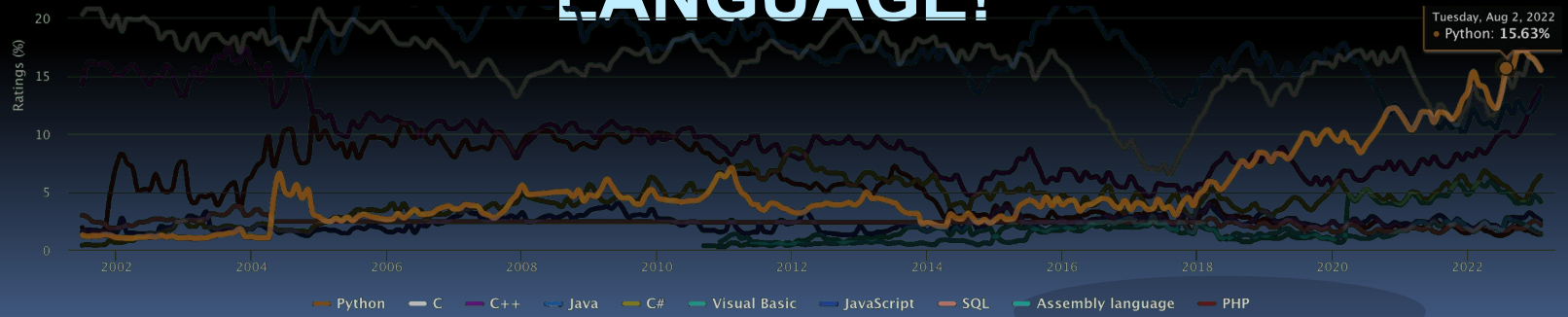
UC Berkeley
Teaching
Professor
Dan Garcia

CS10
The Beauty and Joy of Computing

Python I – Basics

PYTHON NOW THE MOST POPULAR CODING LANGUAGE!



www.tiobe.com/tiobe-index

Computational Thinking (1/2)

- BJC's coding wasn't about learning Snap*!*
- Instead, it was about learning computational thinking.
  - Using abstraction (removing detail and providing generalization via parameters)
  - Understanding the value of a precise specification
    - Example: HW3 blocks
  - Introduction to Software Engineering
    - Design, implement, test.. (and iterate)
- Every CS course @ Cal aims to teach concepts, not languages…

Computational Thinking

- Thinking about how solutions scale, parallelize, and generalize and foreseeing unintended consequences.
  - Scale: Orders of growth for algorithms.
  - Parallelize: Some tasks parallelize more nicely than others. Using functional style is incredibly helpful!
  - Generalization: Reusing software that already exists to serve your needs; these include libraries
  - Unintended Consequences: How could anyone use your tool in ways you normally wouldn't consider?

Why do we use Snap! ?

- Two factors of programming:
  - The conceptual solution to a problem/challenge.
  - Solution syntax in a programming language

BJC tries to isolate and strengthen the first

- Snap*!* has huge pedagogical benefits!
  - More or less removes worry of syntax
  - Code reads like pseudo-code
    - Spaces, inputs interspersed
  - No need to memorize command names!
  - Multimedia comes for free (sounds, events, graphics)
  - Manage complexity through OOP -- sprites

Why learn Python?

- Python is easy to read!
- Very little syntax
- Multi-paradigm!
- Scripting language
  - Implement programs quickly

```
def fact(n):
    if (n < 1):
        return 1
    else:
        return n*fact(n-1)
```

- Widely used as a teaching tool
- Powerful and Fast, with hundreds of community supported code libraries
- Spark allows VERY EASY distributed computing

Python in the World

Whenever you need help…

- Online documentation well-written & intuitive!

4. More Control Flow Tools

Besides the `while` statement just introduced, Python knows the usual control flow statements known from other languages, with some twists.

4.1. `if` Statements

Perhaps the most well-known statement type is the `if` statement. For example:

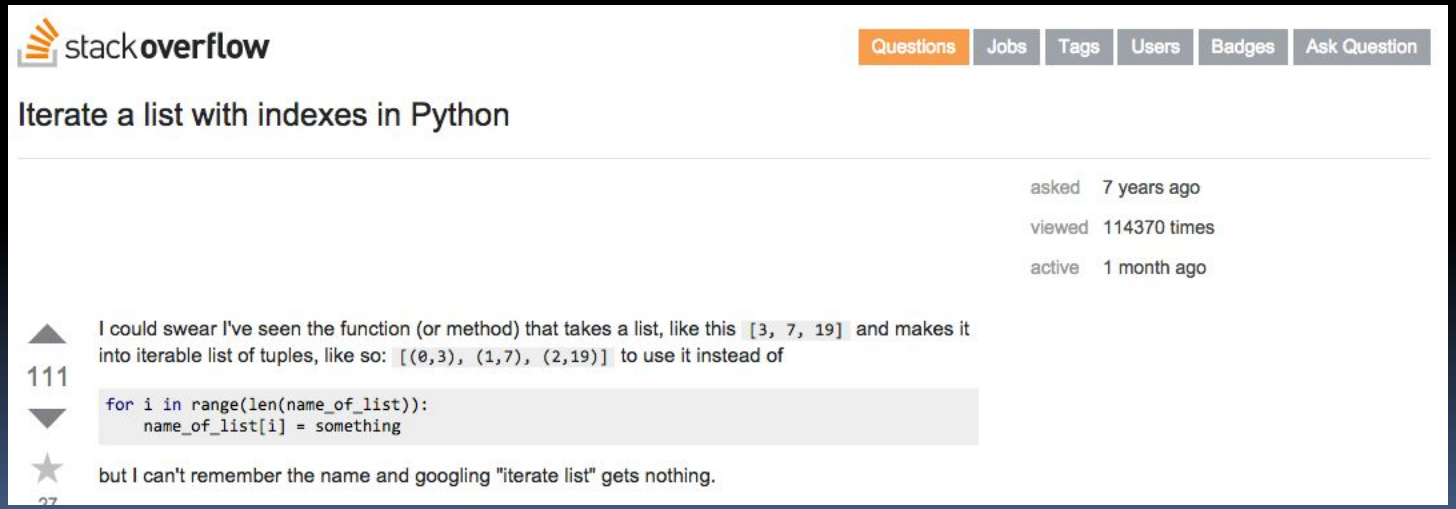
```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword '`elif`' is short for 'else if', and is useful to avoid excessive indentation. An `if` ... `elif` ... `elif` ... sequence is a substitute for the `switch` or `case` statements found in other languages.

docs.python.org

Whenever you need help…

- Public forum boards (like Piazza!)
- There's a lot of Python programmers out there
- Our first course for majors teaches it (CS61A), so pretty much every Cal CS student knows it.



Drawbacks of Python…

- It's still slow compared to Java, C, C++
  - much faster than Snap*!* though…
- Very little error checking and reporting, due to minimalist syntax (it's dynamically typed)
  - Data types exist, but they don't have to be declared (similar to Snap!)
  - Other languages, like C, require that the programmer specifies the type of their variables... and you aren't allowed to change it!
- Variable scopes sometimes tricky
- Absence from mobile computers & browsers

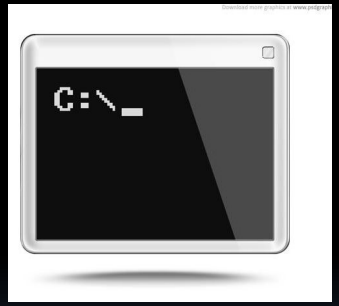
Zen of Python

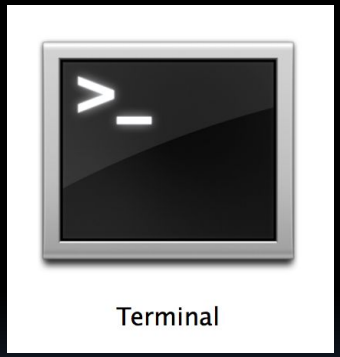
```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

Getting Started!

- Opening the Interpreter
- Command line action: Hello World
- Loading a program





Let's give it a try!

From Snap! to Python: Variables

Snap!

Python

```
>>> foo=5
>>>
```

You can see why learning Python might be hard…
= is assignment, not equality testing!

From Snap! to Python: Variables

Snap!

Python

```
>>> foo=5
>>> foo
5
```

In practice, we'd use: `print(foo)`

From Snap! to Python: Operators I

Snap!	Python
	`+`
	`-`
	`*`
	`/`

However: `5//2` is `2` (rounds down)

Other arithmetic operators available!

`2 ** 3` (is 8, `**` is exponent)

From Snap! to Python: Operators II

Snap!	Python
	`<`
	`>`
	`==`

Notice the difference:

`foo=3` vs `foo==3`

(the most common thing to forget)

From Snap! to Python: Operators III

Snap!	Python
	`and`
	`or`
	`not`

Note: Operator precedence often needs to be controlled with brackets

e.g.: `(a == b) or (c > d)`

From Snap! to Python: Operators IV

Snap!

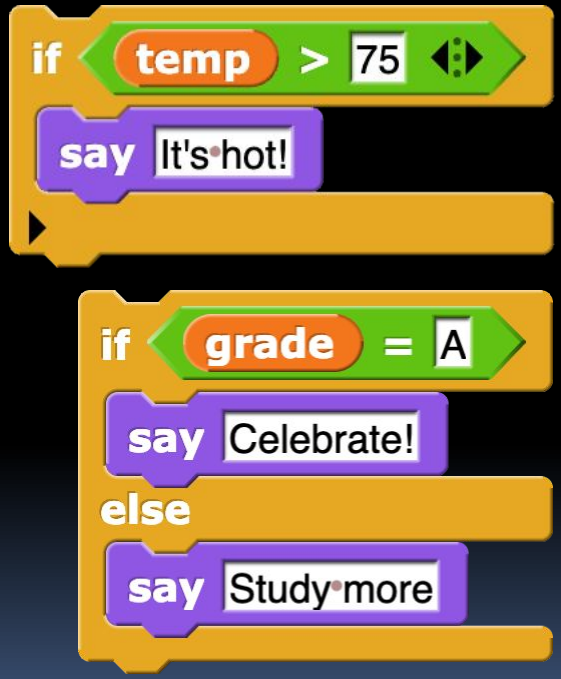
Python

```
%
"hello "+"world"
len("fish")
"world"[1]
>>> "world"[2:4]
"rl"
```

New: slicing (and zero-indexing)

From Snap! to Python: Conditionals

Snap!



Python

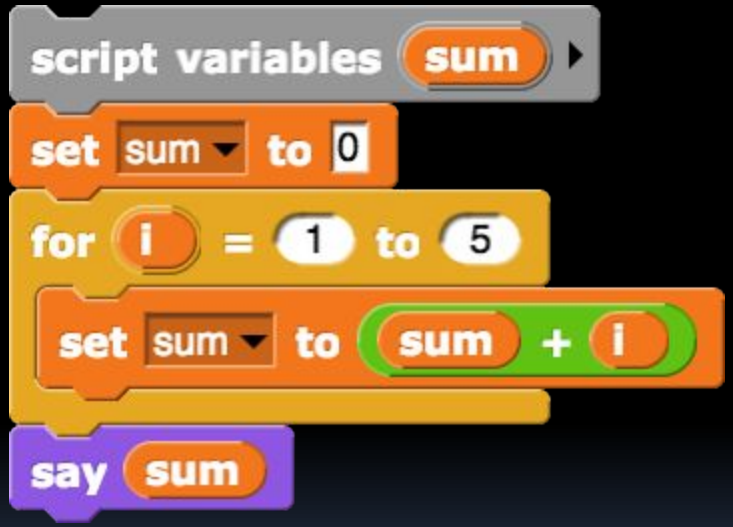
```
if (temp > 75):
    print("It's hot!")

if (grade == "A"):
    print("Celebrate!")
else:
    print("Study more")
```

Important: Indentation is a requirement!

From Snap! to Python: Loops I

Snap!



Python

```
sum = 0
for i in range(1,6):
    sum = sum + i
print(sum)
```

Explanation: `range(1,6)` returns something the `for` construct can iterate over: *1,2,3,4,5*

From Snap! to Python: Loops II

Snap!

Python

```
bottles = 99
while bottles > 0:
    print(str(bottles)+ " beer")
    bottles -= 1
print("No more beer!")
```

Python `while` is similar to `repeat until` except:

`repeat until` ends on `True` (go until bad)

`while` ends on `False` (go while good)

Clicker question

- Will the following code cause an error?

```
def mystery(array):
    if array[0] == array[len(array)]:
        print("The first and last items are the same!")
    else:
        print("The first and last items are NOT the same!")
```

a) Yes
b) No

L15: Will the following cause an error?

Yes

No

```
def mystery(array):
    if array[0] == array[len(array)]:
        print("The first and last items are the same!")
    else:
        print("The first and last items are NOT the same!")
```

Python Beauty: Iterators:

`for` iterates over lists

```
data = [2, 4, 6, 8]
sum = 0
for n in data:
    sum = sum + n
print("The sum is:" + str(sum))
```

Python Beauty: Speaking of list…

Lists can contain anything! (just like Snap*!* lists)

```
data = [1,2,3]
data = ["Hello","World"]
data = ["Hello",1,"World",2.3]
data = [[1,2],[3,4],[5,6]]
data = [(1,2),[3,4],5,6]
```

Python Beauty: List Comprehension I

How to work with lists…

```
>>> data = [0,1,2,3,4,5]
>>> data[2]
2
>>> data[2:4]
[2, 3]
>>> data[4:]
[4, 5]
>>> data[:4]
[0, 1, 2, 3]
```

Python Beauty: List Comprehension II

How to work with lists…

```
fruits = ['Banana', 'Apple', 'Lime']
>>> [fruit.upper() for fruit in fruits]
['BANANA', 'APPLE', 'LIME']

>>> [10*i for i in range(5) if i != 2]
[0, 10, 30, 40]
```

Anyone recognize “map” and “keep” here?

“keep” and “map” in one line!

Python Beauty: List Comprehension III

How to work with lists…

```
>>> data = [0,1,2,3,4,5,6,7,8,9]
>>> data.reverse()
>>> data.sort()
>>> data.count(4)
1
>>> data.insert(5,123)
>>> data.append(10)
```

This is called “Dot Notation”

Lists are objects and this is the way to call the methods of an object.

Summary: similarities to Snap!, different name

- `a % b` is 

, `a ** b` 

- `A+B` is 

(numbe 

(strings), or 
(if lists)
- `for ite` 

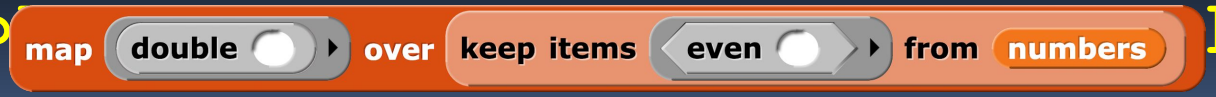
`data` is 

- `print` 

- `len` is 

and 

- `A[0]` is 

`foo"]` is
- `"a" in data` is
- `[doub` 

`]` is

Summary: key differences from Snap!

- `a = 3` is 

- `a == 3` is 

- `while` is opposite logic from 

- Sequences are 0-indexed
- Ranges are *exclusive* of right number; e.g. `range(1,3)` is 

- `foo(a,b,c)` All parameters comma-separated at the end of a function, parentheses around them
- No spaces in variable and function names, many symbols are illegal too
- You need quotes around strings: `"hello"`
- Procedures defined with: `def … return`
- Colons at the end of some lines: `if even(n):`

Summary: New to Python

- `a // b` returns `a` divided by `b`, then rounded down
- `data.reverse()` Variables are objects, methods called through ”dot notation” … but Snap*!* ha

- `range()` Provides quick sequence of #s … but Snap*!* has

- `str()` Converts arguments to strings
  - Other types and their converters in Python II
- Whitespace *matters!*
- Print vs return; `foo` could be

```
>>> foo()
42
```

```
def foo():
    return 42
```

```
def foo():
    print(42)
```

Python Beauty: More Information

- How to work with…
  - Functions
  - Dictionaries
  - Tuples
  - APIs
- …and much more in the Python II lecture!