May 8, 2019 at 13:38

akauppi
18k ● 15 ● 102 ● 125

NeoZoom.lua
2,901 ● 4 ● 40 ● 83

3

The existing answers mostly agree that Python is a strongly-typed language because it doesn't implicitly convert values from one type to another. Those answers mention the case of adding a string to an integer to support this claim; `"foo" + 3` raises a `TypeError` in Python, whereas in Javascript (generally considered to be a weakly-typed language), the number `3` is implicitly converted to a string and then concatenated, so the result is the string `"foo3"`.

But there are some other situations where Python *does* do implicit type conversions:

```
# implicit conversion from int to float
1 + 1.0

# implicit conversion from list to bool
if []: pass
```

In comparison, F# (generally considered to be a strongly-typed language) disallows both of these:

```
  1 + 1.0;;
  ----^^^
error FS0001: The type 'float' does not match the type 'int'

  if [] then 1 else 2;;
  ---^^
error FS0001: This expression was expected to have type bool but here has type 'a
list
```

So really there isn't a strict dichotomy of "strongly-typed" and "weakly-typed" languages, rather we can say that Python is more strongly-typed than Javascript but not as strongly-typed as F#.

Share Improve this answer Follow

answered Nov 30, 2021 at 21:48

kaya3
51.1k ● 7 ● 83 ● 115

1 If str + int is acceptable (in other answers) as an example to show that Python is strongly typed, why is float + int not acceptable as an example to show that it is not as strongly typed as some other languages? – kaya3 Apr 27, 2024 at 11:49

1 I will repeat since you ignored it: since you are so insistent that it doesn't matter whether or not `int + float` does a conversion, why are you not equally insistent that it doesn't matter whether or not `str + int` does a conversion? There are plenty of other answers to this question which say `str + int` does matter, but you aren't complaining about those. My answer only says that `int + float` should matter as much as `str + int` — and that there are languages where `int + float` is an error, the same way `str + int` is an error in Python. – kaya3 May 8, 2024 at 14:53

1 @PeterR "By my definition"? I did not propose any definition for "weakly typed" or "strongly typed", so do not put words into my mouth. I used the definition adopted by other answers, and only said that F# is *more* strongly typed than Python. It's right there in my answer: *"So really there isn't a strict dichotomy of "strongly-typed" and "weakly-typed" languages"* — so it is quite ridiculous for you to say that I am

placing languages on the wrong side of a dichotomy which I explicitly disavowed entirely. – kaya3 May 9, 2024 at 13:38

1 On the other hand, your definition that a language is strongly typed if "the rules are clearly defined" would make every language strongly typed, unless the language has no specification or some expressions like `int + float` have undefined behaviour; otherwise "the rules are clearly defined". That is a rather unorthodox definition — there are many different definitions of "strongly typed", but "the rules are clearly defined" seems to be one you made up yourself. – kaya3 May 9, 2024 at 13:43

1 But don't take my word for it! JavaScript likewise "clearly defines" the behaviour of all operators for all combinations of types, some of them perform conversions while others (e.g. `1 + 1n`) throw errors, and yet Wikipedia says JavaScript is weakly typed! Perhaps Wikipedia is not using the definition you just made up? – kaya3 May 9, 2024 at 13:44

2

i think, this simple example should you explain the diffs between strong and dynamic typing:

```
>>> tup = ('1', 1, .1)
>>> for item in tup:
...     type(item)
...
<type 'str'>
<type 'int'>
<type 'float'>
>>>
```

java:

```
public static void main(String[] args) {
        int i = 1;
        i = "1"; //will be error
        i = '0.1'; // will be error
    }
```

Share Improve this answer Follow

answered Jul 4, 2012 at 13:35

Dmitry Zagorulkin
8,548 ● 4 ● 38 ● 61

Your python code demonstrates dynamic typing while the java demonstrates static typing. A better example would be $var = '2' + 1 // result is 3 – erichlf Feb 25, 2015 at 22:02

@ivleph i agree. it's also possible to write something like this: "a" * 3 == "aaa" – Dmitry Zagorulkin Feb 26, 2015 at 9:09

2

Python is strongly typed because it has no *unchecked* dynamic type errors. In other words, you cannot violate its type system.

Definitions

- *Type:* set of values. It partially defines the intended usage (behaviour) of its values. It can be specified in extension with an enumeration or in intension with a predicate.
- *Type system:* system checking that values are used as intended, avoiding undefined behaviour partially. Undefined behaviour should be avoided because it can lead to late program crash or silent loss of data and production of incorrect results.
- *Typed language:* language with a type system.
- *Type error:* program error checkable by a type system.
- *Expression:* program text denoting a value.
- *Static/dynamic type:* compile-time/run-time type of an expression. The run-time type of an expression is the type of the value that it denotes.
- *Static/dynamic type system:* type system checking static/dynamic types.
- *Statically/dynamically typed language:* language with a static/dynamic type system.
- *Static/dynamic type error:* program error checkable by a static/dynamic type system.
- *Weakly/strongly typed language:* language with/without unchecked dynamic type errors. Statically typed, dynamically typed, or both imply strongly typed.
- *Monomorphic/polymorphic expression:* expression having a single dynamic type/multiple dynamic types. A monomorphic expression has single intended usage, and a polymorphic expression has multiple intended usage.
- *Universal/ad-hoc polymorphic expression:* real/virtual polymorphic expression. A real polymorphic expression denotes a single value that has multiple types, and a virtual polymorphic expression denotes multiple values that have single types.
- *Parametric/inclusion polymorphic expression:* universal polymorphic expression based on generic types/subtypes of a type (e.g. the C++ expression `&` denotes a single `T* (T&)` operator value where `T` is a generic type/the C++ expression `std::exception` denotes a single `S` class value where `S` is a generic subtype of `std::exception`).
- *Overloading/coercion polymorphic expression:* ad-hoc polymorphic expression based on expression/value conversion (e.g. the C++ expression `+` denotes `int (int&, int&)` and `float (float&, float&)` operator values/the C++ expression `3.5` denotes `float` and `bool` values).

Reference

Cardelli (Luca), Wegner (Peter), 'On Understanding Types, Data Abstraction, and Polymorphism', *Computing Surveys,* volume 17, issue 4, 1985, p. 471-523, DOI: https://doi.org/10.1145/6041.6042.

Share Improve this answer Follow edited Apr 11, 2022 at 16:47

answered Apr 3, 2022 at 23:48

Géry Ogam
8,037 ● 5 ● 47 ● 84

-3

```
class testme(object):
    ''' A test object '''
    def __init__(self):
        self.y = 0

def f(aTestMe1, aTestMe2):
    return aTestMe1.y + aTestMe2.y




c = testme            #get a variable to the class
c.x = 10              #add an attribute x inital value 10
c.y = 4               #change the default attribute value of y to 4

t = testme()          # declare t to be an instance object of testme
r = testme()          # declare r to be an instance object of testme

t.y = 6               # set t.y to a number
r.y = 7               # set r.y to a number

print(f(r,t))         # call function designed to operate on testme objects

r.y = "I am r.y"      # redefine r.y to be a string

print(f(r,t))         #POW!!!!  not good....
```

The above would create a nightmare of unmaintainable code in a large system over a long period time. Call it what you want, but the ability to "dynamically" change a variables type is just a bad idea...

Share Improve this answer Follow

answered Aug 2, 2013 at 17:40

Ryan Alexander

21