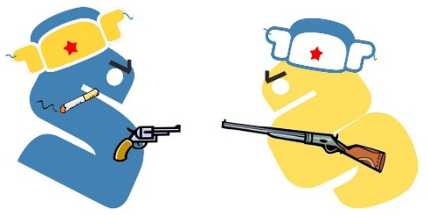
What is Python?

- Python is a widely used programming language
- First implemented in 1989 by Guido van Rossum
- Free, open-source software with community-based development
- Trivia: Python is named after the BBC show "Monty Python's Flying Circus" and has nothing to do with reptiles

Van Rossum is known as a "Benevolent Dictator For Life" (BDFL)

Which Python?

- There are 2 widely used versions of Python: Python2.7 and Python3.x
- We'll use Python**3**
- Many help forums still refer to Python2, so make sure you're aware which version is being referenced



Interacting with Python

There are 2 main ways of interacting with Python:

	Interactive mode	Normal mode
Description	Takes single user inputs, evaluates them, and returns the result to the user (**read–eval–print loop** (**REPL**))	Execute a Python **script** on the Unix command prompt
Benefits	• Use as a **sandbox**: explore new features • Easy to write quick **“throw away” scripts** • Useful for debugging • Use it as a calculator!	• Run long complicated programs • The script contains all of the commands

Usage	`$ python3`
	`Python 3.4.0 (default, Apr 11 2014, 13:05:11) [GCC 4.8.2] on linux2 Type "help", "copyright", "credits" or "license" for more information.`
	`>>>`
		`$ python3 <script.py>`

This is Python’s command prompt. It means, “I’m ready for a command!” Don’t type the “>>>”

Variables

- The most basic component of any programming language are "things," also called **variables**
- A variable has a name and an associated value
- The most common types of variables in Python are:

Type	Description	Example
Integers	A whole number	x = 10
Floats	A real number	x = 5.6
Strings	Text (1 or more **characters**)	x = "Genomics"
Booleans	A binary outcome: true or false	x = True

You can use single quotes or double quotes

Variables (cont.)

- To save a variable, use =

```
>>> x = 2
```

The *value* of the variable

The *name* of the variable

- To determine what type of variable, use the **type function**

```
>>> type(x)
<class 'int'>
```

- IMPORTANT: the variable name must be on the left hand side of the =

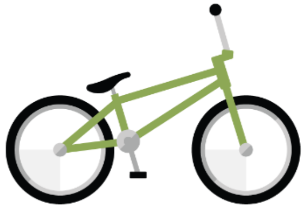
```
>>> x = 2
```
😁

```
>>> 2 = x
```
😢

Variable naming (best) practices

- Must start with a letter
- Can contain letters, numbers, and underscores ← no spaces!
- Python is case-sensitive: x ≠ X
- *Variable names should be descriptive and have reasonable length*
- Use ALL CAPS for constants, e.g., PI
- Do not use names already reserved for other purposes (min, max, int)

Want to learn more tips? Check out http://www.makinggoodsoftware.com/2009/05/04/71-tips-for-naming-variables/



Exercise: defining variables

- Create the following variables for
  - Your favorite gene name
  - The expression level of a gene
  - The number of upregulated genes
  - Whether the *HOXA1* gene was differentially expressed
- What is the type for each variable?

Cheatsheet

Type	Description	Example
Integers	A whole number	x = 10
Floats	A real number	x = 5.6
Strings	Text (1 or more **characters**)	x = “Genomics”
Booleans	A binary outcome: true or false	x = True

You can use single quotes or double quotes

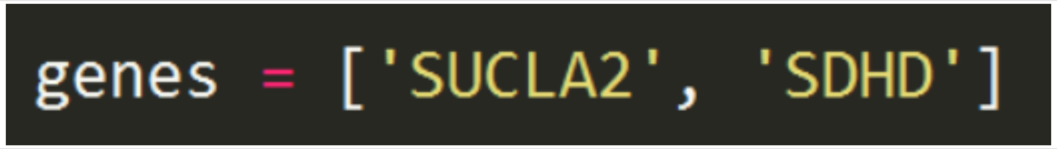
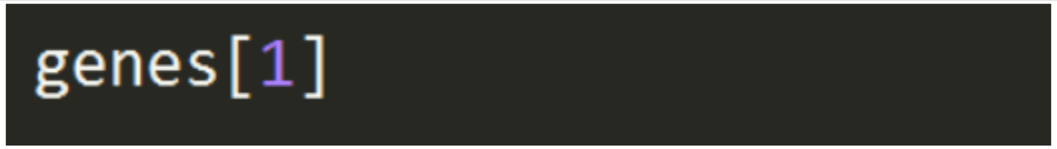
Collections of things

- Why is this concept useful?
  - We often have collections of things, e.g.,
    - A list of genes in a pathway
    - A list of gene fusions in a cancer cell line
    - A list of probe IDs on a microarray and their intensity value
  - We *could* store each item in a collection in a separate variable, e.g.,

    ```
    gene1 = ‘SUCLA2’
    gene2 = ‘SDHD’
    ...
    ```
  - A better strategy is to put all of the items in one container
- Python has several types of containers
  - **List** (similar to arrays)
  - **Set**
  - **Dictionary**

Lists: what are they?

- Lists hold a collection of things in a specified order
  - The things do not have to be the same type
- Many methods can be used to manipulate lists.

Syntax	Example	Output
Create a list		
`<list_name> = [<item1>, <item2>]`	 `genes = ['SUCLA2', 'SDHD']`	
Index a list		
`<listname>[<position>]`	 `genes[1]`	`'SDHD'`

Lists: where can I learn more?

- Python.org tutorial: https://docs.python.org/3.4/tutorial/datastructures.html#more-on-lists
- Python.org documentation: https://docs.python.org/3.4/library/stdtypes.html#list

Doing stuff to variables

- There are 3 common tools for manipulating variables
  - **Operators**
  - **Functions**
  - **Methods**

Operators

- Operators are a special type of function:
  - Operators are symbols that perform some mathematical or logical operation
- Basic mathematical operators:

Operator	Description	Example

`+`	Addition	`>>> 2 + 3`
		`5`

`-`	Subtraction	`>>> 2 - 3`
		`-1`

`*`	Multiplication	`>>> 2 * 3`
		`6`

`/`	Division	`>>> 2 / 3`
		`0.6666666666666666`

Operators (cont.)

You can also use operators on strings!

Operator	Description	Example

+	Combine strings together	`>>> 'Bio' + '5488'`
		`'Bio5488'`
		`>>> 'Bio' + 5488`
		`Traceback (most recent call last):`
		`  File "<stdin>", line 1, in <module>`
		`TypeError: Can't convert 'int' object to str implicitly`

Is it a bird? Is it a plane? No it's a string!

Strings and ints cannot be combined

Relational operators

- Relational operators compare 2 things
- Return a boolean

Operator	Description	Example

`<`	Less than	`>>> 2 < 3`
		`True`

`<=`	Less than or equal to	`>>> 2 <= 3`
		`True`

`>`	Greater than	`>>> 2 > 3`
		`False`

`>=`	Greater than or equal to	`>>> 2 >= 3`
		`False`

`==`	Equal to	`>>> 2 == 3`
		`False`

`!=`	Not equal to	`>>> 2 != 3`
		`True`

== is used to *test for equality*
= is used to *assign a value to a variable*

Logical operators

- Perform a logical function on 2 things
- Return a boolean

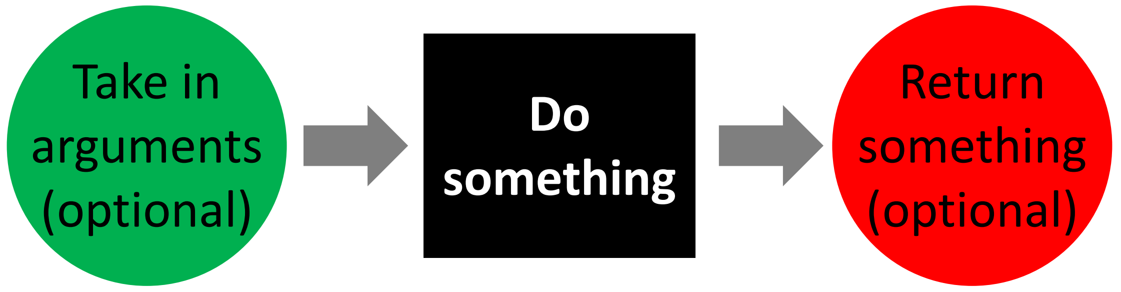
Operator	Description	Example

and	Return True if *both* arguments are true	`>>> True and True`
		`True`
		`>>> True and False`
		`False`

or	Return True if *either* arguments are true	`>>> True or False`
		`True`
		`>>> False or False`
		`False`

Functions: what are they?

- Why are functions useful?
  - Allow you to reuse the same code
    - Programmers are lazy!
- A block of reusable code used to perform a specific task

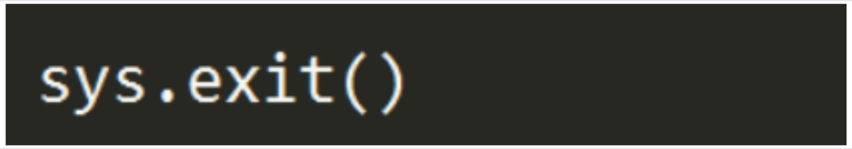
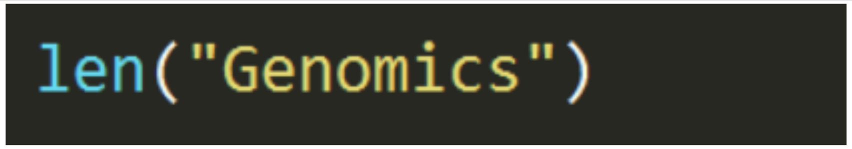


- Similar to mathematical functions, e.g., $f(x) = x^2$
- 2 types:

Built-in
Function prewritten for you
`print`: print something to the terminal
`float`: convert something to a floating point #

User-defined
You create your own functions

Functions: how can I call a function?

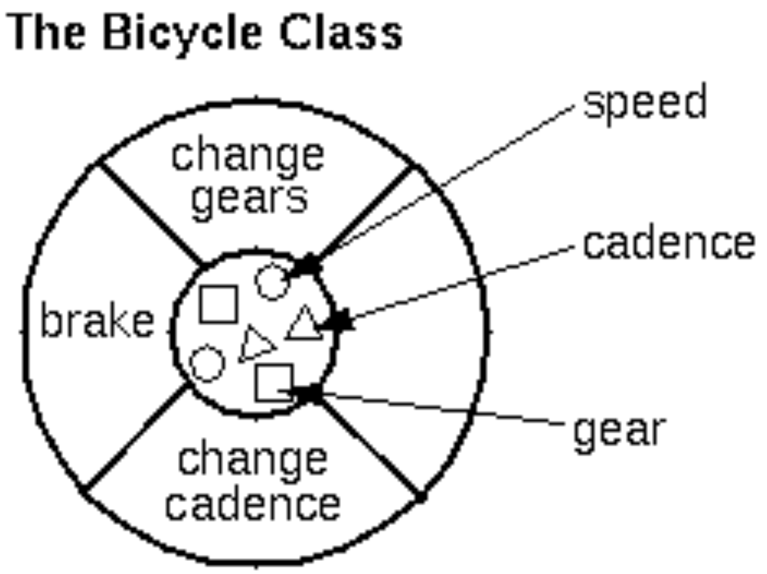
Syntax	Example	Output
Call a function that takes no arguments		
`<function_name>()`	 `sys.exit()`	
Call a function that takes argument(s)		
`<function_name>(<arg1>, <arg2>)`	 `len("Genomics")`	8

Python functions: where can I learn more?

- Python.org tutorial
  - User-defined functions: https://docs.python.org/3/tutorial/controlflow.html#defining-functions
- Python.org documentation
  - Built-in functions: https://docs.python.org/3/library/functions.html

Methods: what are they?

- First a preamble...
  - Methods are a close cousin of functions
  - For this class we'll treat them as basically the same
  - The syntax for calling a method is different than for a function
  - If you want to learn about the differences, google **object oriented programming** (**OOP**)
- Why are ~~functions~~ methods useful?
  - Allow you to reuse the same code



String methods

Syntax	Description	Example

`<str>.upper()`	• Returns the string with all letters uppercased	`>>> x = "Genomics"`
		`>>> x.upper()`

`<str>.lower()`	• Returns the string with all letters lowercased	`>>> x.lower()`
		`'genomics'`

`<str>.find(<pattern>)`	• Returns the first index of <pattern> in the string
	• Returns -1 if the if <pattern> is not found
		`>>> x.find('nom')`
		`2`

`<str>.count(<pattern>)`	• Returns the number of times <pattern> is found in the string
	• HINT: explore how .count deals with overlapping patterns
		`>>> x.count('g')`
		`0`

`<str>[<index>]`	• Returns the letter at the <index>th position	`>>> x[1]`
		`'e'`

0	1	2	3	4	5	6	7
G	e	n	o	m	i	c	s

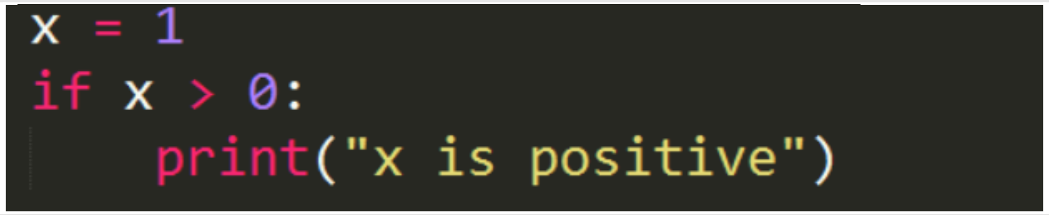
https://docs.python.org/3.4/library/stdtypes.html?#string-methods
https://docs.python.org/3/library/stdtypes.html#str

Making choices (conditional statements)

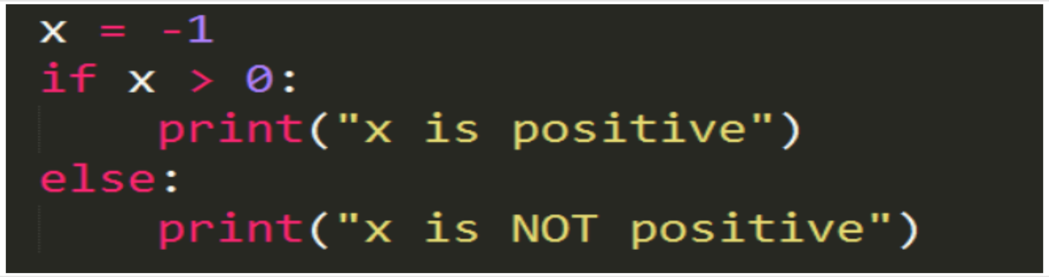
- Why is this concept useful?
  - Often we want to check if a condition is true and take one action if it is, and another action if the condition is false
  - *E.g., If the alternative allele read coverage at a particular location is high enough, annotate the position as a SNP otherwise, annotate the position as reference*

Conditional statement syntax

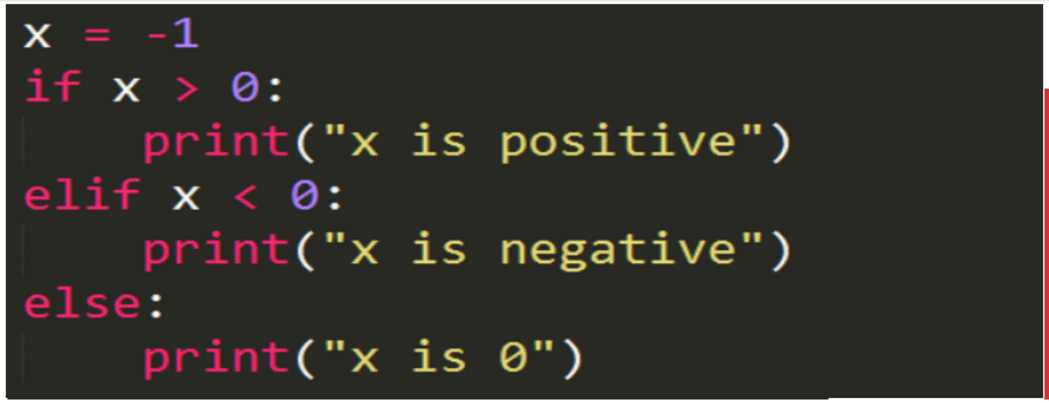
Syntax	Example	Output
If		

`if <condition>:`
    `# Do something`
	
	`x = 1`
	`if x > 0:`
	    `print("x is positive")`
		`x is positive`

If/else		

`if <condition>:`
    `# Do something`
`else:`
    `# Do something else`
	
	`x = -1`
	`if x > 0:`
	    `print("x is positive")`
	`else:`
	    `print("x is NOT positive")`
		`x is NOT positive`

If/else if/else		

`if <condition1>:`
    `# Do something`
`elif <condition2>:`
    `# Do something else`
`else:`
    `# Do something else`
	
	`x = -1`
	`if x > 0:`
	    `print("x is positive")`
	`elif x < 0:`
	    `print("x is negative")`
	`else:`
	    `print("x is 0")`
		`x is negative`
		
		Indentation matters!!!
		Indent the lines of code that belong to the same code block
		Use 1 tab

Commenting your code

- Why is this concept useful?
  - Makes it easier for--you, your future self, TAs ☺, anyone unfamiliar with your code--to understand what your script is doing
- Comments are human readable text. They are ignored by Python.
- Add comments for

  The how
  - What the script does
  - How to run the script
  - What a function does
  - What a block of code does

  The why
  - Biological relevance
  - Rationale for design and methods
  - Alternatives

TREAT YOUR CODE LIKE A LAB NOTEBOOK

Always code [and comment] as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. Code for readability.

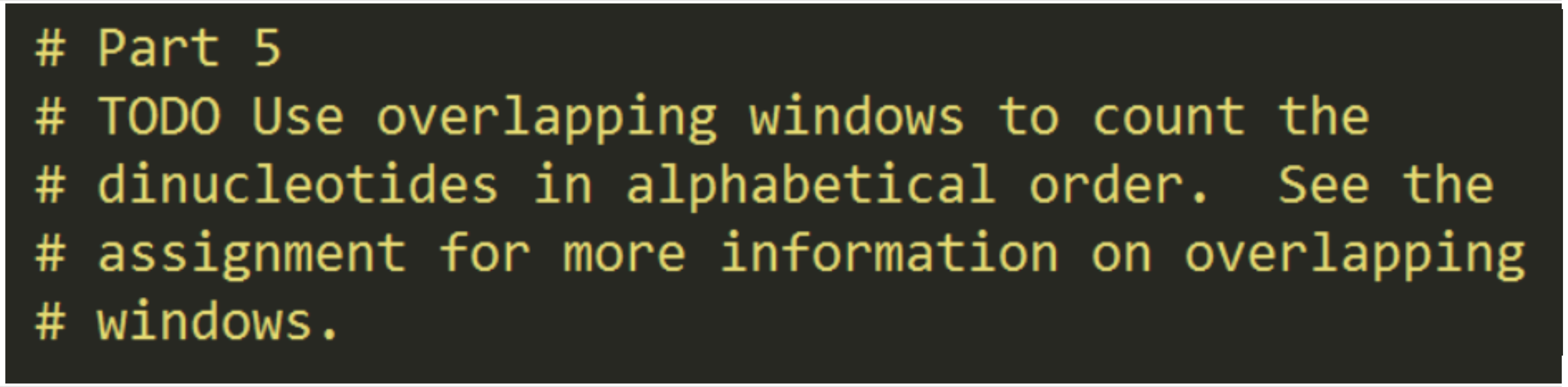
-- John Woods

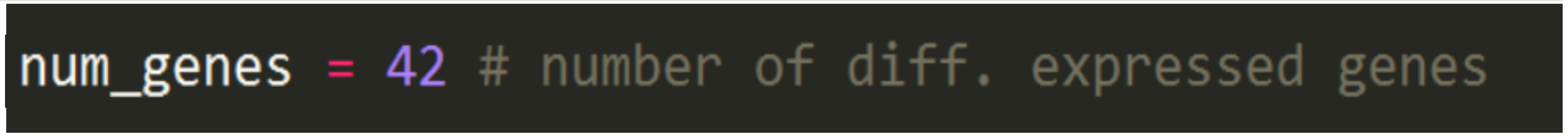
Commenting your code (cont.)

- Commenting is extremely important!
- **Points will be deducted if you do not comment your code**
- **If you use code from a resource, e.g., a website, cite it**

Comment syntax

Syntax	Example
Block comment	

`# <your_comment>`
`# <your_comment>`
	
	`# Part 5`
	`# TODO Use overlapping windows to count the`
	`# dinucleotides in alphabetical order.  See the`
	`# assignment for more information on overlapping`
	`# windows.`

In-line comment	
`<code> # <your_comment>`	 `num_genes = 42 # number of diff. expressed genes`

Python modules

- A module is file containing Python definitions and statements for a particular purpose, e.g.,
  - Generating random numbers
  - Plotting
- Modules must be imported at the beginning of the script
  - This loads the variables and functions from the module into your script, e.g.,

```
import sys
import random
```

- To access a module's features, type `<module>.<feature>`, e.g., `sys.exit()`

Random module

- Contains functions for generating random numbers for various distributions
- TIP: will be useful for assignment 1

Function	Description
`random.choice`	Return a random element from a list
`random.randint`	Return a random interger in a given range
`random.random`	Return a random float in the range [0, 1)
`Random.seed`	Initialize the (pseudo) random number generator

https://docs.python.org/3.4/library/random.html

How to repeat yourself (for loops)

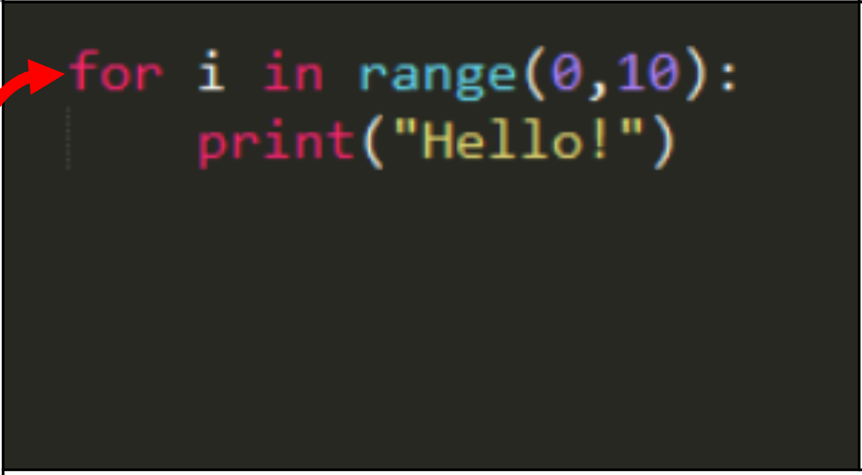
- Why is this useful?
  - Often, you want to do the same thing over and over again
    - *Calculate the length of each chromosome in a genome*
    - *Look up the gene expression value for every gene*
    - *Align each RNA-seq read to the genome*
  - A for loop takes out the monotony of doing something a bazillion times by executing a block of code over and over for you
    - Remember, programmers are lazy!
- A for loop **iterates** over a collection of things
  - Elements in a list
  - A range of integers
  - Keys in a dictionary

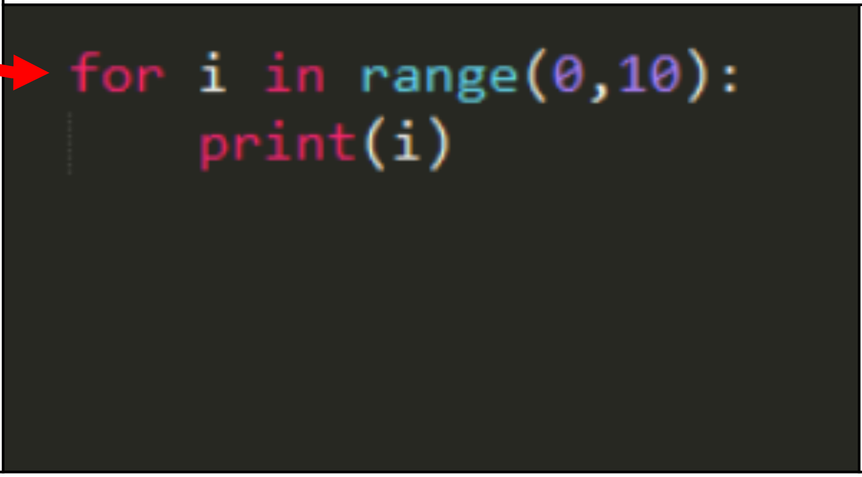
FOUR-LEAF CLOVERS ARE NOT MUTANT FREAKS
FOUR-LEAF CLOVERS ARE NOT MUTANT FREAKS
FOUR-LEAF CLOVERS ARE NOT MUTANT FREAKS
FOUR-LEAF CLOVERS ARE NOT MUTANT FREAKS
FOUR-LEAF CLOVERS ARE NOT MUTANT FR
FOUR-LEAF CLOVERS ARE NOT MUTANT FR
FOUR-LEAF CLOVERS ARE NOT MUTANT FREA
FOUR-LEAF CLOVERS ARE NOT M

Indentation matters!!!
Indent the lines of code that belong to the same code block
Use 1 tab

For loop syntax

Syntax	Example	Output

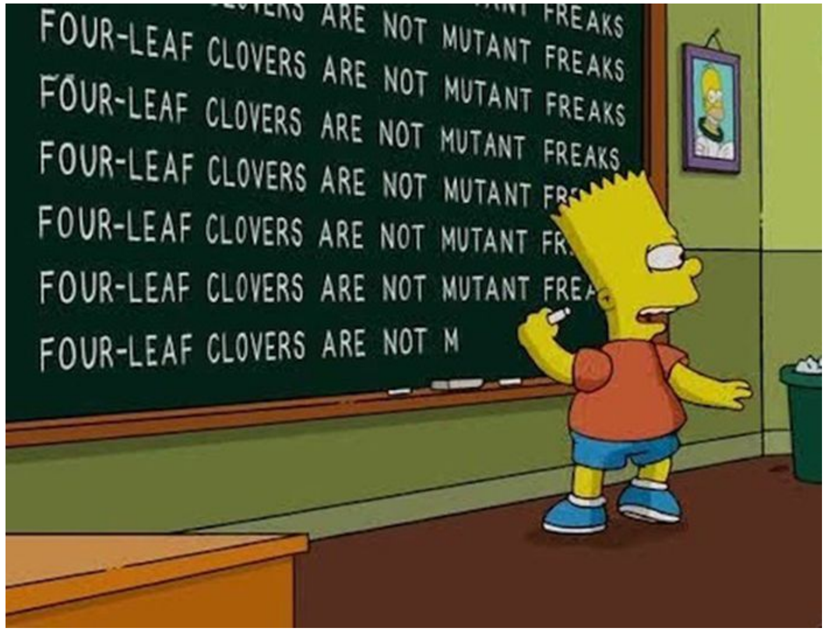
`for <counter> in <collection_of_things>:`
`    # Do something`
	
	`for i in range(0,10):`
	`    print("Hello!")`
		Hello!
		Hello!
		Hello!
		Hello!
		Hello!
		Hello!
		Hello!
		Hello!
		Hello!
		Hello!

	
	`for i in range(0,10):`
	`    print(i)`
		0
		1
		2
		3
		4
		5
		6
		7
		8
		9

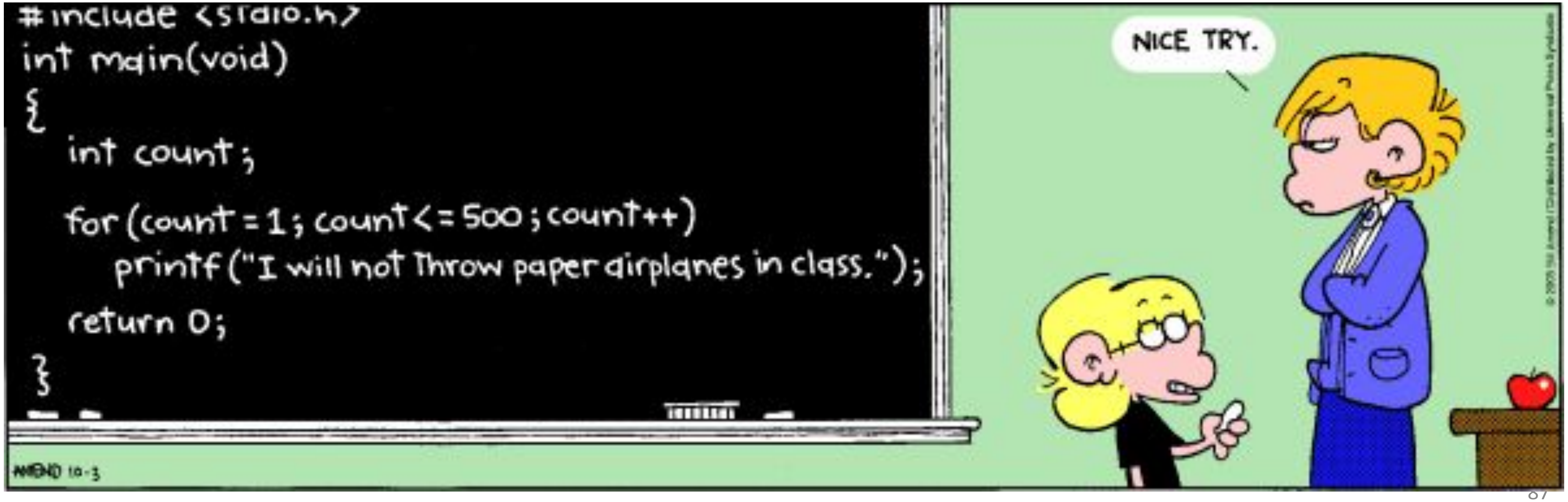
- The <counter> variable is the value of the current item in the collection of things
  - You can ignore it
  - You can use its value in the loop
- All code in the for loop's code block is executed at each iteration
- TIP: If you find yourself repeating something over and over, you can probably convert your code to a for loop!

Which option would you rather do?

A


ARE NOT MUTANT FREAKS
FOUR-LEAF CLOVERS ARE NOT MUTANT FREAKS
FOUR-LEAF CLOVERS ARE NOT MUTANT FREAKS
FOUR-LEAF CLOVERS ARE NOT MUTANT FREAKS
FOUR-LEAF CLOVERS ARE NOT MUTANT FR
FOUR-LEAF CLOVERS ARE NOT MUTANT FR
FOUR-LEAF CLOVERS ARE NOT MUTANT FREA
FOUR-LEAF CLOVERS ARE NOT M

B



```
#include <stdio.h>
int main(void)
{
   int count;
   for (count = 1; count <= 500; count++)
      printf("I will not Throw paper airplanes in class.");
   return 0;
}
```

How to repeat yourself (cont.)

- For loops have a close cousin called **while loops**
- The major difference between the 2
  - For loops repeat a block of code a predetermined number of times (really, a collection of things)
  - While loops repeat a block of code as long as an expression is true
    - e.g., while it's snowing, repeat this block of code
    - While loops can turn into **infinite while loops** → the expression is never false so the loop never exits. Be careful!
    - See http://learnpythonthehardway.org/book/ex33.html for a tutorial on while loops