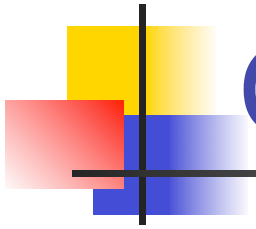
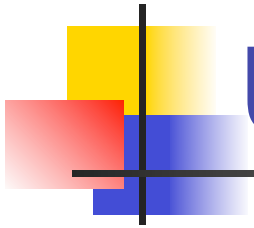


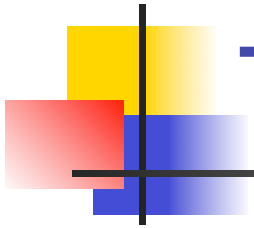
CS 11 python track: lecture 3

- Today: Useful coding idioms



Useful coding idioms

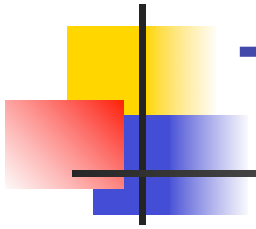
- "Idiom"
  - Standard ways of accomplishing a common task
- Using standard idioms won't make your code more correct, but
  - more concise
  - more readable
  - better designed (sometimes)



Trivial stuff (1)

- The `None` type and value:
- Sometimes, need a way to express the notion of a value which has no significance
  - often a placeholder for something which will be added later, or for an optional argument
- Use `None` for this
  - `None` is both a value and a type

```
>>> None
>>> type(None)
<type 'NoneType'>
```

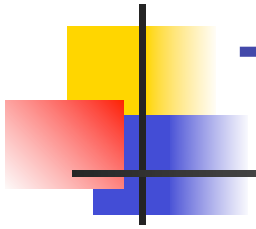


Trivial stuff (2)

- Can use the `return` keyword with no argument:

```
def foo(x):
    print x
    return  # no argument!
```

- Here, not needed; function will return automatically once it gets to the end
- Can use `return` with no argument if you want to exit the function before the end
- `return` with no argument returns a `None` value

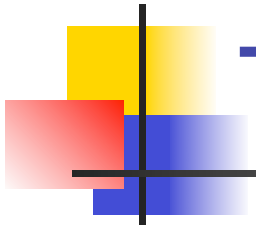


Trivial stuff (3)

- Can write more than one statement on a line, separated by semicolons:

```
>>> a = 1; b = 2
>>> a
1
>>> b
2
```

- Not recommended; makes code harder to read



Trivial stuff (4)

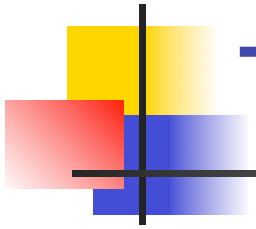
- Can write one-line conditionals:

```
if i > 0: break
```

  - Sometimes convenient
- Or one-line loops:

```
while True: print "hello!"
```

  - Not sure why you'd want to do this



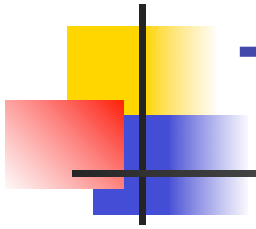
Trivial stuff (5)

- Remember the short-cut operators:
  - `+= -= *= /=` etc.
- Use them where possible
  - more concise, readable
- Don't write

```
i = i + 1
```

- Instead, write

```
i += 1
```

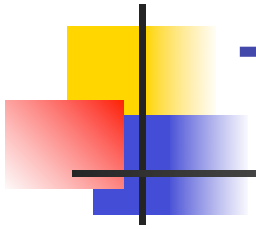


Trivial stuff (6)

- Unary minus operator
- Sometimes have a variable `a`, want to get its negation
- Use the unary minus operator:

```
a = 10
b = -a
```

- Seems simple, but I often see
  - `b = 0 - a`
  - `b = a * (-1)`



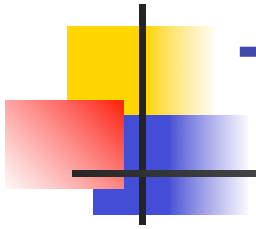
Trivial stuff (7)

- The `%g` formatting operator
- Can use `%f` for formatting floating point numbers when printing
- Problem: `%f` prints lots of trailing zeros:

```
>>> print "%f" % 3.14
3.140000
```

- `%g` is like `%f`, but suppresses trailing zeros:

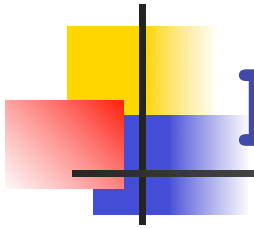
```
>>> print "%g" % 3.14
3.14
```



Trivial stuff (8)

- The `%s` formatting operator:
- `%s` can be used for any data type
  - all python data knows how to convert itself to a string
- Use `%s` in cases where you may not know what the type of the data is

```
print "data: %s" % some_unknown_data
```

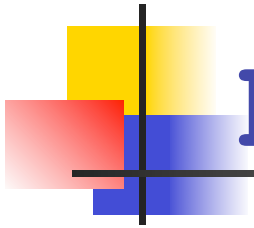


`print` (1)

- Recall that print always puts a newline after it prints something
- To suppress this, add a trailing comma:

```
>>> print "hello"; print "goodbye"
hello
goodbye
>>> print "hello", ; print "goodbye"
hello goodbye
>>>
```

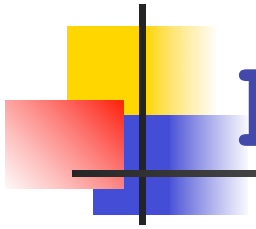
- N.B. with the comma, `print` still separates with a space



`print` (2)

- To print something without a trailing newline or a space, need to use the `write()` method of file objects:

```
>>> import sys
>>> sys.stdout.write("hello"); sys.stdout.write("goodbye")
hellogoodbye>>>
```



`print` (3)

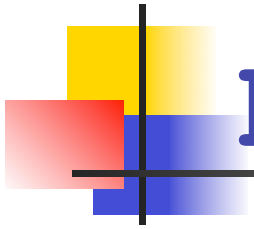
- To print a blank line, use `print` with no arguments:

```
>>> print
```

- Don't do this:

```
>>> print ""
```

- (It's just a waste of effort)

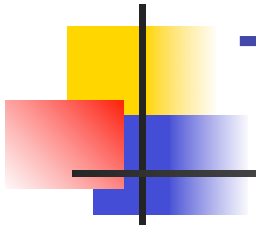


`print` (4)

- Can print multiple items with `print`:

```
>>> a = 10; b = "foobar"; c = [1, 2, 3]
>>> print a, b, c
10 foobar [1, 2, 3]
```

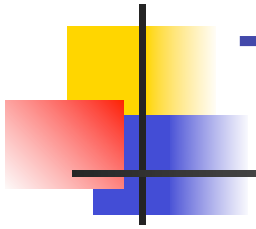
- `print` puts a space between each pair of items
- Usually better to use a format string
  - get more control over the appearance of the output



The `range()` function (1)

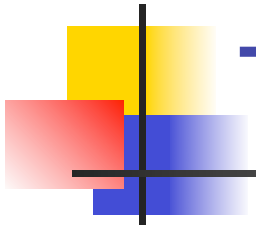
- The `range()` function can be called in many different ways:

```
range(5)              # [0, 1, 2, 3, 4]
range(3, 7)           # [3, 4, 5, 6]
range(3, 9, 2)        # [3, 5, 7]
range(5, 0, -1)       # [5, 4, 3, 2, 1]
```



The `range()` function (2)

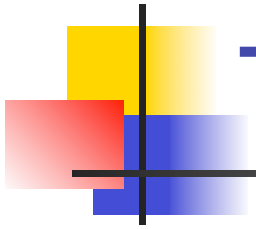
- `range()` has at most three arguments:
  - starting point of range
  - end point (really, 1 past end point of range)
  - step size (can be negative)
- `range()` with one argument
  - starting point == 0
  - step size == 1
- `range()` with two arguments
  - step size == 1



Type checking (1)

- Often want to check whether an argument to a function is the correct type
- Several ways to do this (good and bad)
- Always use the `type()` built-in function

```
>>> type(10)
<type 'int'>
>>> type("foo")
<type 'str'>
```



Type checking (2)

- To check if a variable is an integer:
- Bad:

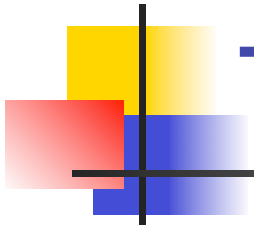
```
if type(x) == type(10): ...
```

- Better:

```
import types
if type(x) == types.IntType: ...
```

- Best:

```
if type(x) is int: ...
```



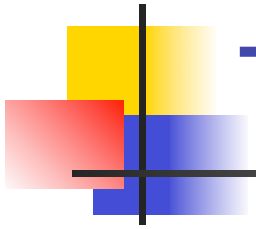
Type checking (3)

- Many types listed in the `types` module
- `IntType`, `FloatType`, `ListType`, ...
- Try this:

```
import types
dir(types)
```

  - (to get a full list)

```
>>> types.IntType
<type 'int'>
```

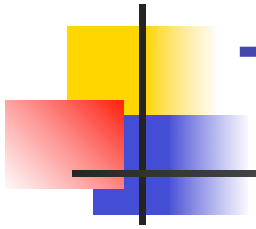


Type checking (4)

- Some type names are now built in to python:

```
>>> int
<type 'int'>
>>> list
<type 'list'>
>>> tuple
<type 'tuple'>
```

- So we don't usually need to `import types` any more



Type checking (5)

- You could write

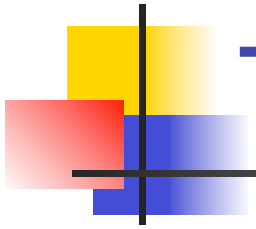
```
if type(x) == int: ...
```

- but this is preferred:

```
if type(x) is int: ...
```

- It looks better
- `is` is a rarely-used python operator
  - equivalent to `==` for types
- Can negate by writing the "is not" operator:

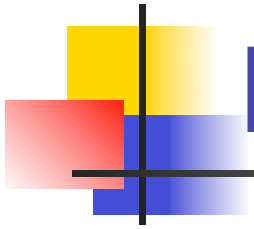
```
if type(x) is not int: ...
```



Type checking (6)

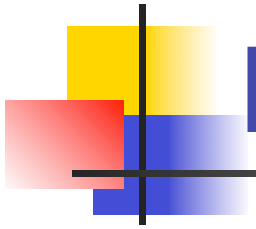
- How to check arguments to a function:

```
def foo(x):  # x should be an int
    if type(x) is not int:
        raise TypeError("bad type!")
    # code for the normal case
    # where x is an int
```



Note on exception handling (1)

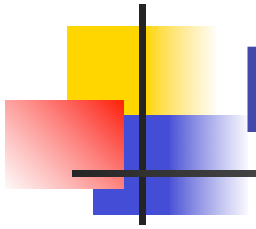
- When handling errors in function arguments, *do not print error messages to the terminal!*
  - and *especially* don't call `sys.exit(1)` !!!
- Instead, raise an exception, and make the error message an argument to the exception
  - most exceptions can take an error message as their first argument
- Then let the code that called the function decide what to do with the error (*e.g.* by catching the exception or ignoring it)



Note on exception handling (2)

- Reasons for this:
  - Error messages printed to the terminal are only useful for debugging
  - In contrast, exceptions can be caught by other code and possibly recovered from
  - Calling `sys.exit(1)` terminates the entire program, which is much too drastic!

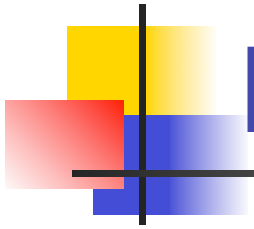
Note on exception handling (3)



- Can also include other relevant data in the error message *e.g.*

```
raise TypeError("expected int for arg 1, \
  got: %s" % arg1)
# arg1 is the 1st argument in this case
```

- Here, the error message reveals *why* the error occurred, not just *that* it occurred

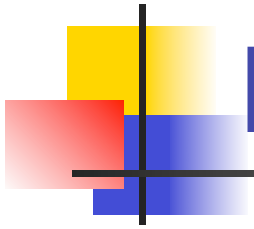


Note on exception handling (4)

- This is bad:

```
def foo(x):  # x should be an int
    if type(x) is not int:
        print >> sys.stderr, "bad type!"
        sys.exit(1)
    # code for the normal case...
```

- Why?

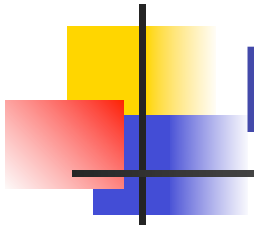


Note on exception handling (5)

- This is also bad:

```
def foo(x):  # x should be an int
    if type(x) is not int:
        print >> sys.stderr, "bad type!"
        raise TypeError
    # code for the normal case...
```

- Why?

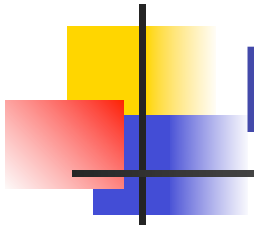


Note on exception handling (6)

- This is also bad:

```
def foo(x):  # x should be an int
    if type(x) is not int:
        raise TypeError("bad type")
        return
    # code for the normal case...
```

- Why?

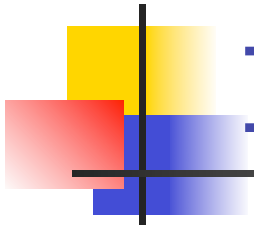


Note on exception handling (7)

- This is good:

```
def foo(x):  # x should be an int
    if type(x) is not int:
        raise TypeError("bad type")
    # code for the normal case...
```

- Why?

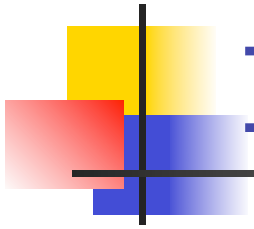


Instance checking (1)

- Instances of classes don't type check usefully:

```
class Foo: pass
class Bar: pass
f = Foo()
b = Bar()
print type(f) # <type 'instance'>
print type(b) # <type 'instance'>
```

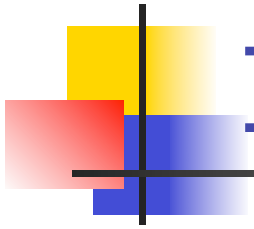
- Instances of different classes have same "type"
- What do we do to check for particular instance?



Instance checking (2)

- Use the `isinstance()` function:

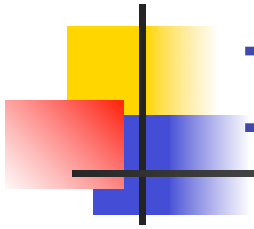
```
class Foo: pass
class Bar: pass
f = Foo()
b = Bar()
print isinstance(f, Foo)   # True
print isinstance(f, Bar)   # False
print isinstance(b, Foo)   # False
print isinstance(b, Bar)   # True
```



Instance checking (3)

- `isinstance()` and argument checking:

```
# f should be a Foo instance
def myfunction(f):
    if not isinstance(f, Foo):
        raise TypeError("invalid f")
    # code for the normal case...
```

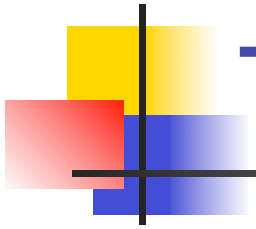


Instance checking (4)

- Another way to check instances:

```
# f should be a Foo instance
def myfunction(f):
    if f.__class__ is not Foo:
        raise TypeError("invalid f")
    # code for the normal case...
```

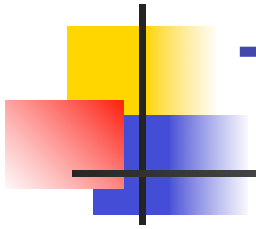
- `__class__` is another "magic attribute"
- returns the class of a given instance



Type conversions (1)

- Lots of built-in functions to do type conversions in python:

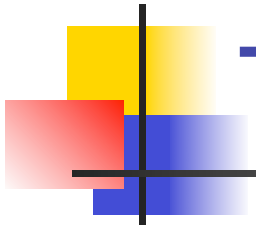
```
>>> float("42")
42.0
>>> float(42)
42.0
>>> int(42.5)
42
>>> int("42")
42
```



Type conversions (2)

- Converting to strings:

```
>>> str(1001)
'1001'
>>> str(3.14)
'3.14'
>>> str([1, 2, 3])
'[1, 2, 3]'
```

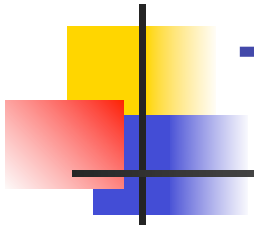


Type conversions (3)

- Different way to convert to strings:

```
>>> `1001`    # "back-tick" operator
'1001'
>>> a = 3.14
>>> `a`
'3.14'
>>> `[1, 2, 3]`
'[1, 2, 3]'
```

- Means the same thing as the `str` function



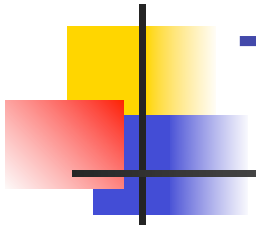
Type conversions (4)

- Converting to lists:

```
>>> list("foobar")
['f', 'o', 'o', 'b', 'a', 'r']
>>> list((1, 2, 3))
[1, 2, 3]
```

- Converting from list to tuple:

```
>>> tuple([1, 2, 3])
(1, 2, 3)
```



The "`in`" operator (1)

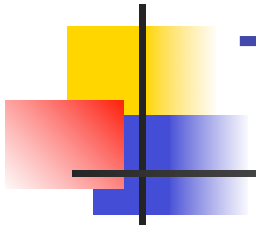
- The `in` operator is used in two ways:
  - 1) Iterating over some kind of sequence
  - 2) Testing for membership in a sequence
- Iteration form:

```
for item in sequence: ...
```

- Membership testing form:

```
item in sequence
```

(returns a boolean value)



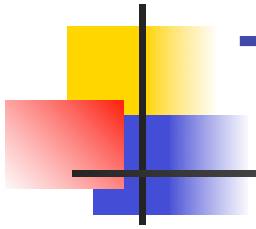
The "in" operator (2)

- Iterating over some kind of sequence

```
for line in some_file: ...
    # line is bound to each
    # successive line in the file "some_file"

for item in [1, 2, 3, 4, 5]: ...
   # item is bound to numbers 1 to 5

for char in "foobar": ...
   # char is bound to 'f', then 'o', ...
```



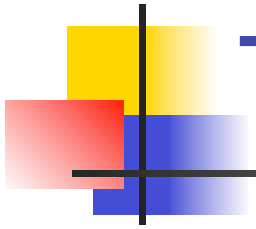
The "in" operator (3)

- Testing for membership in a sequence

```
# Test that x is either -1, 0, or 1:
lst = [-1, 0, 1]
x = 0
if x in lst:
    print "x is a valid value!"
```

- Can test for membership in strings, tuples:

```
if c in "foobar": ...
if x in (-1, 0, 1): ...
```



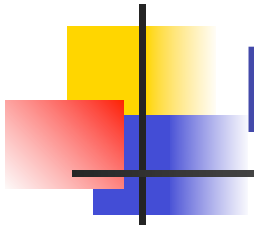
The "in" operator (4)

- Testing for membership in a dictionary:

```
>>> d = { "foo" : 1, "bar" : 2 }
>>> "foo" in d
True
>>> 1 in d
False
```

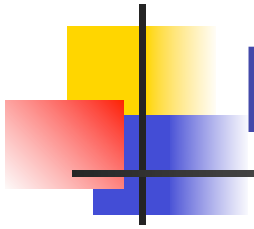
- Iterating through a dictionary:

```
>>> for key in d: print key
foo
bar
```



More stuff about lists (1)

- Use `lst[-1]` to get the last element of a list `lst`
- Similarly, can use `lst[-2]` to get second-last element
  - though it won't wrap around if you go past the first element
- The `pop()` method on lists:
  - `lst.pop()` will remove the last element of list `lst` and return it
  - `lst.pop(0)` will remove the first element of list `lst` and return it
  - and so on for other values

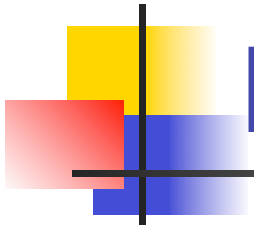


More stuff about lists (2)

- To copy a list, use an empty slice:

```
copy_of_lst = lst[:]
```

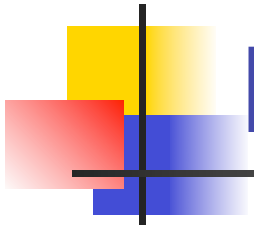
- This is a *shallow copy*
  - If `lst` is a list of lists, the inner lists will not be copied
  - Will just get a copy of the reference to the inner list
  - *Very* common source of bugs!
- If you need a *deep copy* (full copy all the way down), can use the `copy.deepcopy` function (in the `copy` module)



More stuff about lists (3)

```
>>> lst = [[1, 2], [3, 4]]
>>> copy_of_lst = lst[:]
>>> lst[0][0] = 10
>>> lst
[[10, 2], [3, 4]]
>>> copy_of_lst
[[10, 2], [3, 4]]
```

- This is probably not what you expected



More stuff about lists (4)

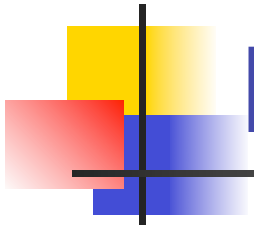
- Often want to make a list containing many copies of the same thing
- A shorthand syntax exists for this:

```
>>> [0] * 10     # or 10 * [0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- Be careful! This is still a shallow copy!

```
>>> [[1, 2, 3]] * 2
[[1, 2, 3], [1, 2, 3]]
```

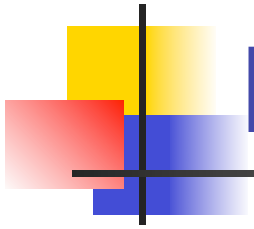
- Both elements are the *same* list!



More stuff about lists (5)

- The `sum()` function
- If a list is just numbers, can sum the list using the `sum()` function:

```
>>> lst = range(10)
>>> lst
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sum(lst)
45
```



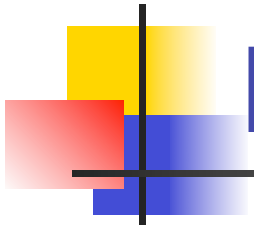
More stuff about strings (1)

- If you need a string containing the letters from `a` to `z`, use the `string` module

```
>>> import string
>>> string.lowercase
'abcdefghijklmnopqrstuvwxyz'
```

- If you need the count of a particular character in a string, use `string.count` or the `count` method:

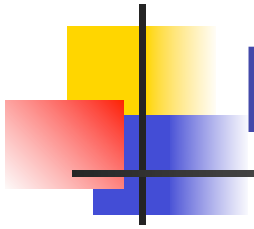
```
string.count("foobar", "o")   # 2
"foobar".count("o")   # also 2
```



More stuff about strings (2)

- Comparison operators work on strings
- Uses "lexicographic" (dictionary) order

```
>>> "foobar" < "foo"
False
>>> "foobar" < "goo"
True
```

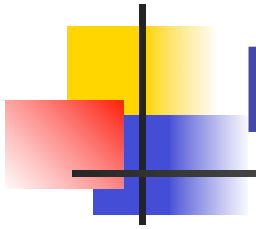


More stuff about strings (3)

- Can "multiply" a string by a number:

```
>>> "foo" * 3
'foofoofoo'
>>> 4 * "bar"
'barbarbarbar'
>>> 'a' * 20
'aaaaaaaaaaaaaaaaaaaa'
```

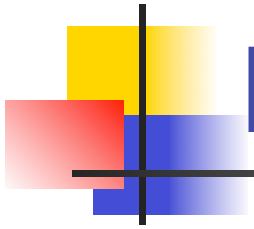
- This is occasionally useful



More stuff about tuples (1)

- Tuples can be used to do an in-place swap of two variables:

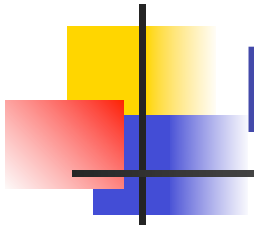
```
>>> a = 10; b = 42
>>> (a, b) = (b, a)
>>> a
42
>>> b
10
```



More stuff about tuples (2)

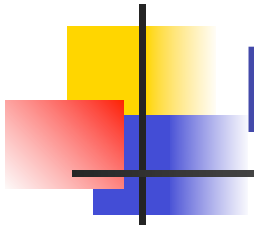
- This can also be written without parentheses:

```
>>> a = 10; b = 42
>>> a, b = b, a
>>> a
42
>>> b
10
```



More stuff about tuples (3)

- Why this works:
  - In python, the right-hand side of the `=` (assignment) operator is always evaluated before the left-hand side
  - the `(b, a)` on the right hand side packs the current versions of `b` and `a` into a tuple
  - the `(a, b) =` on the left-hand side unpacks the two values so that the new `a` is the old `b` etc.
- This is called "tuple packing and unpacking"



Random numbers (1)

- To use random numbers, import the `random` module; some useful functions include:

`random.choice(seq)`

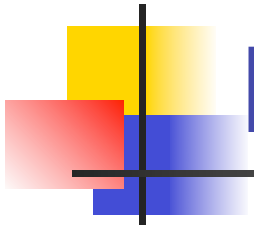
  - chooses a random element from a sequence `seq` (usually a list)

`random.shuffle(seq)`

  - randomizes the order of elements in a sequence `seq` (usually a list)

`random.sample(seq, k)`

  - chooses k random elements from `seq`



Random numbers (2)

- To use random numbers, import the `random` module; some useful functions include:

`random.randrange(start, stop)`

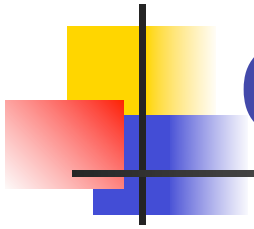
  - chooses a random element from the range [`start`, `stop`] (not including the endpoint)

`random.randint(start, stop)`

  - chooses a random element from the range [`start`, `stop`] (including the endpoint)

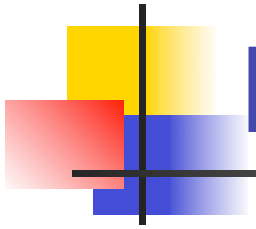
`random.random()`

  - returns a random float in the range (`0`, `1`)



Conclusion

- I expect you to know these idioms and use them where appropriate
  - ignoring them → lose marks!
- There are lots more idioms than are in this lecture
- If in doubt, use the `pydoc` program to access documentation of modules
  - Don't write a function from scratch if python already provides it!
  - That's called "reinventing the wheel" and it's very bad programming practice



Next week

- Finish up discussion of object-oriented programming in python
- Cover class inheritance
- Also a few more idioms and minor features