Today: string char class, more string loops, if/else, Doctests, grid, peeps example

Recall PEP8

See Python Guide: Python Style

Keyboard Shortcuts

Mention these when we are on the experimental server.

See Python Guide: Keyboard Shortcuts

These work on the experimental server and other places, including the PyCharm tool we'll show you on Fri. Ctrl-k works in GMail .. so satisfying!

- Run = cmd-enter
- Indent / Unindent (cmd-tab, shift-cmd-tab)
- Comment / Uncomment (cmd-/, shift-cmd-/)
- ctrl-k = delete line, super handy!

`double_char(s)` Pattern

- "pattern" - not writing code from scratch
- Think about code from a previous problem as a starting point
- Pattern - look at each char in a string
- Many problems look like that
- Think of the `double_char()` code as a starting point

> double_char

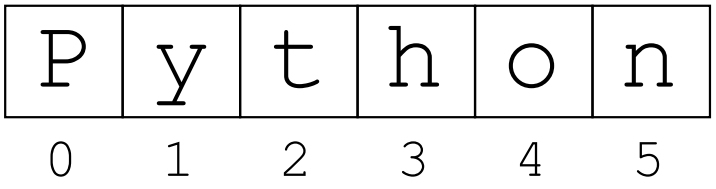
double_char(s): Given string s. Return a new string that has 2 chars for every char in s. So `'Hello'` returns `'HHeellllloo'`. Use a for/i/range loop.

Q: How to Look At Each Char in a String?

Recall: String

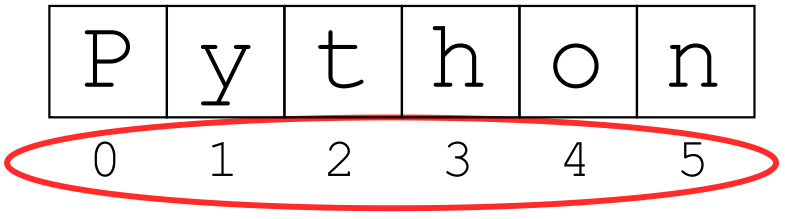
- String data type
- Linear sequence of characters (or "chars")
- Written within single or double quotes
- e.g. `'Python'` or `"Python"`
- Indexed 0 .. len-1

`'Python' (len 6)`



Want to Loop Over Index Numbers

`'Python' (len 6)`



Generate index numbers with:

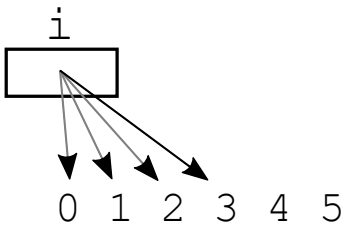
```
range(len(s))
```

Standard for/i/range loop:

```
for i in range(len(s))
    # use s[i]
```

Loop runs `i` through the index numbers. Use `s[i]` in the loop to access each char:

```
for i in range(len(s)):
    # use s[i]
```



`double_char()` Code

> double_char

- This function shows a pattern we can use for many "look at every char" problems
- 1. `result = ''` at start, `return result` last
- 2. `for i in range(len(s)):` - loop over all index numbers
- 3. `s[i]` - look at each char in turn
- 4. `result = result + xxx` - add something to the end of the result

```
def double_char(s):
    result = ''

    for i in range(len(s)):
        result = result + s[i] + s[i]

    return result
```

String Character Class Tests

alpha				
lower	upper	digit	space	misc
abcde..wxyz	ABCDE..WXYZ	0123456789	\n\t	!@#$%

- A string is made of characters
- Characters are divided into character classes:
- 1. **Alphabetic** - `'a' 'Z'` .. used to make words
- 2. **Digits** - `'0' '1'` .. `'9'` used to make numbers
- 3. **Space** - space `' '` , newline `'\n'` tab `'\t'`, - aka "whitespace"
  newline `'\n'` is char from the return/enter key on keyboard
- 4 **Misc** - `'$' '%' ';'` ... miscellaneous chars not in the first 3 categories
- There are noun.verb tests `s.isalpha s.isdigit() s.isspace()` returning boolean `True` or `False` if a char is in a category
  There is not a test for "misc", which is just a char which is not alpha or digit or space
- For empty string these return `False` (a weird edge case)

`s.isalpha()` - `True` for alphabetic word char, i.e. `'a-z'` and `'A-Z'`. Python uses "unicode" to support many alphabets, e.g. `'Ω'` is another alphabetic char.

`s.isdigit()` - `True` if all chars in s are digits `'0' '1' .. '9'`

`s.isspace()` - `True` for whitespace char, e.g. space, tab, newline

```
>>> 'a'.isalpha()
True
>>> 'abc'.isalpha()  # works for multiple chars
too
True
>>> 'Z'.isalpha()
True
>>> '$'.isalpha()
False
>>> '@'.isalpha()
False
>>> '9'.isdigit()
True
>>> ' '.isspace()
True
>>> 'Ω'.isalpha()     # Unicode
```

```
True
>>> 'Ω'.isdigit()
False
```

(demo/exercise optional) alpha_only(s)

> alpha_only

```
'H4ip2' -> 'Hip'
'12ab34ba' -> 'abba'
```

- Given string s
- Return a string of its chars which are alphabetic
  Use the `.isalpha()` test for each char in s
- e.g. `'H4ip2'` returns `'Hip'`
- Use the standard for/i/range loop
- If logic inside the loop
  Control if a char is added to result or not
- The double_char() code sets the pattern
  Look at every char in input s
  Build up result string with +=
- Example or exercise
  Look at double_char code above if needed
  First step - build straight copy of s
  Second - add logic to grab only alpha chars

Aside: handy hold music during coding exercises - Creative Commons (CC) licensed music.

Solution code

```
def alpha_only(s):
    result = ''

    # Loop over all index numbers
```

```
    for i in range(len(s)):
        # Access each s[i]
        if s[i].isalpha():
            result += s[i]

    return result
```

Regular if

Just as a reminder, here is the regular if-statement we've used many times. If the test is `True`, the lines are run. Otherwise if the test is `False`, the lines are skipped.

```
if test-expression:
  lines
```

if Variation: if / else

See guide for more if/else details: Python-if

Adding the `else:` clause to the if-statement:

```
if test-expression:
  lines-T  # run if test True
else:
  lines-F  # run if test False
```

- if/else variant:
  There are two sets of lines to control
  Above called "lines-T" and "lines-F"
- Run lines-T if test is `True`
- Run lines-F if test is `False`
- Use the if/else form if...
  There are 2 possible actions
  You always want to run one action or the other

- More common case of 1 action - use regular old if
- Only use "else" to alternate between 2 actions
- There is a more rare "elif" option we may cover later
- You do not need to use "else" on HW2

Example: str_dx()

> str_dx

```
'ab24z' -> 'xxddx'
```

- Return string where every digit changes to 'd'
  And every other char changed to 'x'
- e.g. `'ab42z'` returns `'xxddx'`
- Use if/else

Solution code

```
def str_dx(s):
    result = ''
    for i in range(len(s)):
        if s[i].isdigit():
            result += 'd'
        else:
            result += 'x'
    return result
```

else vs. not

Suppose you want to do something if a test is `False`. Sometimes people sort of back into using else to do that, like the following, but this is not the best way:

```
if some_test:
    pass  # do nothing here
else:
    do_something
```

The correct way to do that is with **not**:

```
if not some_test:
    do_something
```

Early-Return Strategy

The double_char code needs to go all the way to its last line to have its answer.

BUT sometimes an algorithm can figure out an answer earlier, using a `return` in that case to provide an answer without running the lines below. This is form of the pick-off strategy where sometimes we figure out the answer early.

has_alpha() Logic

You know at some level that the interior of the computer is very logical. This problem embodies that theme of very neat, sharp logic.

Consider this function:

> has_alpha

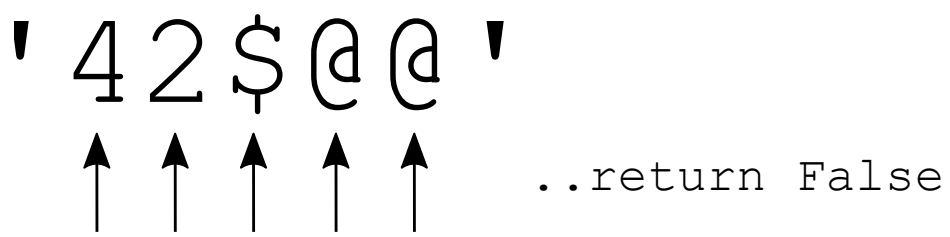
has_alpha(s): Given string s. Return True if there is an alphabetic char within s. If there is no alphabetic char within s, return False.

```
'3$abc' -> True
'42$@@' -> False
```

We might say this algorithm is like the book Are You My Mother?

has_alpha() Cases

Think about solving this problem. Look through the chars of the string. When do you know the answer?



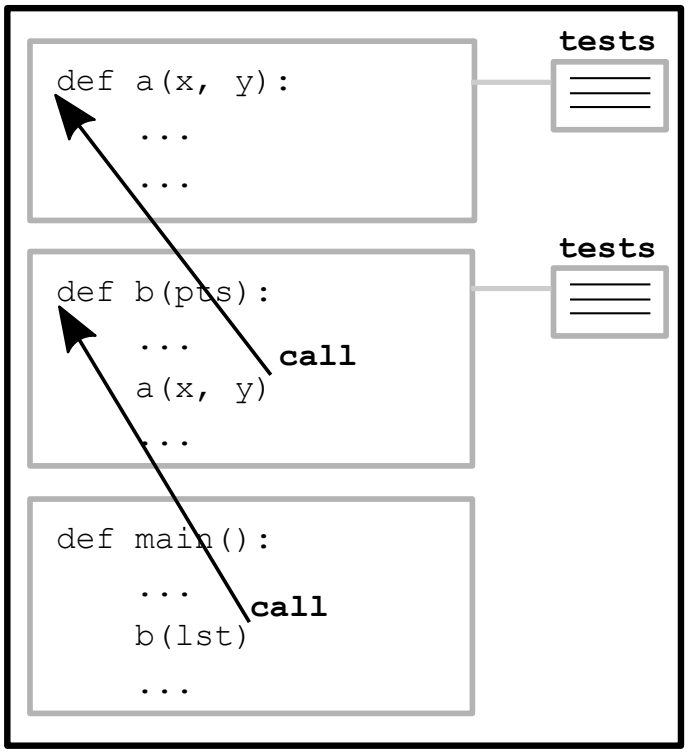
Logic strategy: look at each char. (1) If we see an alpha char, return `True` immediately. We do not need to look at any more chars. (2) If we look at every char, and never see an alpha char, must conclude that there are no alpha chars and the result should be `False`. The (1) code goes in the loop. The (2) code goes after the loop - a sort of "by exhaustion" strategy.

has_alpha() Solution

```
def has_alpha(s):
    for i in range(len(s)):
        if s[i].isalpha():
            # 1. Return True immediately if
find
            # alpha char - we're done.
            return True
    # 2. If we get to here, there was no
    # alpha char, so return False.
    return False
```

Big Picture - Program, Functions, Testing

Big Picture - program made of many functions. Want to build out the functions efficiently, concentrating on one function at a time.



How Do You Know A Function is Correct?

Bad news: You can't tell if a function is correct by looking at it.

Function Test - Cases - input/output pairs

Good news: It's pretty easy to have a few input/output tests for a function. Tests don't prove 100% that the function is correct, but in practice they work very well.

We're going to show you have to run and write your own function tests in Python. Python is very advanced for this - tests are easy to write, easy to run.

Tests - Work One Function at a Time

- Divide and Conquer strategy
- Divide program into many functions
- Work on one function at a time
  Deal with each function at a time
  Ideally **test** that function at the same time
- Write helper functions first, test them

- Later functions can call the helpers after they are tested
- Avoid debugging multiple functions at once
- Today: Python built-in tech for testing a function as you go

digits_only(s) Function

Given string s, return a string of its digit chars.

```
'Hi4x3' -> '43'
```

str1 project

- Download str1.zip project
- Expand to get "str1" folder
- Open the str1 folder in PyCharm, look at str1.py code

Python Function Doctests

Doctests are the Python technology for easily testing a function.

For more details, see the Doctest chapter in the guide

- Look at the digits_only() or str_dx() functions
- Look at the triple-quote """Pydoc""" of each
- Each ">>>" phrase is a test known as a Doctest

digits_only(s)

```
def digits_only(s):
    """
    Given a string s.
    Return a string made of all
```

```
    the chars in s which are digits,
    so 'Hi4!x3' returns '43'.
    Use a for/i/range loop.
    (this code is complete)
    >>> digits_only('Hi4!x3')
    '43'
    >>> digits_only('123')
    '123'
    >>> digits_only('xyz')
    ''
    >>> digits_only('')
    ''
    """
    result = ''
    for i in range(len(s)):
        if s[i].isdigit():
            result += s[i]
    return result
```

Python Function - Doctest

Here are the key lines that make one Doctest:

```
    >>> digits_only('Hi4!x3')
    '43'
```

- That syntax spells a test of 1 case
- Looks like a function call
- Input between the parens
- Output on the next line
- Very much in the black-box view of a function - input and outputs
- (Interpreter needs to be set for this to work)

- In PyCharm:
  Right click on the text of the Doctest
  Select "Run Doctest ..."
  Sometimes need to click a second time - PyCharm quirk
- Success: little green checkmark below the code area
  "Tests passed, 4 of 4"
  That means it worked perfectly
  This message could be more fun about the result!
- Otherwise: "Tests failed" in red
  Scroll down to see the input / expected / got data
  Look at **got** - your code produced that - why?
- Experiment: try putting in a bug, run Doctest, fix the bug
- Protip:
  Green "play" button at left re-runs most recent test
  Even better: ctrl-r (see top of Run menu for the keystroke for your machine)
  Re-running tests is super-common, so have a quick way to do it

Workflow: Functions with Tests

- Starting work on a function
- Write two or three Doctests **first**, before writing code
- The first test can just be an obvious case, like `'Hi4!x3'`
- Add an "edge" case test, like the empty string `''`
- Work on the code, run the tests to see where you are
  Debugging with these little tests is relatively easy
  Use green triangle button to re-run easily (control-r may work too)
- Eventually the tests pass and you're done!
- Green Checkmark!

We'll use Doctests to drive the examples in section and on homework-3.

Grid – Peeps Example

Today's grid example peeps.zip

Grid - 2D Algorithms

- We have done lots of 2D algorithms on images
  Always with RGB input/output
  Nice, but just one corner of 2D algorithms
- Grid - generic 2D facility
- In the grid.py file
- 2D storage of string, int, .. anything
- Reference: Grid Reference

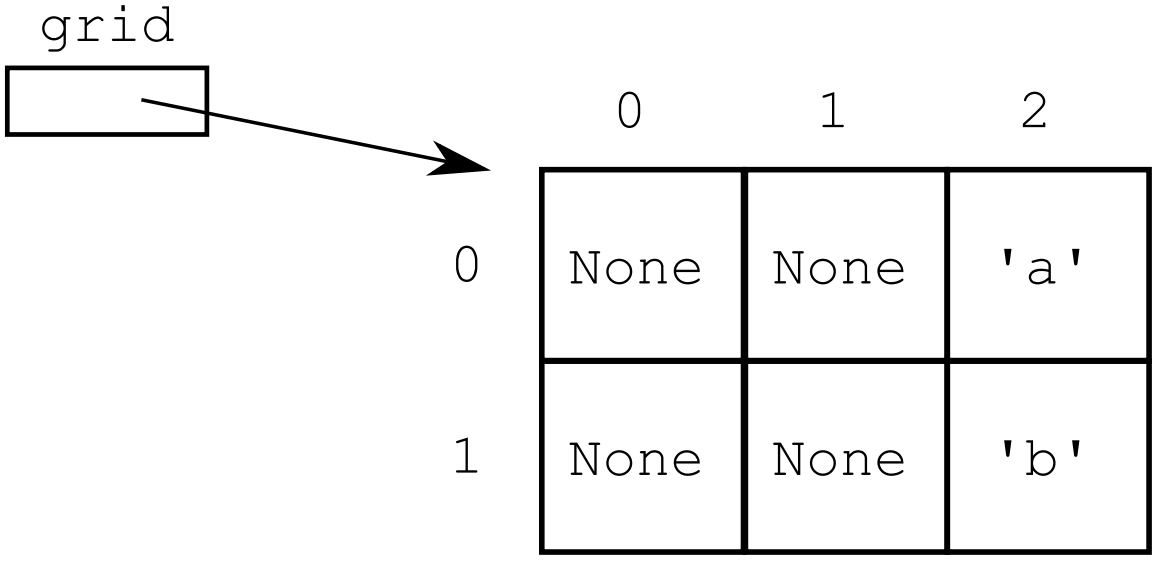
Grid Functions

- `grid = Grid(3, 2)` - create, all squares `None` initially
- Zero based x,y coordinates for every square in the grid:
  origin at upper left
  x: 0..grid.width - 1
  y: 0..grid.height - 1
- `grid.width, grid.height` - access width or height
- `grid.get(0, 0)` - returns contents at x,y (error if out of bounds)
- `grid.set(0, 0, 'a')` - set at x,y
- `grid.in_bounds(2, 2)` - return True if x,y is in bounds

Grid Example Code

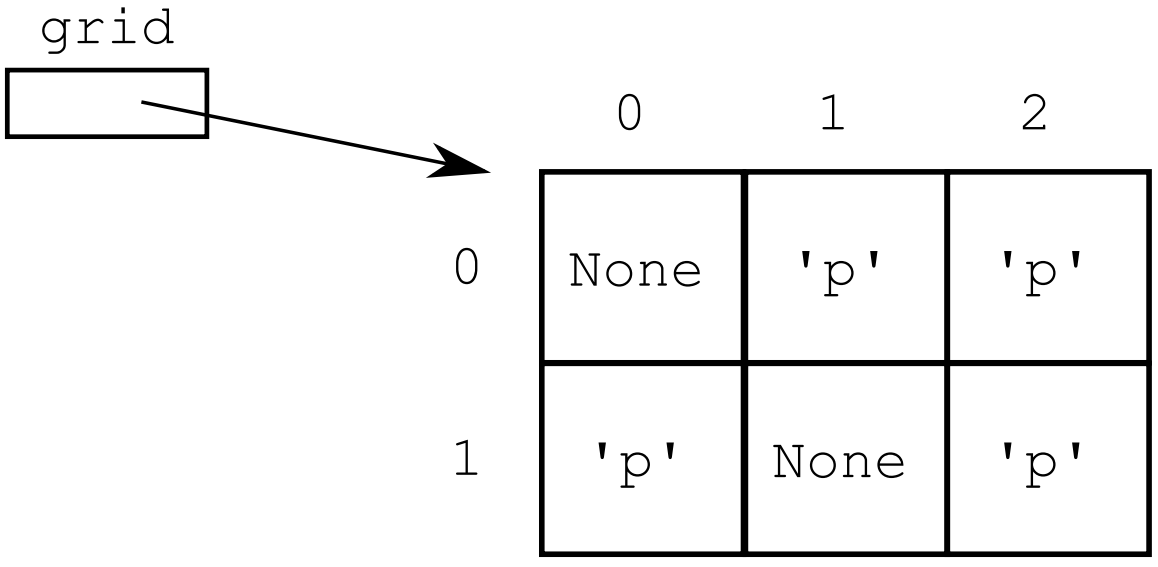
```
grid = Grid(3, 2)
grid.width # returns 3
grid.set(2, 0, 'a')
grid.set(2, 1, 'b')
```

```
grid.get(2, 0)  -> 'a'
grid.in_bounds(2, 0) -> True
```



Grid of Peeps

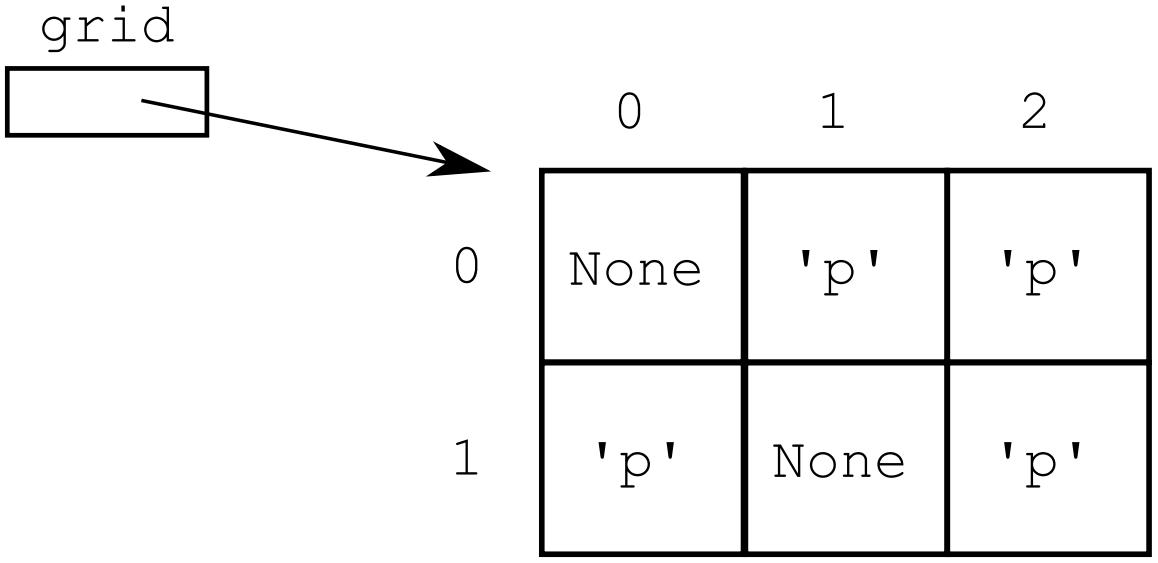
Suppose we have a 2-d grid of peeps candy bunnies. A square in the grid is either `'p'` if it contains a peep, or is `None` if empty.



Peep Happy Problem

We'll say that a peep is "happy" if it has another peep immediately to its left or right.

Look at the grid squares again. For each x,y .. is that a happy peep x,y?



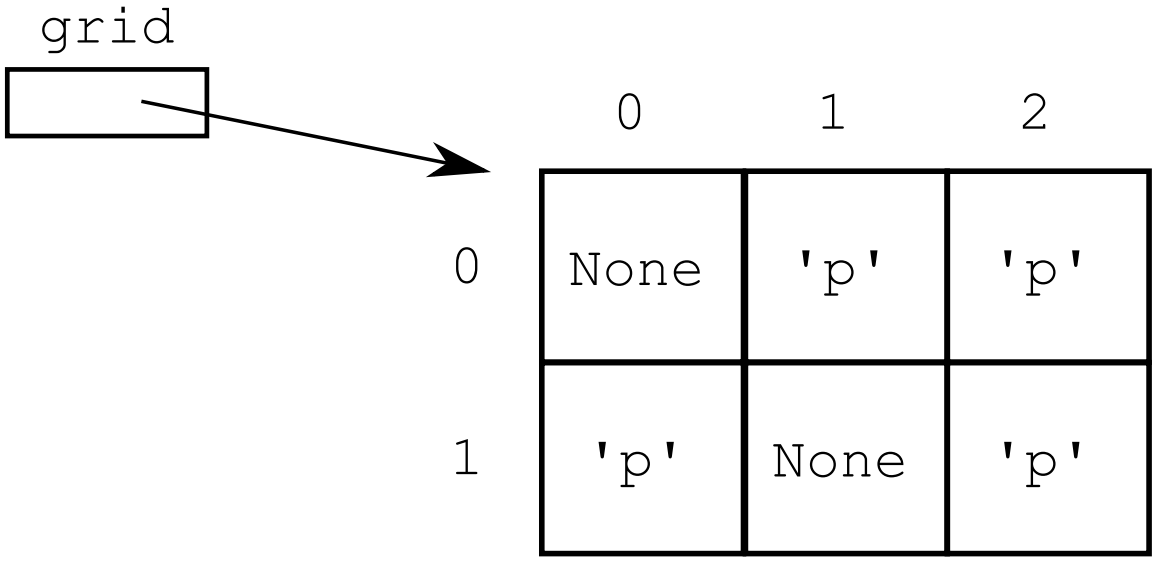

```
x, y happy?
(top row)
0, 0 -> False (no peep there)
1, 0 -> True
2, 0 -> True

(2nd row, nobody happy)
0, 1 -> False
1, 1 -> False
2, 1 -> False
```

is_happy(grid, x, y) Plan

- Build the is_happy(grid, x, y) function
- Write Doctests to check its output
- Need to be able write out a little grid for the tests

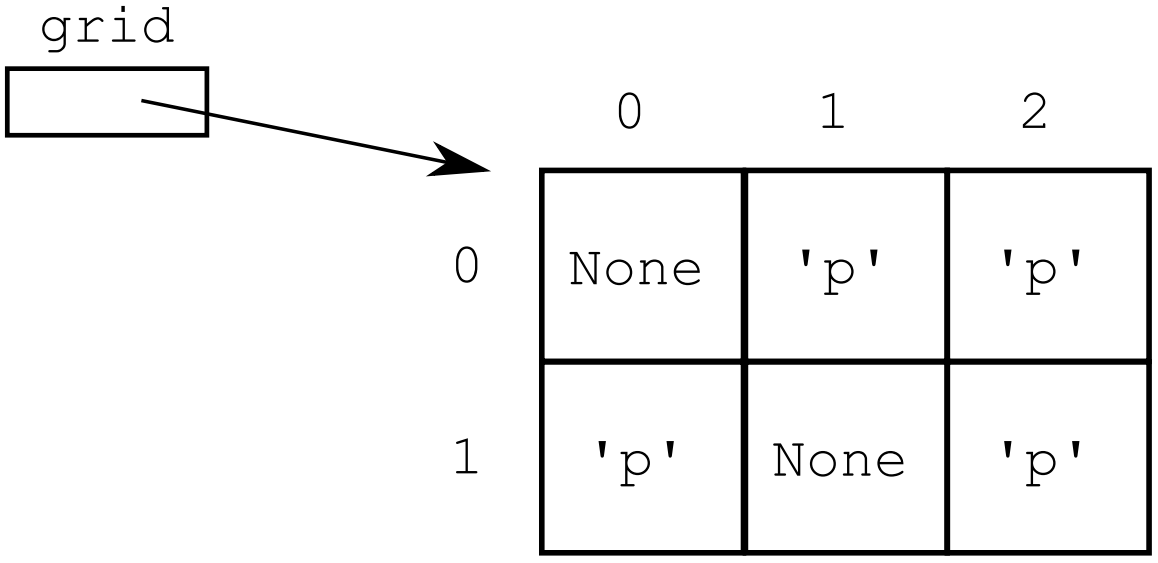
First Need: Grid Square Bracket Syntax



Here is the syntax for the above grid. The first [..] is the first row, the second [..] is the second row. This is fine for writing the data of a small grid, which is good enough for writing a test.

```
grid = Grid.build([[None, 'p', 'p'], ['p',
None, 'p']])
```

is_happy() Step 1 - Doctests



```
def is_happy(grid, x, y):
    """
    >>> grid = Grid.build([[None, 'p', 'p'],
['p', None, 'p']])
    >>> is_happy(grid, 0, 0)
    False
    >>> is_happy(grid, 1, 0)
    True
    >>> is_happy(grid, 0, 1)
    False
    >>> is_happy(grid, 2, 1)
    False
    """
    pass
```

Checking 4 representative squares. Removed """doc words""" so tests and drawing fit on screen at once.

Write is_happy() Code

- Use the if/return pick-off strategy
- 1. Pick off the case where x,y is not a peep
- Below (1), know that x,y is a peep
- 2. Look for another peep to the left
  Left is x-1
  Must check that x-1 is in bounds before calling get()
  Without the check, get an "out of bounds" error
- 3. Look for another peep to the right .. similar code
- 4. If (2) and (3) did not find anything, return False

is_happy() Code v1

This code is fine. Using the "pick-off strategy, looking for cases to return `True`. Then return `False` as the bottom if none of the cases found another peep.

```
def is_happy(grid, x, y):
    """
    Given a grid of peeps and in bounds x,y.
    Return True if there is a peep at that x,y
    and it is happy.
    A peep is happy if there is another peep
    immediately to its left or right.
    >>> grid = Grid.build([[None, 'p', 'p'],
['p', None, 'p']])
    >>> is_happy(grid, 0, 0)
    False
    >>> is_happy(grid, 1, 0)
    True
    >>> is_happy(grid, 0, 1)
    False
    >>> is_happy(grid, 2, 1)
    False
    """
    # 1. Check if there's a peep at x,y
    # If not we can return False immediately.
    if grid.get(x, y) != 'p':
        return False

    # 2. Happy because of peep to left?
    # Must check that x-1 is in bounds before
calling get()
    if grid.in_bounds(x - 1, y):
        if grid.get(x - 1, y) == 'p':
            return True

    # 3. Similarly, is there a peep to the
right?
    if grid.in_bounds(x + 1, y):
        if grid.get(x + 1, y) == 'p':
            return True
```

```
    # 4. If we get to here, not a happy peep,
    # so return False
    return False
```

is_happy() Using `and`

The in_bounds() checks can be done with `and` instead nesting 2 ifs. This works because the "and" works through its tests left-to-right, and stops as soon as it gets a `False`. This code is a little shorter, but both approaches are fine.

```
    # 2. Happy because of peep to left?
    # here using "and" instead of 2 ifs
    if grid.in_bounds(x - 1, y) and grid.get(x
- 1, y) == 'p':
        return True
```

Example - has_happy()

(Do this if we have time.)

Say we want to know for one column in the grid, is there a happy peep in there? A column in the grid is identified by its x value - e.g. x == 2 is one column in the grid.

Below is the def for this. The parameters are **grid** and **x** - the **x** identifies the column to check. The return True/False strategy is similar to the one seen in the has_alpha() example above. Doctests are provided, write the code to make it work.

```
def has_happy(grid, x):
    """
    Given grid of peeps and an in-bounds x.
    Return True if there is a happy peep in
    that column somewhere, or False if there
    is no happy peep.
    >>> grid = Grid.build([[None, 'p', 'p'],
['p', None, 'p']])
    >>> has_happy(grid, 0)
```

```
    False
    >>> has_happy(grid, 1)
    True
    """
    # your code here
    pass
```

has_happy() Solution

```
def has_happy(grid, x):
    """
    Given grid of peeps and an in-bounds x.
    Return True if there is a happy peep in
    that column somewhere, or False if there
    is no happy peep.
    >>> grid = Grid.build([[None, 'p', 'p'],
['p', None, 'p']])
    >>> has_happy(grid, 0)
    False
    >>> has_happy(grid, 1)
    True
    """
    # x is specified in parameter, loop over
all y
    for y in range(grid.height):
        if is_happy(grid, x, y):
            return True
    # if we get here .. no happy peep found
    return False
```

Doctest Strategy

We're just starting down this path with Doctests. Doctests enable writing little tests for each black-box function as you go, which turns out to be big productivity booster. We will play with this in section and on homework-3.