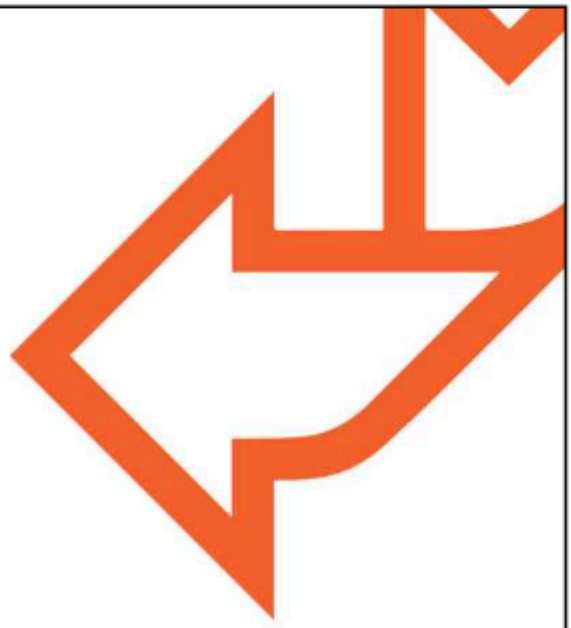
Python 3 Programming

Appendix

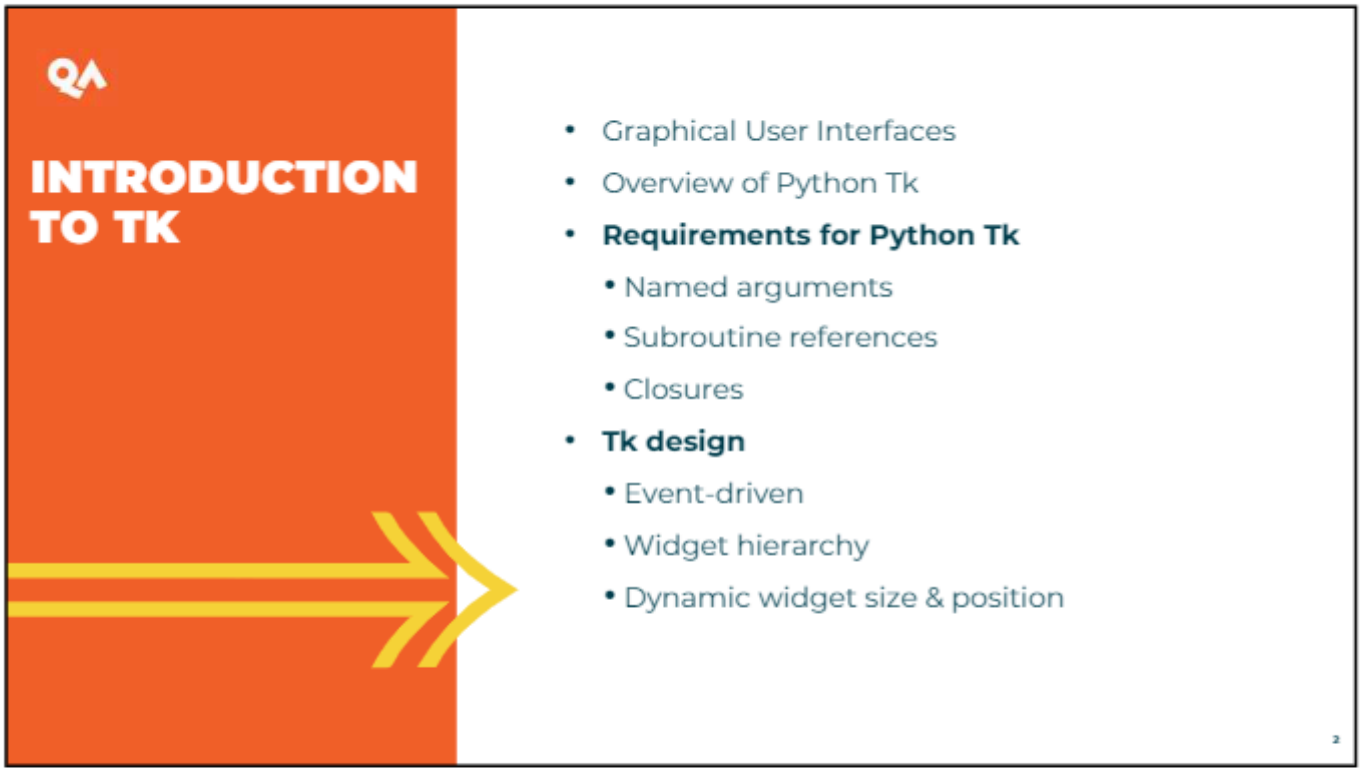
QA.COM



Python 3 Programming

Appendix

Introduction to Tk



A brief introduction to the tkinter modules.

Graphical user interfaces from Python

- **Python supports various GUI extensions**
  - X11
  - Win32
  - Macintosh
  - Gtk (X specific)
  - Tk
  - Qt
- **Tk is most commonly used**
  - Tkinter on Python 2, tkinter on Python 3
  - Portable across UNIX and Windows
  - Based on tcl/Tk toolkit
  - Object-oriented interface

Python supports a number of GUI-extensions, most of which are tied to a specific windowing system (such as X11, Win32 or Macintosh). The Gtk extension is from the Gimp Toolkit, which also runs on Windows.

The Tk extension is most commonly used, because it is portable across platforms, powerful and easy to use. It is based on the Tk toolkit from the tcl language, developed by John Ousterhout (first at Berkeley, then at Sun and now at Scriptics). The same Tk toolkit has also been ported to Perl, Scheme, and Guile.

The main difference between Python Tk and tcl/Tk is that the perl version is fully object-oriented, whereas the tcl version is string based (and generally a mess).

A simple example

- **Create objects**
  - Main window
  - Labels and buttons
  - Define call back
  - Invoke main loop


```
from tkinter import *

def button_proc():
    print('Call-back function for button')

root = Tk()
root.title('Tk Example')
Label(root, text='Read Me').pack()
Button(root, text='Click Me',
       command=button_proc).pack()
root.mainloop()
```

elements.py

The example above starts by including the Tk module; this is required for every perl/Tk program. It then creates a new MainWindow object. Evidently, Tk plays a few tricks here; the Tk module has defined a MainWindow class as well.

Given a main window, a label and a button are defined. By creating these from the main window, Tk knows how the widgets have to be nested. Evident is that Tk uses a named-argument calling convention. The pack() calls are necessary to handle widget layout and are described in more detail later.

Finally, the main loop is invoked. Tk now takes over and handles all keyboard, mouse and window events. Whenever something noteworthy happens, user-defined call-back routines are invoked that handle an event. This is an example of event-driven programming.

Elements of Python Tkinter

- **Event-driven**
  - Create objects, then run a main loop
  - Main loop (indirectly) invokes callback subroutines
- **Methods use named arguments**
  - Default values exist for most arguments
- **Events invoke callback subroutines**
  - Reference to subroutine
  - Anonymous subroutine
  - Closure
  - Object + method name

Tk is like most other graphical environments in that you have to give up control. ("Don't call us – we'll call you").

So, instead of tracking all events and deciding what action to take, you set up the graphical environment and start the main loop, which will then handle all default actions and invoke your code when necessary to take an action.

User input is hidden

- **Data entry objects handle input**
  - Stored in a StringVar, IntVar variable
  - No need to query state


```
from tkinter import *

def button_proc():
    print('Call-back function, printer is now', ent.get())

root = Tk()
root.title('Printer Options')

but = Button(root, text='Done', command=button_proc)
but.pack(side = 'bottom')
Label(text='Printer Name').pack(side='left')

printer = StringVar()
ent=Entry(root, textvariable=printer)
ent.pack(side='left')

printer.set('SAP64')
root.mainloop()
```

printer.py

One of the nicest features of Tk is that, unlike many other GUI toolkits, there is no need to query the user interface elements for the state of options. Instead, communication is done through a reference – the GUI sets a variable which can be read in the call-back subroutines.

This communication is two-way: if a call-back routine modifies the variable, the GUI will change its display accordingly.

Widgets are hierarchical

Widgets inside widgets inside...

- Normally indicated through order of creation and choice of parent.

```
from tkinter import *

root = Tk()

topf = Frame(root).pack()
botf = Frame(root).pack()

Label(topf, text ='One').pack(side='left')
Label(topf, text ='Two').pack(side='left')

Label(botf, text='Three').pack(side='left')
Label(botf, text='Four').pack(side='left')

root.mainloop()
```

hierarchy.py

7

In Tk, choosing your parent wisely is even more important than in real life! In the example above, the main window contains two Frames (invisible place holders) which in turn contain buttons. This kind of nesting is typical for Tk – instead of using a fixed widget setup with absolute positioning, the nesting order creates a flexible and dynamic layout.

This idea is also used in Java, where the AWT uses both nested widgets and so-called layout managers to determine the relative layout of the user interface.

QA Widgets are dynamic

Widgets may be added or removed while running.

- All widgets dynamically sized.

```
from tkinter import *

root = Tk()
button = Button(root)
button.pack()
label  = Label(root, text="I'm here")

def hide_label():
    label.forget();
    button.configure(text="Show Label",
                     command=show_label)

def show_label():
    label.pack()
    button.configure(text="Hide Label",
                     command=hide_label)

show_label()
root.mainloop()
```

Initial size is computed and resized when window size changes

dynamic.py

The example above is slightly unusual in that we hold on to the widgets using global variables and use those inside the call-back routines. As we'll see in the next chapters, you can use either closures or named widgets and dynamic enquiry functions to achieve the same in a neater fashion.

The code above creates the Label widget once and dynamically displays or hides it, using pack and packForget. The Button widget is always there but is re-configured dynamically to change its text and callback functions.

Building a main window

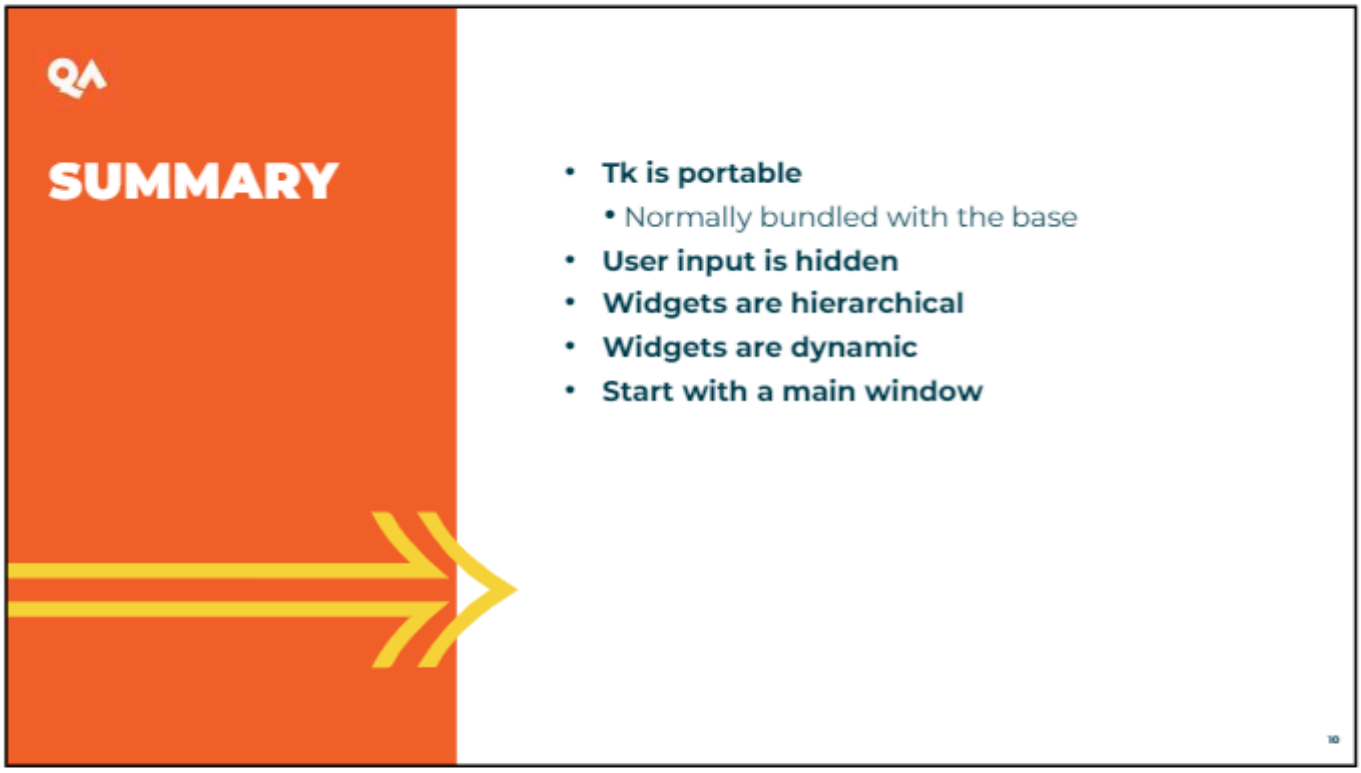
- A main window also requires title, icon, and menus.

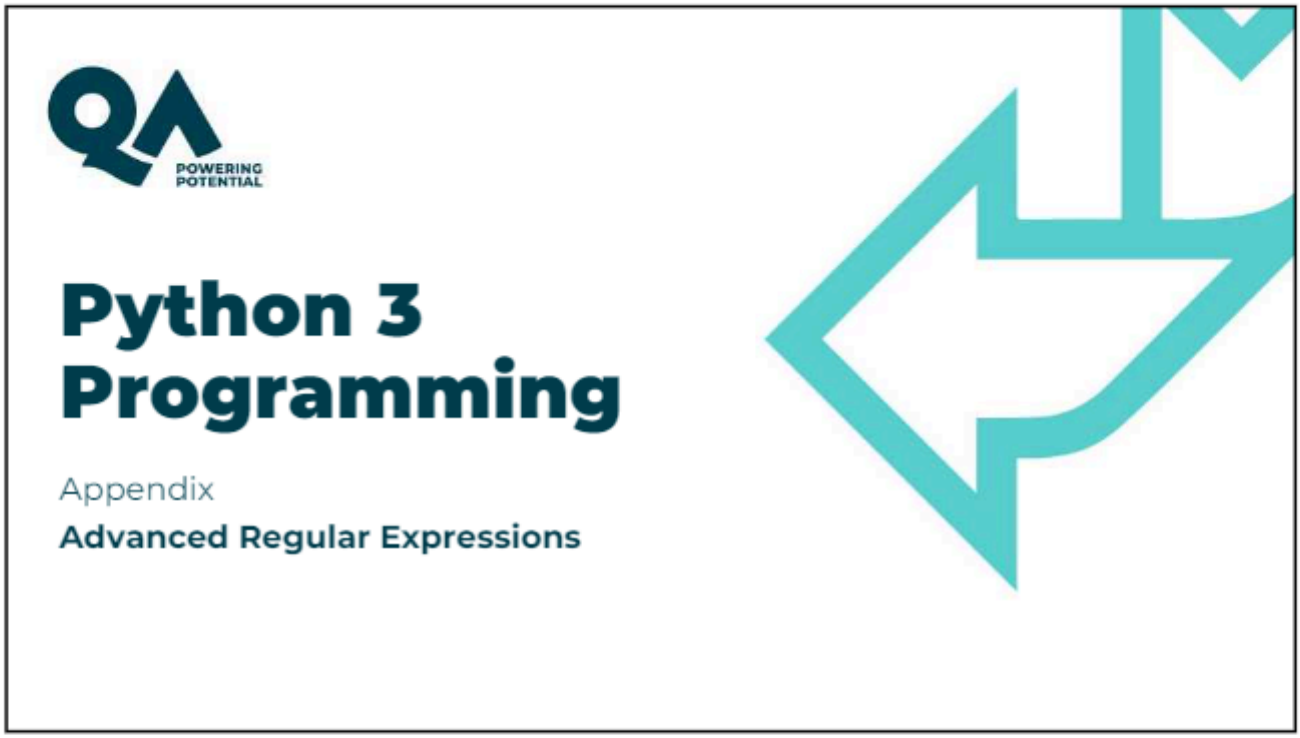
```
from tkinter import *
root = Tk()
root.title('Window Title')
root.wm_iconbitmap('qa.ico')

mbar = Menu(root)
filemenu = Menu(mbar, tearoff=0)
filemenu.add_command(label='Quit',
                     command=lambda: root.destroy())
helpmenu = Menu(mbar, tearoff=0)
helpmenu.add_command(label='RTFM')

mbar.add_cascade(label='File', menu=filemenu)
mbar.add_cascade(label='Help', menu=helpmenu)
root.config(menu=mbar)

bt = Button(root, text ='Click Me!', command=button_proc)
bt.pack()

root.mainloop()
```

mainwindow.py

A main window should always have a title and an icon. The icon bitmap must either be one of the pre-defined bitmaps or can start with an @ to indicate a filename of an xpm file.

In addition, most main windows should have a menu with a 'File – Quit' entry.


SUMMARY
• Tk is portable
• Normally bundled with the base
• User input is hidden
• Widgets are hierarchical
• Widgets are dynamic
• Start with a main window
10


POWERING
POTENTIAL
Python 3
Programming
Appendix
Advanced Regular Expressions


ADVANCED
REGULAR
EXPRESSIONS
Review
Regular expression quoting
Modifiers
Side effect variables and capturing
Minimal matches
Multi-line matching
Global matches
Look-around assertions
Substitution with interpolation
Substitution with expressions
2

QA Elementary extended RE meta-characters

`.`	match any single character	Character Classes
`[a-zA-Z]`	match any char in the `[…]` set	
`[^a-zA-Z]`	match any char *not* in the `[…]` set	
`^`	match beginning of text	Anchors
`$`	match end of text	
x?	match 0 or 1 occurrences of *x*	Quantifiers
x+	match 1 or more occurrences of *x*	
*x**	match 0 or more occurrences of *x*	
x{*m*,*n*}	match between *m* and *n* *x*'s	
abc	match abc	
abc|xyz	match abc or xyz	Alternation

3

The examples listed above, cover the most common regular expression meta-characters. If you are new to regular expressions, this is a good table to remember.

The expression Character Class is introduced here. A Character Class may be specified in the 'traditional' way as show, using the square brackets [ ].
Note that meta-characters used inside [...] are different to those used outside. Escaped meta-characters (those prefixed with /) are literals, as are meta-characters inside [].

Regular expression objects

- **`import re` to use them**
  - `compile` compiles the RE for efficiency, returns a re object.
- **We can search or match**
  - `search` searches for a pattern - like conventional RE's .
  - `match` matches from the start of the string.
  - `fullmatch` matches from the start to the end of the string (3.4).
- **All return a MatchObject, or None (False)**

```
testy = 'The quick brown fox jumps over the lazy dog'

m = re.search(r"(quick|slow).*(fox|camel)", testy)
if m:
    print('Matched', m.groups())
    print('Starting at', m.start())
    print('Ending at', m.end())
```

```
Matched ('quick', 'fox')
Starting at 4
Ending at 19
```

Importing the re module allows methods on the re class to be called, with search and match being the most common. The fullmatch method was introduced at Python 3.4.0. They all return an object of class MatchObject, on which several methods may be called.

The group in parentheses is also known as a capturing parentheses group. Text inside parentheses may be referred to later in the RE as a back-reference. A useful method is groups(), which returns a tuple containing the matched text, and may be used like back-references. Other methods include start() and end(), which return the positions of the match, and group() which returns the matched string. There are several MatchObject attributes, including re, which gives the original regular expression, and string which gives the original input string.

If the search (or match) failed to find the pattern then the empty object None is returned, which is false in Boolean context.

Python RE syntax is similar to that used by lower-level language libraries, such as the GNU C regex package.
We can compile our REs for efficiency, for example:

```
reobj = re.compile (r"([Ii]).*(\1)")
for line in file:
    m = reobj.match(line)
    if m:
```

```
print m.string[m.start():m.end()]
```

Notice the use of raw strings (r"..."), these mean we do not have to escape (\) special characters like brackets and braces – particularly useful with Regular Expressions.

Regular expression substitution

- sub returns the changed string

  re.sub(*pattern, replacement, string*[*, count, flags*])

- subn returns a tuple: (*changed string, number of changes*)

  re.subn(*pattern, replacement, string*[*, count, flags*])

- ***count* gives the number of replacements**
- Default is to replace *all* occurrences

```
line = 'Perl for Perl Programmers'
txt = re.subn('Perl', 'Python', line)
if txt[1]:
    print(txt[0])
txt = re.subn('Perl', 'Python', line, 1)
if txt[1]:
    print(txt[0])
```

```
Python for Python Programmers
```

```
Python for Perl Programmers
```

Substitution is reminiscent of awk, in that we have a couple of method calls. sub is used where we just want the new string, whereas subn returns both the altered string and the number of matches. Both perform global substitutions from the left of the string.

There are other useful re functions, for example split, which is shown on the next slide.

We have also shown a compiled Regular Expression object. Compiling the RE, makes for more efficient code when the same pattern is used many times. For example, in a loop. Methods like match, fullmatch, findall, search, split, sub, and subn may be called on these objects.

QA Compiled REs and RE objects

- **We can compile for efficiency.**

```
reobj = re.compile (r"(1).*(\1)")
for line in file:
    m = reobj.match(line)
    if m:
        print(m.string[m.start():m.end()])
```

- **We can also compile to obtain a re object.**
- Method call parameters are different!

We have shown a compiled Regular Expression object. Compiling the RE makes for more efficient code when the same pattern is used many times, for example in a loop. Methods like match, findall, search, split, sub, and subn may be called on these objects, but the argument lists may be different to the versions you are used to.

Flags

- **Change the behaviour of the match.**

Long name	Short	RE	
re.ASCII	re.A	(?a)	Class shortcuts do not include Unicode
re.IGNORECASE	re.I	(?i)	Case insensitive match
re.LOCALE	re.L	(?L)	Class shortcuts are locale sensitive
re.MULTILINE	re.M	(?m)	^ and $ match start and end of *line*
re.DOTALL	re.S	(?s)	. also matches a new-line
re.VERBOSE	re.X	(?x)	Whitespace is ignored, allow comments

- May be embedded in the RE.
- May be specified as an optional argument to:
  - re.search, re.match, and re.compile
- Multiple flags may be combined.

```
m = re.search(r'(?im)^john', name)
m = re.search(r'^john', name, re.IGNORECASE|re.MULTILINE)
```

In other products, like sed and Perl 5, flags are placed after the regular expression, but in Python they are set before (Perl 6 uses a similar syntax). It would be tempting to state that the short names are the initial letter of the long name, and that the RE syntax is just the short name in lowercase. You can see that this is not the case, the S and X flags are there for compatibility with other RE engines (e.g., Perl).

The examples combine the IGNORECASE and MULTILINE flags. They look for 'john' in any case at the start of the text *or* immediately after a new-line character.

When embedded in the RE, the single characters can be in any order. When using the re module attribute flags, they are combined with a binary OR |, also in any order. The flags and embedded attributes may be mixed, but that might make the RE even more confusing.

Global matches

re.findall

- Returns a list of matches or groups.

```
str='/dev/sd3d 135398 69683 52176 57% /home/stuff'
nums = re.findall(r'\b\d+\b', str)
print(nums)
```

```
['135398', '69683', '52176', '57']
```

re.finditer

- Returns an iterator to a match object.

```
str='/dev/sd3d 135398 69683 52176 57% /home/stuff'

for num in re.finditer(r'\b(\d+)\b', str):
    print(num.groups())
```

```
('135398',)
('69683',)
('52176',)
('57',)
```

By default, most re methods search for the first (leftmost) occurrence of a pattern. These methods work on all occurrences and return either a list or an iterator to the occurrences. These are particularly useful for repeated patterns over multiple lines.

The `findall` method was added in Python 2.2.

Back-references

- **Python also allows 'back-references'.**
  - To create self-referencing regular expressions.
- **Indicated by \n, representing the 'nth' set of parentheses.**
- **Alternatively indicated by \g<n>**
  - Python extension.
  - Also supports \g<0> which is the text of the whole match.
- **May be used in the replacement in sub and subn.**

```
str='copyright 2005-2006'
print(re.sub(r'((19|20)[0-9]{2})-((19|20)[0-9]{2})',
             r'\1-2013', str))
```

```
copyright 2005-2013
```

Python supports the use of parentheses to group a number of characters or regular expressions into a single unit and then apply a regular expression quantifier to the entire group. This can be useful when the pattern consists of recurring blocks of text or words.

The group in parentheses is also known as a capturing parentheses group. Text inside parentheses may be referred to later in the RE as a back-reference. This is done by the use of r'\1', r'\2', etc. to refer to the contents of the first and second sets of parentheses. If you nest parentheses, the order of the opening parenthesis is the order in which the back-references are allocated. Make sure you use 'raw' strings for the back-references, since '\1' is itself a special character.

The alternative back-reference syntax using r'\g<1>', r'\g<2>', etc. is a Python extension, and is associated with named captures (see the Advanced Regular Expressions appendix). It has the additional feature of supporting group 0 (zero) in r'\g<0>', which represents the whole of the matched string (r'\0' is not supported).

The use of many back-references in a RE will cause Python to work harder and increase the time taken to find a match. Use sparingly!

Non-capturing groups

- **A parenthesis group creates group values.**
  - Incurs some run-time overhead for keeping track of this.

```
drink = 'A bottle of Miller'
pattern = 'A (glass|bottle|barrel) of (Bud|Miller|Coors)'
m = re.search(pattern, drink)
if m: print(m.groups())
```

```
('bottle', 'Miller')
```

- **`(?:)` parenthesis group without back-references.**
  - Incurs lower run-time overhead.

```
drink = 'A bottle of Miller'
pattern = 'A (?:glass|bottle|barrel) of (?:Bud|Miller|Coors)'
m = re.search(pattern, drink)
if m: print(m.groups())
```

```
()
```

10

Using (?:pattern) instead of (pattern) is useful for performance tuning, or if you have to introduce an extra group in the middle of a huge expression and you don't want to update all uses of m.groups() following it.

Other match methods

- `start()`: **list of starting offsets of each match.**
  - First element gives the offset to the start of the first match.
- `end()` : **list of ending offsets+1 of each match.**
  - First element gives the offset+1 to the end of the first match.
- **lastindex : index of last group matched.**

```
#       0123456789012345678901234567890123456789012345
txt = '2 456 first 3456 second third 98765 fourth 123'
m = re.search(r'(\d+) ([a-z]+) (\d+)',txt)
if m:
   print(m.groups())
   print([m.start(i) for i in range(1, m.lastindex+1)])
   print([m.end(i)   for i in range(1, m.lastindex+1)])
```

```
('456', 'first', '3456')
[2, 6, 12]
[5, 11, 16]
```

These lists are occasionally useful, offering lookups to text matching parentheses groups.

Named captures

- Give names to captures, not just boring `m.group()`...

```
(?P<name>RE-pattern)
```

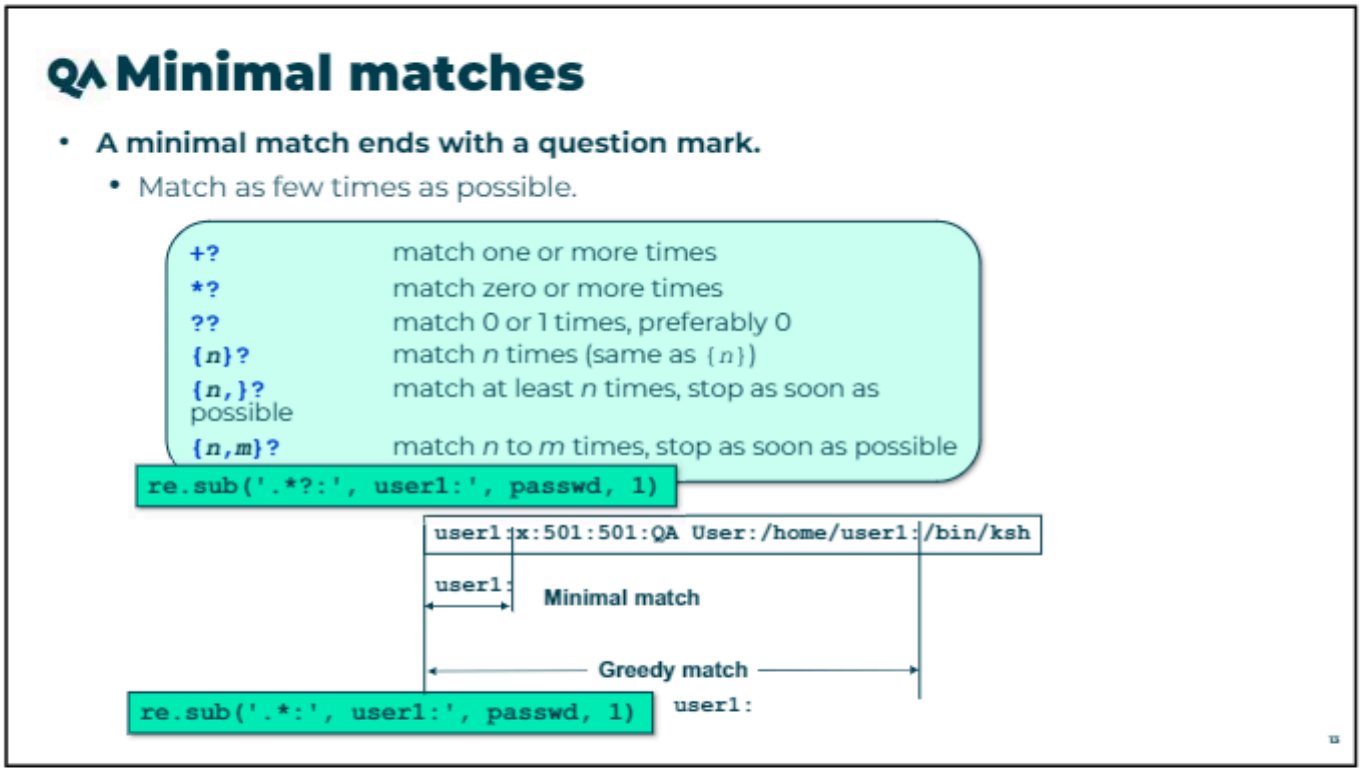
- Capture names and values are in `m.groupdict()`

```
df = '/dev/sd3d  135398  69683  52176  57%  /home/stuff'
m = re.search('''
        ^(?P<fs>\S+)    \s+ (?:\d+)        \s+ (?:\d+) \s+
         (?P<avail>\d+)\s+ (?P<cap>\d+)% \s+ (?P<mnt>\S+)
              ''', df, re.X)
if m:
    gd = m.groupdict()
    print('{0[mnt]} ({0[fs]}) has {0[avail]} ({0[cap]}%) free'.
          format(gd))
```

```
/home/stuff (/dev/sd3d) has 52176 (57%) free
```

One of the most eagerly awaited features of Python was 'named captures'. The ability to give a name to capturing parentheses groups.
The captured values are placed into a dictionary with the keys set to the capture names and the values of whatever was matched. They can be used as back-references in an re.sub replacement string using `(?P=name)`.

By the way, the trailing **`re.X`** option allows a new-line (or any white-space) to be embedded in the regular expression and ignored.



A minimal quantifier with match with as few characters as possible, whereas a maximal (the default) will attempt to match with as many as possible.
The descriptions of these quantifiers may seem strange, for example you might think that a minimal match of zero times will always match nothing! The key point to remember is that a regular expression will match the whole pattern if it possibly can, so a minimal pattern will match different numbers of characters, depending on the rest of the pattern. For example, in the text: "123456AAAAAA":

\d*	matches 123456
\d*?	matches, but $& is empty
\d{2}\d*?A	matches 123456A

In the last example, the \d{2} is followed by a minimal match and is followed by the letter 'A'. In this case, the minimal match has no choice but to match 4 decimal characters, if the whole pattern is to match. Given this text and pattern, the maximal match will produce the same result.
The example above, shows a typical problem with greedy matches

where we want to replace the first field, but end up replacing all except the final field. Incidentally, the traditional solution (in awk, for example) would be:

```
re.sub('[^:]*:','eric:',passwd,1)
```

Multi-line matches

- **Python allows searching a pattern across multiple lines:**
  - Just search a text which contains embedded "\n" characters.
- **However, Python is normally cautious:**
  - . character class does not match newline.
  - .* will only go until the end of the line.
- **re.S flag treats "\n" as a normal character:**
  - . matches any character, including newline.
  - .* will match until the end of the search text.
  - \s and \S shortcuts are not affected.

Normally . does not match a new-line character. With the /s option, a . does match a new-line, but not at the end of text.

Remember that character class shortcut, \s always matches [\t\n\r\f] and \S always matches [^ \t\n\r\f], regardless of the /s option. This also applies to the POSIX character class [:space:], but not [:blank:]. Character class [:ascii:] also includes the new-line character, whatever its position.

Multi-line matches

- **Normally, ^ and $ mean: start and end of search text.**
- **For multiple-line matches, this may be inconvenient.**
  - You may want to search the start and end of each line in the text.
- **`re.M` matches ^ and $ for each line within search text.**
  - Can be combined with `re.S` option.

```
all = open('names.txt').read()

m = re.search(r'(^dpm|^james)', all,re.I)
if m:
    print('File starts with', m.groups())

m = re.search(r'(^dpm|^james)', all, re.I|re.M)
if m:
    print('A line in the file starts with', m.groups())
```

```
Fred
DPM
Clive
Tom
Dick
Harry
```

```
A line in the file starts with ('DPM',)
```

Normally, python will treat the entire search text as a one string and not check for newlines within the string. If you read a file line by line, this does not matter. If you read an entire file into one variable containing multiple lines (slurping), it does matter, and you can use the m flag to handle this special case. Be careful though, \m means that . matches a line boundary, if a "\n" appears at the end-of-line there will be two matches, one for the "\n" and one for end-of-text ($).

Alternatives to ^ and $

\A matches beginning of string

- Will not match multiple times with /m.

\Z matches end of string, or before a "\n"

- The "\n" is optional.

\z matches end of string

- The real end-of-string, the "\n" is significant.

> "Knowing how long a piece of string is can only be useful if you can find its ends."

These special characters are not Python specific, and are used by many other RE engines.

Comments in regular expressions

- **Delimit comments with `(?#comment)`.**

```
re.search(
  r'\d{3}(?# 3 digits)\s(?# space)\d{3,5}(?# 3-5 digits)',
  txt)
```

- **The `re.X` flag allows comments in REs.**
- White-space in the regular expression is discarded.
- Comments until end-of-line allowed.

```
re.search(
    '''
      \d{3}       # 3 digits
      \s          # space
      \d{3,5}     # 3-5 digits
    ''',
    txt, re.X)
```

Whereas most people will agree that comments are a GOOD THING in 'normal' code, not everyone agrees that comments inside regular expressions add clarity. They can confuse the code, as seen in the first example.

The use of the /x modifier makes things more obvious, and that in itself is good. However, this modifier discards white-space, so if you need to include white-space in your RE you have to 'escape' (\ ) it.

Lambda in re.sub

- **The re.sub method can take a function as a replacement.**
  - Passes a match object to the function.
  - The return value is the value substituted.

```
import re

numbers = ['zero', 'wun', 'two', 'tree', 'fower',
           'fife', 'six', 'seven', 'ait', 'niner']
alphas  = ['alpha', 'bravo', 'charlie', 'delta', 'echo',
           'foxtrot', 'golf', 'hotel', 'india', 'juliet',
           'kilo', 'lima', 'mike', 'november', 'oscar', 'papa',
           'quebec', 'romeo', 'sierra', 'tango', 'uniform',
           'victor', 'whisky', 'xray', 'yankee', 'zulu']

codes = {str(i):name for i, name in enumerate(numbers)}
codes.update({name[0].upper():name for name in alphas})
reg = 'WG07 OKD'

result = re.sub(r'(\w)',
                lambda m: codes[m.groups()[0]] + ' ', reg)
```

The regular expression substitute operations (re.sub and re.subn) allow the second parameter to be a function (this is where the replacement string normally goes). The function can be a pre-defined named function, or a lambda.

In the example, we are constructing a dictionary. The keys come from a list of character ranges, and values are from the NATO phonetic alphabet. Care must be taken to ensure these lists are in the correct order.

The substitution matches any alphanumeric, and copies this to a match object by the use of a parentheses group `(\w)`. The single character is then used as a key to the dictionary.

The output in this case will be:

```
whisky golf zero seven  oscar kilo delta
```

Look-around assertions

- **Do not consume the pattern, or capture:**
  - Look-ahead: Positive: `(?=pattern)` Negative: `(?!pattern)`
  - Look-behind: Positive: `(?<=pattern)` Negative: `(?<!pattern)`
- **Without look-arounds:**

```
var = '<h1>This is a header</h1>'
m = re.search(r'<([hH]\d)>.*</\1>', var)
print("Matched: ", m.group())
```

```
Matched: <h1>This is a header</h1>
```

- **With look-arounds:**

```
var = '<h1>This is a header</h1>'
m = re.search(r'(?<=<([hH]\d)>).*(?=</\1>) ', var)
print("Matched: ", m.group())
```

```
Matched: This is a header
```

Look-ahead and look-behind can be used as an extra safeguard in a regular expression: match this, but only if (not) ...
Perl supports the following zero-width look-around assertions,

(?=pattern)	positive look-ahead
(?!pattern)	negative look-ahead
(?<=pattern)	positive look-behind
(?<!pattern)	negative look-behind

In the first example, without look-arounds, we have a parentheses group which captures the header number part of the tag, and then uses a back-reference \1 to match the same header. The RE is enclosed in ! because we have a literal / as part of the pattern. This matches the HTML statement, including the tags.

The second example looks behind for the first tag and looks ahead for the second. The value of \1 is not affected by the extra parentheses groups. This time the tags are not included in the match.

Look-behinds can be a performance overhead in Python.

Unusually, there is a restriction with the Python regular expression engine - it (currently) does not support variable length look-behinds. This restriction is shared with Perl and Ruby, but Java and PCRE do allow the patterns to be variable lengths. Other engines (and earlier versions) do not allow alternation in look-behinds. Look-aheads have no such restrictions.

Another example

Subtract 1 from the 2nd field for:

- `.log` files where the user (3rd field) is `root`
- `.dat` files where the type (4th field) is `system`

```
for line in open('log.txt'):
   nline = re.sub(
     '(?:(?<=\.log,)(\d+)(?=,root,)) |
      (?:(?<=\.dat,)(\d+)(?=.*,system$))',
     lambda m: str(int(m.group(1))-1) if m.group(1) \
               else str(int(m.group(2))-1), line, re.X)

   print(nline, end = "")
```

```
accounts.dat,2,user12,system
apache.log,1682,apache,daemon
var.log,23,root,user
profile.dat,57,home,user
payroll.dat,887,prd,system
```

→

```
accounts.dat,1,user12,system
apache.log,1682,apache,daemon
var.log,22,root,user
profile.dat,57,home,user
payroll.dat,886,prd,system
```


This RE uses an alternation | and each side must be grouped. We don't want to capture them, so `(?: ... )`.

The actual substitution is executed code, and it uses the capture generated in the match.

Finally, the **`re.X`** modifier allows us to split the RE over two lines to fit it on the slide!

SUMMARY

- Modifiers alter the effect of the RE.
- Side effect variables and capturing can be controlled.
- Minimal matches use a ? after the quantifier.
- Multi-line matching uses re.M and re.S.
- Look-around assertions.
- Substitution with interpolation.
- Substitution with expressions.

PEP008
Style Guide for Python Code

PEP:	8
Title:	Style Guide for Python Code
Version:	1a40d4eaa00b
Last-Modified:	2013-11-01 12:56:37 -0400 (Fri, 01 Nov 2013)
Author:	Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Nick Coghlan <ncoghlan at gmail.com>
Status:	Active
Type:	Process
Content-Type:	text/x-rst
Created:	05-Jul-2001
Post-History:	05-Jul-2001, 01-Aug-2013

Contents

Introduction

This document gives coding conventions for the Python code comprising the standard library in the main Python distribution. Please see the companion informational PEP describing style guidelines for the C code in the C implementation of Python [1].

This document and PEP 257 (Docstring Conventions) were adapted from Guido's original Python Style Guide essay, with some additions from Barry's style guide [2].

This style guide evolves over time as additional conventions are identified and past conventions are rendered obsolete by changes in the language itself.

Many projects have their own coding style guidelines. In the event of any conflicts, such project-specific guides take precedence for that project.

A Foolish Consistency is the Hobgoblin of Little Minds

One of Guido's key insights is that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. As PEP 20 says, "Readability counts".

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.

But most importantly: know when to be inconsistent -- sometimes the style guide just doesn't apply. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

In particular: do not break backwards compatibility just to comply with this PEP!

Some other good reasons to ignore a particular guideline:

1. When applying the guideline would make the code less readable, even for someone who is used to reading code that follows this PEP.
2. To be consistent with surrounding code that also breaks it (maybe for historic reasons) -- although this is also an opportunity to clean up someone else's mess (in true XP style).
3. Because the code in question predates the introduction of the guideline and there is no other reason to be modifying that code.
4. When the code needs to remain compatible with older versions of Python that don't support the feature recommended by the style guide.

Code lay-out

Indentation

Use 4 spaces per indentation level.

Continuation lines should align wrapped elements either vertically using Python's implicit line joining inside parentheses, brackets and braces, or using a hanging indent. When using a hanging indent, the following considerations should be applied; there should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation line.

Yes:

```
# Aligned with opening delimiter
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# More indentation included to distinguish this from the rest.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)
```

No:

```
# Arguments on first line forbidden when not using vertical alignment
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# Further indentation required as indentation is not distinguishable
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Optional:

```
# Extra indentation is not necessary.
foo = long_function_name(
  var_one, var_two,
  var_three, var_four)
```

The closing brace/bracket/parenthesis on multi-line constructs, may either line up under the first non-whitespace character of the last line of list, as in:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

or it may be lined up under the first character of the line that starts the multi-line construct, as in:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

Tabs or Spaces?

Spaces are the preferred indentation method.

Tabs should be used solely to remain consistent with code that is already indented with tabs.

Python 3 disallows mixing the use of tabs and spaces for indentation.

Python 2 code indented with a mixture of tabs and spaces should be converted to using spaces exclusively.

When invoking the Python 2 command line interpreter with the `-t` option, it issues warnings about code that illegally mixes tabs and spaces. When using `-tt`, these warnings become errors. These options are highly recommended!

Maximum Line Length

Limit all lines to a maximum of 79 characters.

For flowing long blocks of text with fewer structural restrictions (docstrings or comments), the line length should be limited to 72 characters.

Limiting the required editor window width makes it possible to have several files open side-by-side, and works well when using code review tools that present the two versions in adjacent columns.

The default wrapping in most tools disrupts the visual structure of the code, making it more difficult to understand. The limits are chosen to avoid wrapping in editors with the window width set to 80, even if the tool places a marker glyph in the final column when wrapping lines. Some web based tools may not offer dynamic line wrapping at all.

Some teams strongly prefer a longer line length. For code maintained exclusively or primarily by a team that can reach agreement on this issue, it is okay to increase the nominal line length from 80 to 100 characters (effectively increasing the maximum length to 99 characters), provided that comments and docstrings are still wrapped at 72 characters.

The Python standard library is conservative and requires limiting lines to 79 characters (and docstrings/comments to 72).

The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. Long lines can be broken over multiple lines by wrapping expressions in parentheses. These should be used in preference to using a backslash for line continuation.

Backslashes may still be appropriate at times. For example, long, multiple `with`-statements cannot use implicit continuation, so backslashes are acceptable:

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
        open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

Another such case is with `assert` statements.

Make sure to indent the continued line appropriately. The preferred place to break around a binary operator is *after* the operator, not before it. Some examples:

```
class Rectangle(Blob):

    def __init__(self, width, height,
                 color='black', emphasis=None, highlight=0):
        if (width == 0 and height == 0 and
                color == 'red' and emphasis == 'strong' or
                highlight > 100):
            raise ValueError("sorry, you lose")
        if width == 0 and height == 0 and (color == 'red' or
                                           emphasis is None):
            raise ValueError("I don't think so -- values are %s, %s" %
                             (width, height))
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)
```

Blank Lines

Separate top-level function and class definitions with two blank lines.

Method definitions inside a class are separated by a single blank line.

Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (e.g. a set of dummy implementations).

Use blank lines in functions, sparingly, to indicate logical sections.

Python accepts the control-L (i.e. ^L) form feed character as whitespace; Many tools treat these characters as page separators, so you may use them to separate pages of related sections of your file. Note, some editors and web-based code viewers may not recognise control-L as a form feed and will show another glyph in its place.

Source File Encoding

Code in the core Python distribution should always use UTF-8 (or ASCII in Python 2).

Files using ASCII (in Python 2) or UTF-8 (in Python 3) should not have an encoding declaration.

In the standard library, non-default encodings should be used only for test purposes or when a comment or docstring needs to mention an author name that contains non-ASCII characters; otherwise, using `\x`, `\u`, `\U`, or `\N` escapes is the preferred way to include non-ASCII data in string literals.

For Python 3.0 and beyond, the following policy is prescribed for the standard library (see PEP 3131): All identifiers in the Python standard library MUST use ASCII-only identifiers, and SHOULD use English words wherever feasible (in many cases, abbreviations and technical terms are used which aren't English). In addition, string literals and comments must also be in ASCII. The only exceptions are (a) test cases testing the non-ASCII features, and (b) names of authors. Authors whose names are not based on the Latin alphabet MUST provide a Latin transliteration of their names.

Open source projects with a global audience are encouraged to adopt a similar policy.

Imports

- Imports should usually be on separate lines, e.g.:

  ```
  Yes: import os
       import sys

  No:  import sys, os
  ```

  It's okay to say this though:

  ```
  from subprocess import Popen, PIPE
  ```

- Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.

  Imports should be grouped in the following order:

  1. standard library imports
  2. related third party imports
  3. local application/library specific imports

  You should put a blank line between each group of imports.

  Put any relevant `__all__` specification after the imports.

- Absolute imports are recommended, as they are usually more readable and tend to be better behaved (or at least give better error messages) if the import system is incorrectly configured (such as when a directory inside a package ends up on `sys.path`):

  ```
  import mypkg.sibling
  from mypkg import sibling
  from mypkg.sibling import example
  ```

  However, explicit relative imports are an acceptable alternative to absolute imports, especially when dealing with complex package layouts where using absolute imports would be unnecessarily verbose:

  ```
  from . import sibling
  from .sibling import example
  ```

  Standard library code should avoid complex package layouts and always use absolute imports.

  Implicit relative imports should *never* be used and have been removed in Python 3.

- When importing a class from a class-containing module, it's usually okay to spell this:

```
from myclass import MyClass
from foo.bar.yourclass import YourClass
```

If this spelling causes local name clashes, then spell them

```
import myclass
import foo.bar.yourclass
```

and use "myclass.MyClass" and "foo.bar.yourclass.YourClass".

- Wildcard imports (`from <module> import *`) should be avoided, as they make it unclear which names are present in the namespace, confusing both readers and many automated tools. There is one defensible use case for a wildcard import, which is to republish an internal interface as part of a public API (for example, overwriting a pure Python implementation of an interface with the definitions from an optional accelerator module and exactly which definitions will be overwritten isn't known in advance).

  When republishing names this way, the guidelines below regarding public and internal interfaces still apply.

Whitespace in Expressions and Statements

Pet Peeves

Avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets or braces.

```
Yes: spam(ham[1], {eggs: 2})
No:  spam( ham[ 1 ], { eggs: 2 } )
```

- Immediately before a comma, semicolon, or colon:

```
Yes: if x == 4: print x, y; x, y = y, x
No:  if x == 4 : print x , y ; x , y = y , x
```

- Immediately before the open parenthesis that starts the argument list of a function call:

```
Yes: spam(1)
No:  spam (1)
```

- Immediately before the open parenthesis that starts an indexing or slicing:

```
Yes: dict['key'] = list[index]
No:  dict ['key'] = list [index]
```

- More than one space around an assignment (or other) operator to align it with another.

  Yes:

```
x = 1
y = 2
long_variable = 3
```

  No:

```
x             = 1
y             = 2
long_variable = 3
```

Other Recommendations

- Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleans (and, or, not).
- If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). Use your own judgment; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator.

  Yes:

  ```
  i = i + 1
  submitted += 1
  x = x*2 - 1
  hypot2 = x*x + y*y
  c = (a+b) * (a-b)
  ```

  No:

  ```
  i=i+1
  submitted +=1
  x = x * 2 - 1
  hypot2 = x * x + y * y
  c = (a + b) * (a - b)
  ```

- Don't use spaces around the = sign when used to indicate a keyword argument or a default parameter value.

  Yes:

  ```
  def complex(real, imag=0.0):
      return magic(r=real, i=imag)
  ```

  No:

  ```
  def complex(real, imag = 0.0):
      return magic(r = real, i = imag)
  ```

- Compound statements (multiple statements on the same line) are generally discouraged.

  Yes:

  ```
  if foo == 'blah':
      do_blah_thing()
  do_one()
  do_two()
  do_three()
  ```

Rather not:

```
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

- While sometimes it's okay to put an if/for/while with a small body on the same line, never do this for multi-clause statements. Also avoid folding such long lines!

  Rather not:

  ```
  if foo == 'blah': do_blah_thing()
  for x in lst: total += x
  while t < 10: t = delay()
  ```

  Definitely not:

  ```
  if foo == 'blah': do_blah_thing()
  else: do_non_blah_thing()

  try: something()
  finally: cleanup()

  do_one(); do_two(); do_three(long, argument,
                               list, like, this)

  if foo == 'blah': one(); two(); three()
  ```

Comments

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. If a comment is a phrase or sentence, its first word should be capitalised, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

If a comment is short, the period at the end can be omitted. Block comments generally consist of one or more paragraphs built out of complete sentences, and each sentence should end in a period.

You should use two spaces after a sentence-ending period.

When writing English, Strunk and White apply.

Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don't speak your language.

Block Comments

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a `#` and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single `#`.

Inline Comments

Use inline comments sparingly.

An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1                 # Increment x
```

But sometimes, this is useful:

```
x = x + 1                 # Compensate for border
```

Documentation Strings

Conventions for writing good documentation strings (a.k.a. "docstrings") are immortalised in PEP 257.

- Write docstrings for all public modules, functions, classes, and methods. Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the `def` line
- PEP 257 describes good docstring conventions. Note that most importantly, the `"""` that ends a multiline docstring should be on a line by itself, and preferably preceded by a blank line, e.g.:

  ```
  """Return a foobang

  Optional plotz says to frobnicate the bizbaz first.

  """
  ```

  For one liner docstrings, it's okay to keep the closing `"""` on the same line.

Version Bookkeeping

If you have to have Subversion, CVS, or RCS crud in your source file, do it as follows.

```
__version__ = "$Revision: 1a40d4eaa00b $"
# $Source$
```

These lines should be included after the module's docstring, before any other code, separated by a blank line above and below.

Naming Conventions

The naming conventions of Python's library are a bit of a mess, so we'll never get this completely consistent -- nevertheless, here are the currently recommended naming standards. New modules and packages (including third party frameworks) should be written to these standards, but where an existing library has a different style, internal consistency is preferred.

Overriding Principle

Names that are visible to the user as public parts of the API should follow conventions that reflect usage rather than implementation.

Descriptive: Naming Styles

There are a lot of different naming styles. It helps to be able to recognise what naming style is being used, independently from what they are used for.

The following naming styles are commonly distinguished:

- `b` (single lowercase letter)
- `B` (single uppercase letter)
- `lowercase`
- `lower_case_with_underscores`
- `UPPERCASE`
- `UPPER_CASE_WITH_UNDERSCORES`
- `CapitalizedWords` (or CapWords, or CamelCase -- so named because of the bumpy look of its letters [3]). This is also sometimes known as StudlyCaps.

  Note: When using abbreviations in CapWords, capitalise all the letters of the abbreviation. Thus HTTPServerError is better than HttpServerError.

- `mixedCase` (differs from CapitalizedWords by initial lowercase character!)
- `Capitalized_Words_With_Underscores` (ugly!)

There's also the style of using a short unique prefix to group related names together. This is not used much in Python, but it is mentioned for completeness. For example, the `os.stat()` function returns a tuple whose items traditionally have names like `st_mode`, `st_size`, `st_mtime` and so on. (This is done to emphasise the correspondence with the fields of the POSIX system call struct, which helps programmers familiar with that).

The X11 library uses a leading X for all its public functions. In Python, this style is generally deemed unnecessary because attribute and method names are prefixed with an object, and function names are prefixed with a module name.

In addition, the following special forms using leading or trailing underscores are recognized (these can generally be combined with any case convention):

- `_single_leading_underscore`: weak "internal use" indicator. E.g. `from M import *` does not import objects whose name starts with an underscore.
- `single_trailing_underscore_`: used by convention to avoid conflicts with Python keyword, e.g.
- `Tkinter.Toplevel(master, class_='ClassName')`
- `__double_leading_underscore`: when naming a class attribute, invokes name mangling (inside class FooBar, `__boo` becomes `_FooBar__boo`; see below).
- `__double_leading_and_trailing_underscore__`: "magic" objects or attributes that live in user-controlled namespaces. E.g. `__init__`, `__import__` or `__file__`. Never invent such names; only use them as documented.

Prescriptive: Naming Conventions

Names to Avoid

Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names.

In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use 'l', use 'L' instead.

Package and Module Names

Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability. Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

Since module names are mapped to file names, and some file systems are case insensitive and truncate long names, it is important that module names be chosen to be fairly short - this won't be a problem on Unix, but it may be a problem when the code is transported to older Mac or Windows versions, or DOS.

When an extension module written in C or C++ has an accompanying Python module that provides a higher level (e.g. more object oriented) interface, the C/C++ module has a leading underscore (e.g. `_socket`).

Class Names

Class names should normally use the CapWords convention.

The naming convention for functions may be used instead, in cases where the interface is documented and used primarily as a callable.

Note that there is a separate convention for builtin names: most builtin names are single words (or two words run together), with the CapWords convention used only for exception names and builtin constants.

Exception Names

Because exceptions should be classes, the class naming convention applies here. However, you should use the suffix "Error" on your exception names (if the exception actually is an error).

Global Variable Names

(Let's hope that these variables are meant for use inside one module only). The conventions are about the same as those for functions.

Modules that are designed for use via `from M import *` should use the `__all__` mechanism to prevent exporting globals, or use the older convention of prefixing such globals with an underscore (which you might want to do to indicate these globals are "module non-public").

Function Names

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

mixedCase is allowed only in contexts where that's already the prevailing style (e.g. threading.py), to retain backwards compatibility.

Function and method arguments

Always use `self` for the first argument to instance methods.

Always use `cls` for the first argument to class methods.

If a function argument's name clashes with a reserved keyword, it is generally better to append a single trailing underscore rather than use an abbreviation or spelling corruption. Thus, `class_` is better than `clss`. (Perhaps better is to avoid such clashes by using a synonym.)

Method Names and Instance Variables

Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability.

Use one leading underscore only for non-public methods and instance variables.

To avoid name clashes with subclasses, use two leading underscores to invoke Python's name mangling rules.

Python mangles these names with the class name: if class Foo has an attribute named `__a`, it cannot be accessed by `Foo.__a`. (An insistent user could still gain access by calling `Foo._Foo__a`.) Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed.

Note: there is some controversy about the use of __names (see below).

Constants

Constants are usually defined on a module level and written in all capital letters with underscores separating words. Examples include `MAX_OVERFLOW` and `TOTAL`.

Designing for inheritance

Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public.

Public attributes are those that you expect unrelated clients of your class to use, with your commitment to avoid backward incompatible changes. Non-public attributes are those that are not intended to be used by third parties; you make no guarantees that non-public attributes won't change or even be removed.

We don't use the term "private" here, since no attribute is really private in Python (without a generally unnecessary amount of work).

Another category of attributes are those that are part of the "subclass API" (often called "protected" in other languages). Some classes are designed to be inherited from, either to extend or modify aspects of the class's behavior. When designing such a class, take care to make explicit decisions about which attributes are public, which are part of the subclass API, and which are truly only to be used by your base class.

With this in mind, here are the Pythonic guidelines:

- Public attributes should have no leading underscores
- If your public attribute name collides with a reserved keyword, append a single trailing underscore to your attribute name. This is preferable to an abbreviation or corrupted spelling. (However, notwithstanding this rule, 'cls' is the preferred spelling for any variable or argument which is known to be a class, especially the first argument to a class method)

  Note 1: See the argument name recommendation above for class methods.

- For simple public data attributes, it is best to expose just the attribute name, without complicated accessor/mutator methods. Keep in mind that Python provides an easy path to future enhancement, should you find that a simple data attribute needs to grow functional behavior. In that case, use properties to hide functional implementation behind simple data attribute access syntax

  Note 1: Properties only work on new-style classes.

  Note 2: Try to keep the functional behavior side-effect free, although side-effects such as caching are generally fine.

  Note 3: Avoid using properties for computationally expensive operations; the attribute notation

- If your class is intended to be subclassed, and you have attributes that you do not want subclasses to use, consider naming them with double leading underscores and no trailing underscores. This invokes Python's name mangling algorithm, where the name of the class is mangled into the attribute name. This helps avoid attribute name collisions should subclasses inadvertently contain attributes with the same name

  Note 1: Note that only the simple class name is used in the mangled name, so if a subclass chooses both the same class name and attribute name, you can still get name collisions.

  Note 2: Name mangling can make certain uses, such as debugging and `__getattr__()`, less convenient. However, the name mangling algorithm is well documented and easy to perform manually.

  Note 3: Not everyone likes name mangling. Try to balance the need to avoid accidental name clashes with potential use by advanced callers.

Public and internal interfaces

Any backwards compatibility guarantees, apply only to public interfaces. Accordingly, it is important that users be able to clearly distinguish between public and internal interfaces.

Documented interfaces are considered public, unless the documentation explicitly declares them to be provisional or internal interfaces exempt from the usual backwards compatibility guarantees. All undocumented interfaces should be assumed to be internal.

To better support introspection, modules should explicitly declare the names in their public API using the `__all__` attribute. Setting `__all__` to an empty list indicates that the module has no public API.

Even with `__all__` set appropriately, internal interfaces (packages, modules, classes, functions, attributes or other names) should still be prefixed with a single leading underscore.

An interface is also considered internal if any containing namespace (package, module or class) is considered internal.

Imported names should always be considered an implementation detail. Other modules must not rely on indirect access to such imported names, unless they are an explicitly documented part of the containing module's API, such as `os.path` or a package's `__init__` module that exposes functionality from submodules.

Programming Recommendations

- Code should be written in a way that does not disadvantage other implementations of Python (PyPy, Jython, IronPython, Cython, Psyco, and such)

  For example, do not rely on CPython's efficient implementation of in-place string concatenation for statements in the form `a += b` or `a = a + b`. This optimisation is fragile even in CPython (it only works for some types) and isn't present at all in implementations that don't use refcounting. In performance sensitive parts of the library, the `''.join()` form should be used instead. This will ensure that concatenation occurs in linear time across various implementations

- Comparisons to singletons like None should always be done with `is` or `is not`, never the equality operators

  Also, beware of writing `if x` when you really mean `if x is not None` -- e.g. when testing whether a variable or argument that defaults to None was set to some other value. The other value might have a type (such as a container) that could be false in a Boolean context!

- When implementing ordering operations with rich comparisons, it is best to implement all six operations (`__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, `__ge__`) rather than relying on other code to only exercise a particular comparison

  To minimise the effort involved, the `functools.total_ordering()` decorator provides a tool to generate missing comparison methods

  PEP 207 indicates that reflexivity rules *are* assumed by Python. Thus, the interpreter may swap `y > x` with `x < y`, `y >= x` with `x <= y`, and may swap the arguments of `x == y` and `x != y`. The `sort()` and `min()` operations are guaranteed to use the < operator and the `max()` function uses the > operator. However, it is best to implement all six operations so that confusion doesn't arise in other contexts

- Always use a def statement instead of an assignment statement that binds a lambda expression directly to a name

  Yes:

  ```
  def f(x): return 2*x
  ```

  No:

  ```
  f = lambda x: 2*x
  ```

The first form means that the name of the resulting function object is specifically 'f' instead of the generic '<lambda>'. This is more useful for tracebacks and string representations in general. The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit def statement (i.e. that it can be embedded inside a larger expression)

- Derive exceptions from `Exception` rather than `BaseException`. Direct inheritance from `BaseException` is reserved for exceptions where catching them is almost always the wrong thing to do

  Design exception hierarchies based on the distinctions that code *catching* the exceptions is likely to need, rather than the locations where the exceptions are raised. Aim to answer the question "What went wrong?" programmatically, rather than only stating that "A problem occurred" (see PEP 3151 for an example of this lesson being learned for the builtin exception hierarchy)

  Class naming conventions apply here, although you should add the suffix "Error" to your exception classes if the exception is an error. Non-error exceptions that are used for non-local flow control or other forms of signaling need no special suffix

- Use exception chaining appropriately. In Python 3, "raise X from Y" should be used to indicate explicit replacement without losing the original traceback

  When deliberately replacing an inner exception (using "raise X" in Python 2 or "raise X from None" in Python 3.3+), ensure that relevant details are transferred to the new exception (such as preserving the attribute name when converting KeyError to AttributeError, or embedding the text of the original exception in the new exception message)

- When raising an exception in Python 2, use `raise ValueError('message')` instead of the older form `raise ValueError, 'message'`

  The latter form is not legal Python 3 syntax

  The paren-using form also means that when the exception arguments are long or include string formatting, you don't need to use line continuation characters thanks to the containing parentheses

- When catching exceptions, mention specific exceptions whenever possible instead of using a bare `except:` clause

  For example, use:

```
try:
    import platform_specific_module
```

```
except ImportError:
    platform_specific_module = None
```

A bare `except:` clause will catch SystemExit and KeyboardInterrupt exceptions, making it harder to interrupt a program with Control-C, and can disguise other problems. If you want to catch all exceptions that signal program errors, use `except Exception:` (bare except is equivalent to `except BaseException:`)

A good rule of thumb is to limit use of bare 'except' clauses to two cases:

1. If the exception handler will be printing out or logging the traceback; at least the user will be aware that an error has occurred.
2. If the code needs to do some cleanup work, but then lets the exception propagate upwards with `raise`. `try...finally` can be a better way to handle this case.

- When binding caught exceptions to a name, prefer the explicit name binding syntax added in Python 2.6:

```
try:
    process_data()
except Exception as exc:
    raise DataProcessingFailedError(str(exc))
```

This is the only syntax supported in Python 3, and avoids the ambiguity problems associated with the older comma-based syntax.

- When catching operating system errors, prefer the explicit exception hierarchy introduced in Python 3.3 over introspection of `errno` values
- Additionally, for all try/except clauses, limit the `try` clause to the absolute minimum amount of code necessary. Again, this avoids masking bugs

Yes:

```
try:
    value = collection[key]
except KeyError:
    return key_not_found(key)
else:
    return handle_value(value)
```

No:

```
try:
    # Too broad!
    return handle_value(collection[key])
except KeyError:
    # Will also catch KeyError raised by handle_value()
    return key_not_found(key)
```

- When a resource is local to a particular section of code, use a `with` statement to ensure it is cleaned up promptly and reliably after use. A try/finally statement is also acceptable
- Context managers should be invoked through separate functions or methods whenever they do something other than acquire and release resources. For example:

  Yes:

  ```
  with conn.begin_transaction():
      do_stuff_in_transaction(conn)
  ```

  No:

  ```
  with conn:
      do_stuff_in_transaction(conn)
  ```

  The latter example doesn't provide any information to indicate that the __enter__ and __exit__ methods are doing something other than closing the connection after a transaction. Being explicit is important in this case

- Use string methods instead of the string module

  String methods are always much faster and share the same API with unicode strings. Override this rule, if backward compatibility with Pythons older than 2.0 is required

- Use `''.startswith()` and `''.endswith()` instead of string slicing to check for prefixes or suffixes

  startswith() and endswith() are cleaner and less error prone. For example:

  ```
  Yes: if foo.startswith('bar'):
  No:  if foo[:3] == 'bar':
  ```

- Object type comparisons should always use isinstance() instead of comparing types directly

  ```
  Yes: if isinstance(obj, int):
  ```

  ```
  No:  if type(obj) is type(1):
  ```

  When checking if an object is a string, keep in mind that it might be a unicode string too! In Python 2, str and unicode have a common base class, basestring, so you can do:

  ```
  if isinstance(obj, basestring):
  ```

  Note that in Python 3, `unicode` and `basestring` no longer exist (there is only `str`) and a bytes object is no longer a kind of string (it is a sequence of integers instead)

- For sequences, (strings, lists, tuples), use the fact that empty sequences are false

```
Yes: if not seq:
     if seq:

No: if len(seq)
    if not len(seq)
```

- Don't write string literals that rely on significant trailing whitespace. Such trailing whitespace is visually indistinguishable and some editors (or more recently, reindent.py) will trim them
- Don't compare boolean values to True or False using ==.

```
Yes:   if greeting:
No:    if greeting == True:
Worse: if greeting is True:
```

- The Python standard library will not use function annotations as that would result in a premature commitment to a particular annotation style. Instead, the annotations are left for users to discover and experiment with useful annotation styles

  It is recommended that third party experiments with annotations use an associated decorator to indicate how the annotation should be interpreted

  Early core developer attempts to use function annotations revealed inconsistent, ad-hoc annotation styles. For example:

  - `[str]` was ambiguous as to whether it represented a list of strings or a value that could be either *str* or *None*
  - The notation `open(file:(str,bytes))` was used for a value that could be either *bytes* or *str* rather than a 2-tuple containing a *str* value followed by a *bytes* value
  - The annotation `seek(whence:int)` exhibited a mix of over-specification and under-specification: *int* is too restrictive (anything with `__index__` would be allowed) and it is not restrictive enough (only the values 0, 1, and 2 are allowed). Likewise, the annotation `write(b: bytes)` was also too restrictive (anything supporting the buffer protocol would be allowed)
  - Annotations such as `read1(n: int=None)` were self-contradictory since *None* is not an *int*. Annotations such as `source_path(self, fullname:str) -> object` were confusing about what the return type should be
  - In addition to the above, annotations were inconsistent in the use of concrete types versus abstract types: *int* versus *Integral* and set/frozenset versus MutableSet/Set
  - Some annotations in the abstract base classes were incorrect specifications. For example, set-to-set operations require *other* to be another instance of *Set* rather than just an *Iterable*
  - A further issue was that annotations become part of the specification but weren't being tested
  - In most cases, the docstrings already included the type specifications and did so with greater clarity than the function annotations. In the remaining cases, the docstrings were improved once the annotations were removed

- The observed function annotations were too ad-hoc and inconsistent to work with a coherent system of automatic type checking or argument validation. Leaving these annotations in the code would have made it more difficult to make changes later so that automated utilities could be supported

References

[1] PEP 7, Style Guide for C Code, van Rossum

[2] Barry's GNU Mailman style guide http://barry.warsaw.us/software/STYLEGUIDE.txt

[3] http://www.wikipedia.com/wiki/CamelCase

[4] PEP 8 modernisation, July 2013 http://bugs.python.org/issue18472

Copyright