Some of the things which make a programming language mathematically elegant: properties of basic operations in Python

Version of May 20, 2010, for Python 3.2
©2010 Jan Plaza
Comments welcome.

Jan Plaza

SUNY Plattsburgh
Computer Science Department
101 Broad Street
Plattsburgh, NY 12901
U.S.A.

jan.plaza@plattsburgh.edu

Abstract
This paper comments on basic properties of constructs id, is, is not, ==, !=, repr, eval, hash in Python 3. Their implementation satisfies many properties commonly used in mathematics, however a programmer who is not careful can define objects which will violate them – one goal of the paper is to warn about such possibilities. Another goal of this paper is to state explicitly that some natural properties are false and either to ask if it is feasible to uphold them or to look for a rationale for not doing so.

The properties listed in the paper are labeled as True, True?, False?, False. Definite statement True is made if the official Python 3 documentation justifies it. Definite statement False is made if we know a programming counter-example. The statements True?, False?, with question marks, relate the author’s conviction but require verification. The status of statements labeled True?, False? and False will remain not fully clear unless they are included in future versions of official Python Language Reference and Python Standard Library Reference.

Contents

Title page 1

1 Preliminaries 3
1 Preliminaries

The quotes and examples in the paper concern Python 3.1.1+ or 3.1.2; The tests were made on a 64-bit Intel compatible machine under Linux (which affects the precision of floating point numbers).

Although properties of arithmetical operations and numeric comparisons make significant contribution to mathematical elegance of Python they are not in the scope of the current version of this paper. Also, the wide extent of orthogonality of programing constructs in Python contributes to its elegance, however it is not in the scope of this writing. Also, the issue of non-destructive assignment will not be considered.

2 Mathematical and programming concepts

We need to avoid confusing mathematical concepts and programming concepts. The term such as “set”, “function” are used both in mathematics and Python but have different meanings. This section explains how we are going to use some common terms and what new terms are needed.

2.1 Mathematical concepts

- **set** – a set as understood in mathematics.

- **relation** and **function** – relation and function as understood in set theory (sets of ordered tuples).

- **domain()** and **range()** – domain and range of relations and functions as understood in set theory.

- Let \( A \) be a set. Let \( f \) be an \( n \)-argument function s.t. \( A^n \subseteq \text{domain}(f) \). Let \( \sim \) be a binary relation. We say that \( f \) **preserves \( \sim \) on \( A \)** if
  \[
  x_1 \sim y_1 \text{ and } ... \text{ and } x_2 \sim y_2 \text{ imply } f(x_1, ..., x_n) \sim f(y_1, ..., y_n)
  \]
  for any \( x_1, ..., x_n, y_1, ..., y_n \in A \).

  If \( \sim \) is an equivalence relation such that \( A \subseteq \text{domain}(\sim) \) and function \( f \) preserves \( \sim \) on \( A \) then \( \sim \) is called
  \( a \) **congruence on \( A \) with respect to \( f \).**

- Let \( A \) be a set. Let \( R \) be an \( n \)-argument relation. Let \( \sim \) be a binary relation. We say that \( R \) **preserves \( \sim \) on \( A \)** if
  \[
  x_1 \sim y_1 \text{ and } ... \text{ and } x_2 \sim y_2 \text{ imply } (R(x_1, ..., x_n) \text{ iff } R(y_1, ..., y_n))
  \]
  for any \( x_1, ..., x_n, y_1, ..., y_n \in A \).

  If \( \sim \) is an equivalence relation such that \( A \subseteq \text{domain}(\sim) \) and relation \( R \) preserves \( \sim \) on \( A \) then \( \sim \) is called
  \( a \) **congruence on \( A \) with respect to \( R \).**
2.2 Python concepts

Objects are central to the philosophy of Python. Strings, tuples, lists, dictionaries are objects. Unlike in Java integers, floats and other numbers are objects. Also, functions, types, classes and modules are objects – pushing the concept of an object to the desired extreme. Every object has a unique identity and type which do not change through the list of the object; they can be found by applying built-in functions \texttt{id} and \texttt{type}.

- \textbf{Python object} – an object as understood in Python,
- \textbf{Python callable} – a callable object as understood in Python (including function objects, method objects and class objects),
- \textbf{Python function} – a function object as understood in Python (including built-in function objects and user-defined function objects),
- \textbf{false-equivalents} – Python object representing truth value false: \texttt{None}, \texttt{False}, 0, 0.0, 0+0j, empty sequences, empty mappings, etc.
- \textbf{true-equivalents} – Python object other than false-equivalents.

In this paper, when we refer to Python concepts we use the adjective “Python”, but this convention is not used in quotes we present.

2.3 Concepts specific to this paper

We will define Python functional expressions – syntactical items which can be supplied with arguments and evaluated in Python; they give raise to functions and relations (in the sense of mathematics).

2.1 Definition By a **Python functional expression** we understand any of the following

- any Python expression that evaluates to a Python callable,
- any Python operator:
  +, -, *, **, /, //, %, 
  \%, <<, >>, &l, |, ^, ~, 
  <, >, <=, >=, ==, !=, 
  is, is not, in, not in, 
  and, or, not,
  ... if ... else ...

For any Python functional expression $e$ we say that $e$ can take $n$-arguments if

- $e$ is a Python expression that evaluates to a Python callable which can be called with $n$ parameters, or
- $e$ is a Python operator among those above and it operates on $n$ objects.
For all Python functional expressions in this paper we reserve the right to use the prefix notation, for instance, writing \( \texttt{is(x,y)} \) as an alternative to \( x \ is \ y \), even if such a syntax is not valid in Python.

Notes.
1. Recall that Python callables are the following:
   - user defined functions, including lambdas,
   - instance methods,
   - generator functions,
   - built-in functions,
   - built-in methods,
   - classes
   - class instances with \_\_call\_\_ method.

2. It could be enough to consider just Python expressions which evaluate to Python callables, because for any Python operator it is possible to define a function with the same behavior, e.g.
   ```python
def plus(x, y): return x + y
```

3. Among Python expressions which evaluate to Python callables there are function names, lambda expressions and expressions constructed using higher-level functions.

4. Notice that Python operators do not evaluate to Python callables. – to test this try to evaluate in the interactive interpreter \( \text{id(+)} \), etc. and see that it raises SyntaxError.

5. Notice that Python allows defining functions which can be called with varying numbers of parameters, so the number of arguments a Python functional expression can take is not unique.

   In mathematics, a function when it is applied multiple times to the same argument, always produces the same value. Not all Python functional expressions behave in this way: if a Python functional expression involves functions which maintain state in a non-local variable, or access external resources such as user input or system clock, successive calls to this function may return different values – such expressions will be called Python pseudo-functional expressions. The remaining expressions, which behave like proper mathematical functions will be called Python proper-functional expressions.

2.2 Definition By \textbf{Object} we understand the set of all Python objects existing in the memory at one particular moment, including objects of user-defined classes.

2.3 Definition Let \( e \) be a Python proper-functional expression which can take two arguments.

1. By \textbf{the relation determined by} \( f \) we understand the binary relation

\[
e^r = \{ \langle x, y \rangle \in \text{Object} \times \text{Object} : e(x, y) \text{ evaluates to a true-equivalent} \}.
\]
2. Let $A$ be a set of Python objects. We say that $e$ determines a binary relation on $A$ if 
\[ A \times A \subseteq \{ (x, y) \in \text{Object} \times \text{Object} : e(x, y) \text{ does not raise any exception} \}. \]
Python functional expressions with other numbers of arguments are treated in a similar way.

2.4 Definition Let $e$ be a Python proper-functional expression which takes one argument.
1. By the function determined by $e$ we understand the binary relation 
\[ e^f = \{ (x, y) \in \text{Object} \times \text{Object} : e(x) \text{ evaluates to } y \} \]
2. We say that $e$ determines a function from $A$ to $B$, and write 
\[ e^f : A \rightarrow B \] if $e^f$ is a function such that $\text{domain}(e^f) = A$ and $\text{range}(e^f) \subseteq B$.
Python functional expressions with other numbers of arguments are treated in a similar way.

Notice that every Python functional expression $e$ gives rise to both a function $e^f$ and a relation $e^r$ – they are related but different mathematical entities. For instance, $\text{is}$ is a set of ordered pairs from $\text{Object} \times \text{Object}$, while $\text{is}^f$ is a set of ordered triples from $\text{Object} \times \text{Object} \times \text{Boolean}$.

2.5 Convention Given a Python functional expression $e$, instead of $e^r$ we will write “relation $e$” and instead of $e^f$ we will write “function $e$”.

For instance relation $\text{is}$ is a binary relation on $\text{Object}$, and function $\text{is}$ is a function $\text{Object} \times \text{Object} \rightarrow \{ \text{True, False} \}$.

2.6 Proposition Let $A \subseteq \text{Object}$. Let $e$ be a Python proper-functional expression. Let $\sim$ be a binary relation. Then, $e^f$ preserves $\sim$ on $A$ iff $e^r$ preserves $\sim$ on $A$.

2.7 Definition ... Python proper-functional expression preserves $\sim$ on $A$ ...

2.4 Sets of objects

2.8 Definition Consider the set of all objects existing in the memory at one particular moment. Let $A$ be a set of Python objects, $A \subseteq \text{Object}$. We say that $A$ is $\text{==}-\text{comparable}$ and write $A \in \text{Comparable}_==$ if $==$ determines a binary relation on $A$.

2.9 Fact $\text{Comparable}_==$ is a non-empty downward closed subfamily of the Boolean algebra of all subsets of $\text{Object}$:
- $\text{Comparable}_== \subseteq \mathcal{P}(\text{Object})$
- $\emptyset \in \text{Comparable}_==$
- If $A \subseteq B \in \text{Comparable}_==$ then $A \in \text{Comparable}_==$
2.10 Definition Consider the set of all objects existing in the memory at one particular moment. We define

1. $\text{Evaluable} = \text{domain}(\text{eval})$

2. $\text{Representable} = \{x \in \text{Object} : \text{repr}(x) \in \text{Evaluable}\} = \text{repr}(\text{Evaluable})$

3. $\text{Hashable} = \text{domain}(\text{hash})$

Notice that $\text{repr}(\text{Evaluable})$ stands for inverse image of $\text{Evaluable}$ under $\text{repr}$.

2.11 Convention Python proper-functional expressions particularly important for this paper will have the following default meaning when used without being qualified as function or relation:

- $\text{id}$ – function $\text{id}^f : \text{Object} \rightarrow \text{Integer}$,
- $\text{is}$ – relation $\text{is}^r$ – binary relation on $\text{Object}$,
- $\text{==}$ – relation $\text{==}^r$ – binary relation on any $A \in \text{Comparable}$,
- $\text{repr}$ – function $\text{repr}^f$ – unary function to $\text{String}$,
- $\text{eval}$ – function $\text{eval}^f$ – unary function from $\text{Evaluable}$ to $\text{Object}$,
- $\text{hash}$ – function $\text{hash}^f$ – unary function from $\text{Hashable}$ to $\text{Object}$.

By $\text{StdObject}$ we understand the set of all objects which can be constructed using Python built-in types and standard library types. By $\text{UserObject}$ we understand the set of all objects whose construction involves user defined classes.

3 Calculations in mathematics and in programming

We intend to compare a fragment of mathematics to Python. We will concentrate on just a small fragment of mathematical activities, namely that of symbolic calculation (which includes calculations on numbers, however without rounding). Simple cases of equation solving will be covered. Constructing proofs (although central to mathematics) will not have Python counterparts in the mapping we are going to present so this activity will not occur in our considerations.

3.1 Text

A mathematician writes some text. If the text satisfies certain conditions it will be considered as a (correct) calculation, but another mathematician will not call it a (correct) calculation if it contains mathematical errors or if it is outside the scope of mathematics, belonging rather to poetry.
A programmer writes some text. If the text satisfies certain conditions it will be considered as a (correct) computer program (in Python), but another programmer will not call it a (correct) computer program (in Python) if it contains errors which cause that it does not compile, crashes during execution or if it is outside the scope of programming, belonging rather to poetry.

3.2 Literals

Although the mathematical term “constant” immediately associates in our minds with numeric constants such as 12, 2/3, 3 - 4i, π, one needs to realize that constants are used in other domains of mathematics as well. \{2, 1, 3\} and \(\mathbb{Z}\) are constants representing certain sets. The following

\[ f(x) = x + 2 \text{ for } x \in \mathbb{Z} \]

is a constant representing a function. Indeed, in first order logic the term “constant” means a symbol which denotes an item in the domain of discourse. In this paper instead of the term “constant” we will use the term “literal”, borrowing it from programming. One could consider a universe of discourse containing all linear equations with one variable and with rational coefficients; in such a case \(x - 1 = 2x + 3i\) is considered a literal.

In programming languages, the term “literal” refers to an item in the program which represents a fixed value. In Python there are numeric literals such as 12, 3-4i, \texttt{math.pi} and literals denoting sets \{2, 1, 3\}, etc. Python allows to construct fractions and define functions; although \texttt{fractions.Fraction(2,3)} (a call to class constructor and initializer) and \texttt{def f(x): return x+1} (a function definition statement) are not called literals in Python documentation, we will think about them as literals. One could define a class of objects which represent linear equations with one variable and with rational coefficients and treat calls to the constructor/initializer as literals.

3.3 Expressions and assignments

A mathematician writes expressions consisting of literals and operations/functions such as 2 + 3 * 5 - x or \(f(f(x - 1))\). In the case of linear equation, a mathematician may apply to it the operation of negating each side of the equation – this could be viewed as an expression

\texttt{negate-sides}(x - 1 = 2x + 3)

which can be evaluated in mathematician’s calculation to

\[-(x - 1) = -(2x + 3)\]

. The mathematician also can make assignments: let \(x = 0\), let \(y = 1, 2, 3\), let \(z = 2 \ast (f(x) + 1)\). Once a variable has been assigned a value, its every subsequent occurrence can be replaced by this value without changing the meaning of the text. It is desirable that a calculation which starts with a
mathematical expression results in a literal (although, there are cases when it is not possible). If a mathematician intends to write a mathematical expression but makes a syntactical mistake, we are not going to call this text an expression. So, one can view a calculation as an operation that starts with a fragment of text and

- if the text is not a (syntactically correct) expression, announces an error.
- if the text is a syntactically correct expression, calculates a literal representing its value or announces an error (in cases such as division by 0).

We can assume that a calculation that starts with a literal, never announces an error and always produces the same literal.

A programmer writes expressions consisting of literals and operators or function calls such as \(2+3*5-x\) and \(f(f(x-1))\). The programmer also can make assignments: \(x = 0, y = \{1,2,3\}, z = 2*(f(x)+1)\). Once a variable has been assigned a value, Python will use this value whenever that variable is encountered in the code (obeying appropriate rules if the variable is reassigned or if there are different variables with the same name but different scopes). Every (correct) expression in the program can be evaluated to an object in computer's memory – this can be also done by typing the expression at the >>> prompt of the Python inactive interpreter. Once an expression evaluated to an object \(x\) in computer's memory, the literal corresponding to the object can be displayed by typing \texttt{print(repr(x))} – this is true in many but not all cases in the current version of Python.

### 3.4 Several kinds of equality

Equality is a central concept of mathematics. In mathematical texts there are really three main kinds of equality (and we are not concerned here with numerous other equivalence relations).

1. **Identity.** Two mathematical objects identical – they are actually one and the same object; two variables have been assigned the one and the same object. For instance, if in a text you see

   \[
   \text{let } x = f(3) \\
   \text{let } y = x
   \]

   then \(x\) and \(y\) have identical values. There is no standard mathematical notation to express that \(x\) and \(y\) have identical values but in this paper we will write \(x\) is \(y\)

   In Python, you can make assignments: \(x = \ldots y = x\) and test that \(x\) and \(y\) refer to the same object by using the operator \texttt{is} which returns \texttt{True} or \texttt{False}.

2. **Tautomorphism.** Two mathematical expressions have the same shape. For instance, if in a text you see

   \[
   \text{let } x = f(3) 
   \]
let $y = f(3)$

then can claim that $x$ and $y$ are equal because they have been assigned values of expressions which are of the same shape (even if you do not know how to evaluate such an expression.) There is no standard mathematical notation or even name of this relation but in this paper we will write $x \sim y$ to express that $x$ and $y$ are defined by expressions of the same shape and we will call them \textit{tautomorphic}.

In Python there is no built-in construct which would allow to test if two objects in the memory of the computer are "of the same shape" although located at possibly different addresses.

3. \textbf{Domain-specific equality.} Two mathematical objects are equal based on axioms specific to the domain of discourse. For instance, in arithmetic: $x + y = y + x$. What this kind of equality is, depends on the universe of discourse and axioms we assume. For instance, there can be two versions of equality concerning sets and numbers. In one version of naive set theory, we assume that a number and a set are always different. for instance $0 \neq \emptyset$ In ZFC, natural numbers are defined to be specific sets and we have $0 = \emptyset$. In arithmetic of real numbers $0 = 0.0$. In ZFC, natural numbers are represented as finite sets using von-Neumann construction, while real numbers are represented using Dedekind cuts which involve infinite sets, so $0 \neq 0.0$.

In Python the construct $==$ is defined for all objects constructed from built in types, and there are means of extending it to user defined classes. In Python, $0 == 0.0$ evaluates to \texttt{True} however it is possible to define a class \texttt{MyNumber} whose objects store either an integer or a float and which has a version of $==$ such that \texttt{MyNumber(0) == MyNumber(0.0)} evaluates to \texttt{False}.

In this paper we will use $==$ to denote equality based on axioms specific to the domain.

In mathematical texts one uses just one symbol for equality, namely $=$, however as we have seen, there can be different reasons for claiming that $=$ holds. Depending on the reason, we will write $x$ is $y$, or $x \sim y$ or $x == y$. Notice that in mathematics

$x$ is $y$ implies $x \sim y$

$x \sim y$ implies $x == y$

The question how this compares to Python, will be considered later in this paper.

Any version of equality in mathematics must be a congruence with respect to all functions and relations of the domain. We will ask if this holds in Python as well.
3.5 Canonical forms

Canonical forms are commonly used in mathematics. It is typical that an object in the universe of discourse is denoted by more than one literal. If so one of the literals denoting the object is chosen as its “canonical form”. For instance, \( \frac{3}{6}, \frac{1}{2} \) and \( \frac{1}{2} \) are literals denoting the same rational number; the one in lowest terms and with a positive denominator is chosen as the canonical form – in our example: \( \frac{1}{2} \).

While discussing polynomials of one variable one version of canonical form could be \( a_0x^0 + ... + a_nx^n \), and another version of a canonical form could be \( a(x-x_1)...(x-x_n) \) where \( a, x_1, ..., x_n \) in \( \mathbb{C} \).

The transformation to canonical form is required to have the property

1. \( x = \text{nf}(x) \)
   and it is desired that the normal form is unique:

2. \( x = y \) implies \( \text{nf}(x) \sim \text{nf}(y) \)

   These two conditions imply:

3. \( \text{nf}(\text{nf}(x)) \sim \text{nf}(x) \)

   A mathematician during a calculation often normalizes a literal, i.e. converts it to canonical form. Notice that converting a linear equation with one variable and rational coefficients to canonical form is the same as solving the equation (however it is not easy to generalize this example to cases of more complex equations.)

In Python, given an object \( x \) in computer’s memory one can evaluate \( \text{eval(repr(x))} \) which will produce an object equal to \( x \) – this is true in many but not all cases in the current version of Python. One of the questions we consider in this paper is if \( \text{eval(repr(...))} \) has properties expected of transformations to canonical forms.

3.6 Plato’s universe

In mathematics, there is only one number four. It can be denoted by literals of many different shapes:

\( \frac{20}{5}, 4, 100 \) (base 2)

We may choose one of these literals as a canonical form, say 4. If a mathematical text contains several occurrences of the literal 4, all these occurrences denote the same single abstract number four. The same holds for mathematical objects other than numbers.

If a normal form transformation satisfies conditions 1 and 2 above,

Let \( \text{Lit} \) be the set of all literals. Let \( \text{CF} = \text{range(nf)} \) – the set of literals in canonical form.

\( \text{CF} \sim \) is isomorphic to \( \text{Lit/==} \)

Each of \( \text{CF/} \) and \( \text{Lit/==} \) can be taken as the Plato’s universe in which a mathematical object has unique existence.

1 (Optional) Of course it is not always possible to compute such \( x_1,...x_n \).
4  About id, is and is not

4.1  Domain and range of id

i1

Property: \( \text{id} : \text{Object} \rightarrow \text{Integer} \)

True by Python 3.2 official documentation: \( \text{id} \) is implemented as object’s memory address represented as an object of type Integer.

4.2  id vs is

i2

Property: \( x \text{ is } y \iff \text{id}(x) == \text{id}(y) \).

True by Python 3.2 official documentation:

"The is operator compares the identity of two objects; the \text{id()} function returns an integer representing its identity"

"The behavior of the is and is not operators cannot be customized; also they can be applied to any two objects and never raise an exception."

4.3  is is an equivalence on Object

i3

Properties:

refl: \( x \text{ is } x \) for \( x \in \text{Object} \);

symm: \( x \text{ is } y \) implies \( y \text{ is } x \) for \( x, y \in \text{Object} \);

tran: \( x \text{ is } y \) and \( y \text{ is } z \) imply \( x \text{ is } z \) for \( x, y, z \in \text{Object} \).

4.4  is is preserved on A

i4

Property: Python proper-functional expressions preserve is for objects in Object.

True?

Note this implies \( x_1 \text{ is } y_1 \) and \( x_2 \text{ is } y_2 \) and \( x_1 = x_2 \) implies \( y_1 = y_2 \).

4.5  is not vs. is

Property: \( x \text{ is not } y \iff \text{not} (x \text{ is } y) \) for \( x, y \in \text{Object} \).

True
5 About == and !=

5.1 Run-time evaluation of ==

\[ =_0 \]

Property: == is always evaluated at the runtime.

True?

Note. This property implies that for any two expressions E1 and E2, typing in the interactive interpreter \( E1 == E2 \) produces the same truth value or raises the same exception as typing

```
>>> x1 = E1
>>> x2 = E2
>>> x == y
```

It is conceivable that in some (poor) implementation of a programming language this property is not satisfied, for instance floating point (exponential notation) literals \( 1e50 \) and \( 2e500 \) both are represented as the same floating point number object as 0.0 but they can be distinguished at the compilation time.

```
>>> 1e500 == 2e500
False
>>> x = 1e500
>>> y = 2e500
>>> x == y
True
```

5.2 \( A \) is ==-comparable

\[ =_1 \]

Property: == determines a binary relation on \( A \).

Property restated: \( A \in \text{Comparable}_{==} \).

True? for \( A \subseteq \text{StdObject} \)

False if \( A \) contains objects of user defined classes poorly defined: If a programmer defines a class with a method \( \_\_eq\_\_ \) that raises an exception when applied to \text{self} and an object of the same class, the set of objects of this class will not be ==-comparable.

Note Let class \( C \) is a direct subclass of 'object'.

1. If \( C \) defines neither \( \_\_eq\_\_ \) nor \( \_\_ne\_\_ \), == coincides with \text{is} and != coincides with \text{not}.
2. If \( C \) defines \( \_\_eq\_\_ \) but not \( \_\_ne\_\_ \), \( x ! = y \) coincides with \text{not} \( (x == y) \)
3. If \( C \) defines \( \_\_ne\_\_ \) but not \( \_\_eq\_\_ \), == coincides with \text{is}.
5.3 == is an equivalence on $A$

$=_{2} =_{3} =_{4}$

Properties:
- reflexivity: $x == x$ for $x \in A$
- symmetry: $x == y$ implies $y == x$ for $x, y \in A$.
- transitivity: $x == y, y == z$ imply $x == z$ for $x, y \in A$.

True? for $A \subseteq \text{StdObject}$

False if $A$ contains objects of user defined classes poorly defined: A programmer who is not careful can define a class whose objects support == but it is not an equivalence, violating even the most basic condition – that of reflexivity.

```python
>>> class MyObject():
...     def __eq__(self, y):
...         return False
...     
... >>> a = MyObject()
... >>> a == a
False
```

5.4 Preserving == on $A$

$=_{5}$

Property: Python proper-functional expressions preserve == for objects in $A$.

False for Python proper-functional expressions involving constructs `type`, `isinstance` which can distinguish between equal (==) number objects such as 1 and 1.0 or between True and 1.

False for Python proper-functional expressions whose definitions use `id` to distinguish between equal (==) objects such as a container object and its copy.

True? if `id`, `type`, `isinstance` are excluded

True? if `id` is excluded, number objects other than integers are excluded (leaving only one number type).

NOTE. One could define in the implementation of the language a new operator === such that 1 === 1.0 evaluates to False Both == and === would be a applicable to arbitrary object and they would compare recursively their components. For this new operator a counterparts of properties $=_{1}$ through $=_{5}$ would be True? in cases when just `id` is excluded and non-local variables are not used. Notice that defining `eq` as

```python
def eq(x, y): return type(x) == type(y) and x == y
```

is not correct: although `eq(1, 1.0)` returns False `eq([1], [1.0])` returns True
5.5  != is the negation of ==

Property: x != y is equivalent to not (x == y) for x,y in A.

True? for A ⊆ StdObject
False for A ⊆ UserObject – if A contains objects of user defined classes, poorly defined.

5.6  is vs. ==

i5

Property: x is y implies x == y for x,y ∈ A for any A ∈ Comparable

True? if A ⊆ StdObject
False for objects of user defined classes, if poorly defined. It is possible to define a class such that x == y always evaluates to to False.

Note: but not vice versa:

```python
>>> x = [1,2]
>>> y == copy.copy(x)
>>> x is not copy.copy(x)
True
```

Note. If f is a function f == f gives True.

Note. A property stronger than i2: x is y iff id(x) is id(y).

is False.

5.7  == on Integer

⟨Integer, <, <=, >, >=, ==, !=, +, -, *, 0, 1⟩/== ≅ ⟨Z, <, <=, >, >=, +, -, *, 0, 1⟩

6  About repr and eval

repr(123) == ’123’
repr(’abc’) == ”’abc’”
repr(True) = ’True’
repr(None) = ’None’

6.1  Range of repr

Property: range(repr) ⊆ String

True
6.2 Domain of \texttt{repr} \\
\texttt{r0} \\
\textbf{Property:} $A \subseteq \text{domain} (\texttt{repr})$ \\
\textbf{True?} for built-in objects \\
\textbf{Quote:} "\texttt{object.__repr__(self)} Called by the \texttt{repr()} built-in function to compute the official string representation of an object. If at all possible, this should look like a valid Python expression that could be used to recreate an object with the same value (given an appropriate environment). If this is not possible, a string of the form \texttt{<...some useful description...>} should be returned. The return value must be a string object. This is typically used for debugging, so it is important that the representation is information-rich and unambiguous." Python 3.1 docs. \\
\textbf{Warning.} A programmer who is not careful can define a class whose objects do not support \texttt{repr}.

6.3 \texttt{repr} preserves $==$ on $A$ \\
\textbf{Property:} $x == y$ imply $\texttt{repr}(x) == \texttt{repr}(y)$, for $x, y \in A$ for any $A \in \text{Comparable}_==$. \\
\textbf{True?} \\
\textbf{Question:} do equal dictionaries have equal representation (in the same order)? \\
\textbf{Note:} $=_5$ implies r1. \\
\textbf{Warning.} A programmer who is not careful can define a class whose objects do support \texttt{repr} but violate this property.

6.4 \texttt{repr} is one-to-one \\
\texttt{r2} \\
\textbf{Property:} $\texttt{repr}(x) == \texttt{repr}(y)$ implies $x == y$, for $x, y \in \text{Comparable}_==$. \\
\textbf{True?} \\
\textbf{Warning.} A programmer who is not careful can define a class whose objects do support \texttt{repr} but violate this property.

\textbf{Quote:} \texttt{eval(source [...]) -> value} Evaluate the source in the context [...]. The source may be a string representing a Python expression or a code object as returned by \texttt{compile()}.” From \texttt{help(eval)}. \\
\textbf{Assumption:} In this paper we consider calls to Python’s \texttt{eval} with an argument of type \texttt{<type 'str'>} and without optional arguments, so that they determine a one-argument function on \texttt{String}. \\

6.5 domain of eval

Property: range(repr |_A_) ⊆ domain(eval)

Property restated: repr ∈ domain(eval) for x ∈ A.

False, if x is a Python function, because repr(x) ∉ domain(eval).

Question: What is Python’s rationale for handling functions this way?

Recall definition: Evaluable = domain(eval).

Note. Evaluable ⊆ String.

Property restated: range(repr) ⊆ Evaluable.

Note. range(repr) = Evaluable would be too strong a condition: we want for dictionaries that
\[ \text{eval("\{a':1, b':2\}"}) = \text{eval\{\{b':2, a':1\}\}} \]
but this condition would imply that only one of the strings
"{a':1, b':2}", "{b':2, a':1}" is in Evaluable.

Recall definition Representable = \{ x : repr(x) ∈ Evaluable \}

Note. LTIS is a subset of Evaluable.

Property restated: Representable = Object.

Warning. A programmer who is not careful can define even a simple container class whose objects do support repr but violate this property.

6.6 eval preserves ==

Property: r₁ == r₂ implies eval(r₁) == eval(r₂) for r₁, r₂ ∈ Evaluable.

True?

Note: =5 implies e3.

6.7 eval is one-to-one

Property: eval(r₁) == eval(r₂) implies r₁ == r₂ for r₁, r₂ ∈ Evaluable.

False eval("(1,2)")) == eval("(1 , 2)"
also eval("{a':1,'b':2}")) == eval("{b':2,'a':1}"
6.8 eval vs. repr

e5

Property: eval(repr(x)) == x evaluates to True for x ∈ Representable.

True?

Quotes "For most object types, eval(repr(object)) == object." (Python 3.1.1 help({\tt repr})) "The numbers 0.1 and 0.10000000000000001 and 0.100000000000000005511151231257827021181583404541015625 are all approximated by 3602879701896397 / 2 ** 55. Since all of these decimal values share the same approximation, any one of them could be displayed while still preserving the invariant eval(repr(x)) == x." (Python 3.1.1 Tutorial, ch. 14)

Warning. A programmer who is not careful can define a class whose objects do support repr but violate this property.

6.1 Fact

A weaker version of r2: repr(x) == repr(y) evaluates to True implies x == y\verb, for x,y ∈ Representable is implied by e3, e5.

Proof

1. Assume repr(x) == repr(y)\verb.
2. Assume x,y \in \name{Evaluable}.
3. So, repr(x),repr(y) ∈ Evaluable, by 2.
4. So, eval(repr(x)) == {eval(repr(y)) evaluates to True by 1, 3, e3.
5. So, x == y evaluates to True by 4, e5.

6.2 Corollary

r2 is implied by e2, e3, e5.

Proof

Use the fact above and notice that e2 can be stated as Representable = Object.

NOTE I asked myself if it would be interesting to define a function srepr (strong-repr) with properties analogous to those enjoyed by {\tt repr} but such that: eval(srepr(x)) is x evaluates to True. Such a srepr would be somewhat as id. The idea not practical. It is feasible to define a function given which given object’s id returns the object. It would have the property: given(id(x)) is x evaluates to True However if you store the value returned by id(x) and then x gets garbage collected, given will not recreate it.

7 About hash

7.1 Range of hash

h1
Property: hash is a function to Integer.

True

Definition Hashable = domain(hash)

Note Class Hashable in module "collections" is different.

Property restated. hash : Hashable → Integer

Note. Intuition: x in Hashable iff x is unchangeable (immutable object with immutable components at all levels.) TIS objects are hashable, LTIS are not.

Warning. A programmer who is not careful can define a class whose objects are unchangeable but do not support hash.

7.2 hash preserves ==

h2

Property: x == y evaluates to True implies hash(x) == hash(y) evaluates to True for x, y ∈ Hashable ∩ Comparable

True

Note: =_5 implies h1.

Quotes "Two objects with the same value have the same hash value." (help(hash))
"Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0). (Python 3.1.1 Std Library reference: Functions)"

Warning. A programmer who is not careful can define a class whose objects support hash but violate this property.

7.3 hash is one-to-one

h3

Property: hash(x) == hash(y) evaluates to True implies x == y evaluates to True for x, y ∈ Hashable ∩ Comparable

False?

Quote. "not necessarily true, but likely." (Python 3.1 help(hash))

Note: I tried to produce an argument for falsity without knowing the actual hash algorithm but I failed: Hash seems to produce integers in a finite range; if so it cannot give distinct values to infinitely many objects; however at any time there are only finitely many objects in the memory."
Important question: could we have a function prehash with properties h1-h3 and then define hash in terms of pre-hash so that it has a finite range?

8 Appendix: Quotes from Python documentation

8.1 About id and is

8.1 Quote "id(object) 
Return the 'identity' of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same id() value." (The Python Standard Library, 2. Built-in Functions.)

8.2 Quote "Objects, values and types 
Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann’s model of a 'stored program computer', code is also represented by objects.)

Every object has an identity, a type and a value. An object’s identity never changes once it has been created; you may think of it as the object address in memory. The is operator compares the identity of two objects; the id() function returns an integer representing its identity (currently implemented as its address). An object type is also unchangeable. An object type determines the operations that the object supports (e.g., 'does it have a length?') and also defines the possible values for object of that type. The type() function returns an object type (which is an object itself). The value of some objects can change. Objects whose value can change are said to be mutable; Objects whose value is unchangeable once they are created are called immutable. (The value of an immutable container object that contains a reference to a mutable objects can change when the latter’s value is changed; however the container is still considered immutable, because the collection of objects it contains cannot be changed. So, immutability is not strictly the same as having an unchangeable value, it is more subtle.) An object’s mutability is determined by its type; for instance, numbers, strings and tuples are immutable, while dictionaries and lists are mutable.

Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected. An implementation is allowed to postpone garbage collection or omit it altogether it is a matter of implementation quality how garbage collection is implemented, as long as no objects are collected that are still reachable." (The Python Language Reference, 3 Data Model).

Note: It is possible to implement id in such a way that no two objects with non-overlapping life-times can have the same id. This can be done, for instance by storing within every object a time-stamp and having id return the object’s address and the time-stamp. However it is not clear what we would gained by such an approach.
8.2 About == and !=

8.3 Quote “==
compare the values of two objects. The objects need not have the same type.
If both are numbers, they are converted to a common type. Otherwise, the
== and != operators always consider objects of different types to be unequal.”
Python 3.1.1. Reference.

8.4 Quote “some types (for example, function objects) support only a degenerate
notion of comparison where any two [non-identical] objects of that type
are unequal.”

8.5 Quote “Non-identical instances of a class normally compare as non-equal
unless the class defines the _eq_() method.”

8.6 Quote “x==y calls x._eq_(y)
There are no swapped-argument versions of these methods (to be used when
the left argument does not support the operation but the right argument
does.)” (Python Reference.)

8.7 Quote ”object._eq_(self, other)
object._ne_(self, other))
These are the so-called ‘rich comparison’ methods. The correspondence
between operator symbols and method names is as follows: x==y calls
x._eq_(y), x!=y calls x._ne_(y). A rich comparison method may
return the singleton NotImplemented if it does not implement the operation
for a given pair of arguments. By convention, False and True are
returning for a successful comparison. However, these methods can return
any value, so if the comparison operator is used in a Boolean context (e.g.,
in the condition of an if statement), Python will call bool() on the value
to determine if the result is true or false. There are no implied relations-
ships among the comparison operators. The truth of x==y does not imply
that x!=y is false. Accordingly, when defining _eq_(), one should also
define _ne_() so that the operators will behave as expected. See the para-
graph on __hash__() for some important notes on creating hashable objects
which support custom comparison operations and are usable as dictionary
keys. There are no swapped-argument versions of these methods (to be
used when the left argument does not support the operation but the right
argument does); rather, [..] and _eq_() and _ne_()\verb are their
own reflection. Arguments to rich comparison methods are never coerced.”
(Python v3.1.2 documentation, The Python Language Reference, 3. Data
model)

8.3 About repr and eval

8.8 Quote ”repr(object) Return a string containing a printable representa-
tion of an object. For many types, this function makes an attempt to return
a string that would yield an object with the same value when passed to
eval(), otherwise the representation is a string enclosed in angle brackets
that contains the name of the type of the object together with additional information often including the name and address of the object. A class can control what this function returns for its instances by defining a `__repr__()` method.

8.9 Quote "`eval(expression, globals=None, locals=None)` The arguments are a string and optional globals and locals. [...] The expression argument is parsed and evaluated as a Python expression (technically speaking, a condition list). [...] The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:

```python
>>> x = 1
>>> eval('x+1')
2
```

This function can also be used to execute arbitrary code object (such as those created by `compile()`). In this case pass a code object instead of a string. If the code object has been compiled with 'exec' as the kind argument, `eval()`'s return value will be `None`.

8.10 Quote "there are many different decimal numbers that share the same nearest approximate binary fraction. For example, the numbers 0.1 and 0.10000000000000001 and 0.10000000000000000000000555111512312578270218158340541015625 are all approximated by 3602879701896397 / 2 ** 55. Since all of these decimal values share the same approximation, any one of them could be displayed while still preserving the invariant `eval(repr(x)) == x`.

Historically, the Python prompt and built-in `repr()` function would chose the one with 17 significant digits, 0.10000000000000001. Starting with Python 3.1, Python (on most systems) is now able to choose the shortest of these and simply display 0.1. "(Python v3.1.2 documentation, The Python Tutorial, 14. Floating Point Arithmetic: Issues and Limitations: http://docs.python.org/py3k/tutorial/floatingpoint.html)

9 References

Python v3.1.2 documentation (2010),
  The Python Language Reference,
  3 Data model
  5. Expressions
  5.14 Summary (operators)
The Python Standard Library,
  2. Built-in Functions
  3. Built-in Constants
  8. Data Types
  8.3. Collections – Container datatypes
  8.3.1. ABCs - abstract base classes
  The Python Tutorial
  14. Floating Point Arithmetic: Issues and Limitations
>>> 10**10 is 10*10
False

>>> 10000000000 is 10000000000
True

>>> 1**1 is 1**1
True

-----------------------------

>>> 1.0 == True
True

-----------------------------

>>> 0.0 == 1e-323
False

>>> 0.0 == 1e-324
True

-----------------------------