Python Top 10:

1. Do not use floating-point arithmetic when integers or booleans would suffice.
2. Use of enumeration requires careful attention to readability, performance, and safety. There are many complex, but useful ways to simulate enums in Python [(Enums for Python (Python recipe))]and many simple ways including the use of sets:

   ```
   colors = {'red', 'green', 'blue'}
   if  red  in colors: print('valid color')
   ```

   Be aware that the technique shown above, as with almost all other ways to simulate enums, is not safe since the variable can be bound to another object at any time.
3. Ensure that when examining code that you take into account that a variable can be bound (or rebound) to another object (of same or different type) at any time.
4. Avoid implicit references to global values from within functions to make code clearer. In order to update globals within a function or class, place the global statement at the beginning of the function definition and list the variables so it is clearer to the reader which variables are local and which are global (for example, global a, b, c).
5. Use only spaces or tabs, not both, to indent to demark control flow. Never use form feed characters for indentation.
6. Use Python's built-in documentation (such as docstrings) to obtain information about a class' method before inheriting from it.
7. If coding an extension utilize Python's extension API to ensure a correct signature match.
8. Either avoid logic that depends on byte order or use the sys.byteorder variable and write the logic to account for byte order dependent on its value ('little' or 'big').
9. When launching parallel tasks don't raise a BaseException subclass in a callable in the Future class.
10. Do not depend on the way Python may or may not optimize object references for small integer and string objects because it may vary for environments or even for releases in the same environment.

For reference, here is all of the Python specific guidance from 24773 (n0461):

• Pay special attention to issues of magnitude and precision when using mixed type expressions;
• Be aware of the consequences of shared references;
• Be aware of the conversion from simple to complex;
• Do not check for specific types of objects unless there is good justification, for example, when calling an extension that requires a specific type.
• Keep in mind that using a very large integer will have a negative effect on performance;
• Don't use bit operations to simulate multiplication and division.
• Use floating-point arithmetic only when absolutely needed;
• Do not use floating-point arithmetic when integers or booleans would suffice;

• Be aware that precision is lost for some real numbers (that is, floating=point is an approximation with limited precision for some numbers);
• Be aware that results will frequently vary slightly by implementation (see 6.50 (Provision of Inherently Unsafe Operations [SKL] for more on this subject);
• Testing floating-point numbers for equality (especially for loops) can lead to unexpected results. Instead, if floating-point numbers are needed for loop control use >= or <= comparisons.
• Use of enumeration requires careful attention to readability, performance, and safety. There are many complex, but useful ways to simulate enums in Python [(Enums for Python (Python recipe))]and many simple ways including the use of sets:

```
colors = {'red', 'green', 'blue'}
if  red  in colors: print('valid color')
```

Be aware that the technique shown above, as with almost all other ways to simulate enums, is not safe since the variable can be bound to another object at any time.
• Though there is generally no need to be concerned with an integer getting too large (rollover) or small, be aware that iterating or performing arithmetic with very large positive or small (negative) integers will hurt performance;
• Be aware of the potential consequences of precision loss when converting from floating point to integer.
• Be cognizant that most arithmetic and bit manipulation operations on non-integers have the potential for undetected wrap=around errors.
• Avoid using floating point or decimal variables for loop control but if you must use these types then bound the loop structures so as to not exceed the maximum or minimum possible values for the loop control variables.
• Test the implementation that you are using to see if exceptions are raised for floating point operations and if they are then use exception handling to catch and handle wrap-around errors.
• For more guidance on Python's naming conventions, refer to Python Style Guides contained in PEP 8 at http://www.python.org/dev/peps/pep=0008/ .
• Avoid names that differ only by case unless necessary to the logic of the usage;
• Adhere to Python's naming conventions;
• Do not use overly long names;
• Use names that are not similar (especially in the use of upper and lower case) to other names;
• Use meaningful names;
• Use names that are clear and visually unambiguous because the compiler cannot assist in detecting names that appear similar but are different.
• Avoid rebinding except where it adds value;
• Ensure that when examining code that you take into account that a variable can be bound (or rebound) to another object (of same or different type) at any time;
• Variables local to a function are deleted automatically when the encompassing function is exited but, though not a common practice, you can also explicitly delete variables using the del statement when they are no longer needed.
• Do not use identical names unless necessary to reference the correct object;

• Avoid the use of the global and nonlocal specifications because they are generally a bad programming practice for reasons beyond the scope of this annex and because their bypassing of standard scoping rules make the code harder to understand;
• Use qualification when necessary to ensure that the correct variable is referenced.
• When practicable, consider using the import statement without the *from* clause. This forces the importing program to use qualification to access the imported module's attributes. While it is true that using the from statement is more convenient due to less typing required (for example, no need to qualify names), the from statement can cause namespace corruption;
• When using the import statement, rather than use the from X import * form (which imports all of module X's attributes into the importing program's namespace), instead explicitly name the attributes that you want to import (for example, from X import a, b, c) so that variables, functions and classes are not inadvertently overlaid;
• Avoid implicit references to global values from within functions to make code clearer. In order to update globals within a function or class, place the global statement at the beginning of the function definition and list the variables so it is clearer to the reader which variables are local and which are global (for example, global a, b, c).
• Ensure that it is not logically possible to reach a reference to a variable before it is assigned. The example above illustrates just such a case where the programmer wants to print the value of x but has not assigned a value to x this proves that there is missing, or bypassed, code needed to provide x with a meaningful value at runtime.
• Use parenthesis liberally to force intended precedence and increase readability;
• Be aware that short-circuited expressions can cause subtle errors because not all sub-expressions may be evaluated;
• Break large/complex statements into smaller ones using temporary variables for interim results.
• Be aware of Python's short-circuiting behaviour when expressions with side effects are used on the right side of a Boolean expression; if necessary perform each expression first and then evaluate the results:

```
x = a()
y = b()
if x or y …
```

• Be aware that, even though overlaps between the left hand side and the right hand side are safe, it is possible to have unintended results when the variables on the left side overlap with one another so always ensure that the assignments and left=to=right sequence of assignments to the variables on the left hand side never overlap. If necessary, and/or if it makes the code easier to understand, consider breaking the statement into two or more statements;

```
# overlapping
a = [0,0]
i = 0
i, a[i] = 1, 2 #=> Index is set to 1; list is updated at [1]
print(a) #=> 0,2
# Non-overlapping
a = [0,0]
```

```
i, a[0] = 1, 2
print(a) #=> 2,0
```

• Be sure to add parentheses after a function call in order to invoke the function;

• Keep in mind that any function that changes a mutable object in place returns a None object – not the changed object since there is no need to return an object because the object has been changed by the function.

• Import just the attributes that are required by using the from statement to avoid adding dead code;

• Be aware that subsequent imports have no effect; use the reload statement instead if a fresh copy of the module is desired.

• Use if/elif/else statements to provide the equivalent of switch statements.

• Use only spaces or tabs, not both, to indent to demark control flow.

• Be careful to only modify loop control variables in ways that are easily understood and in ways that cannot lead to a premature exit or an endless loop.

• When using the for statement to iterate through a mutable object, do not add or delete members because it could have unexpected results.

• Be aware of Python's indexing from zero and code accordingly.

• Python offers few constructs that could lead to unstructured code. However, judicious use of break statements is encouraged to avoid confusion.

• Use Python's exception handling with care in order to not catch errors that are intended for other exception handlers;

• Use exception handling, but directed to specific tolerable exceptions, to ensure that crucial processes can continue to run even after certain exceptions are raised.

• Use Python's exception handling statements to implement an appropriate termination strategy.

• Release all objects when they are no longer required.

• Inherit only from trusted classes;

• Use Python's built-in documentation (such as docstrings) to obtain information about a class' method before inheriting from it.

• Do not override built-in intrinsics unless absolutely necessary

• Use the language interface APIs documented on the Python web site for interfacing to C/C++, the Jython web site for Java, the IronPython web site for .NET languages, and for all other languages consider creating intermediary C or C++ modules to call functions in the other languages since many languages have documented API's to C and C++.

• Avoid using exec or eval and never use these with untrusted code;

• Be careful when using Guerrilla patching to ensure that all users of the patched classes and/or modules continue to function as expected; conversely, be aware of any code that patches classes and/or modules that your code is using to avoid unexpected results;

• Ensure that the file path and files being imported are from trusted sources.

• Use only trusted modules as extensions;

• If coding an extension utilize Python's extension API to ensure a correct signature match.

• Wrap calls to library routines and use exception handling logic to intercept and handle exceptions when practicable.

• Use only trusted modules;
• Avoid the use of the exec and eval functions.
• Be aware of when a variable is local versus global;
• Do not use mutable objects as default values for arguments in a function definition unless you absolutely need to and you understand the effect;
• Be aware that when using the += operator on mutable objects the operation is done in place;
• Be cognizant that assignments to objects, mutable and immutable, always create a new object;
• Understand the difference between equivalence and equality and code accordingly;
• Ensure that the file path used to locate a persisted file or DBMS is correct and never ingest objects from an untrusted source.
• Do not rely on the content of error messages – use exception objects instead;
• When persisting object using pickling use exception handling to cleanup partially written files;
• Do not depend on the way Python may or may not optimize object references for small integer and string objects because it may vary for environments or even for releases in the same environment.
• Understand the difference between testing for equivalence (for example, ==) and equality (for example, is) and never depend on object identity tests to pass or fail when the variables reference immutable objects;
• Do not depend on the sequence of keys in a dictionary to be consistent across implementations.
• When launching parallel tasks don't raise a BaseException subclass in a callable in the Future class;
• Never modify the dictionary object returned by a vars call;
• Never use form feed characters for indentation;
• Consider using the id function to test for object equality;
• Do not try to use the catch_warnings function to suppress warning messages when using more than one thread;
• Never inspect or change the content of a list when sorting a list using the sort() method.
• Always use either spaces or tabs (but not both) for indentations;
• Consider using the -tt command line option to raise an IndentationError;
• Consider using a text editor to find and make consistent, the use of tabs and spaces for indentation;
• Either avoid logic that depends on byte order or use the sys.byteorder variable and write the logic to account for byte order dependent on its value ('little' or 'big').
• Use zero (the default exit code for Python) for successful execution and consider adding logic to vary the exit code according to the platform as obtained from sys.platform (such as, 'win32', 'darwin', or other).
• Interrogate the sys.float.info system variable to obtain platform specific attributes and code according to those constraints.
• Call the sys.getfilesystemcoding() function to return the name of the encoding system used.
• When high performance is dependent on knowing the range of integer numbers that can be used without degrading performance use the sys.int_info struct sequence to obtain the number of bits per digit (bits_per_digit) and the number of bytes used to represent a digit (sizeof_digit).
• When practicable, migrate Python programs to the current standard.