



National University
of Singapore

NUS | Computing

IT5001 Software Development Fundamentals

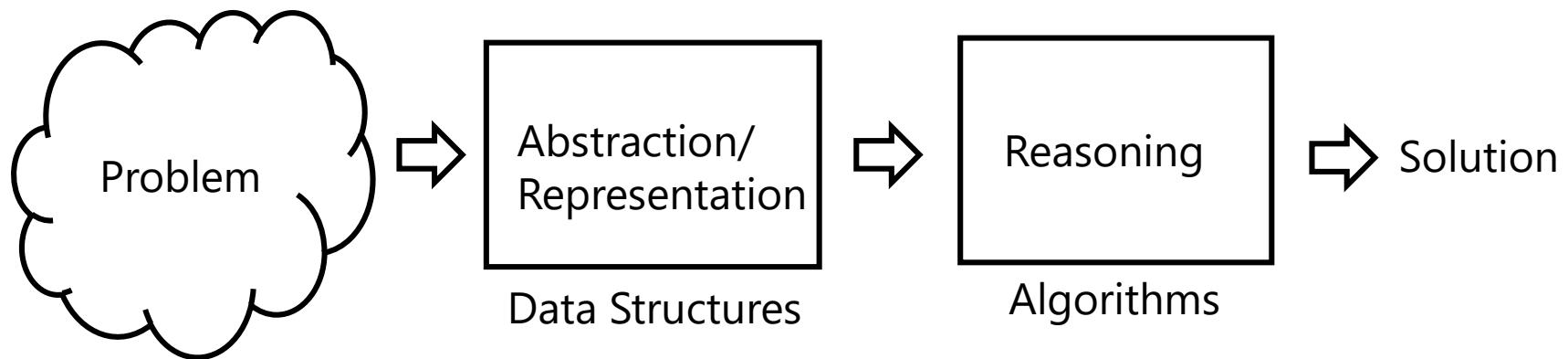
1. Introduction

Rajendra Prasad Sirigina

August-2023

Software Development

Steps involved in problem-solving:



Programming languages provide tools to:

1. build data structures
2. do reasoning

Representation

- Numbers
- Strings
- Arrays
- Multi-dimensional Arrays
- Graphs and Trees

What is an
Algorithm?



Algorithm (noun.)

Word used by programmers when...
they do not want to explain what they did.

Algorithms

- Named for al-Khwārizmī (780-850)
 - Persian mathematician
- Many ancient algorithms
 - Multiplication: Rhind Papyrus
 - Babylon and Egypt: ~1800BC
 - Euclidean Algorithm: Elements
 - Greece: ~300BC
 - Sieve of Eratosthenes
 - Greece: ~200BC



Algorithm

- An **algorithm** is a well-defined computational procedure consisting of *a set of instructions*, that takes some value or set of values as *input*, and produces some value or set of values as *output*.

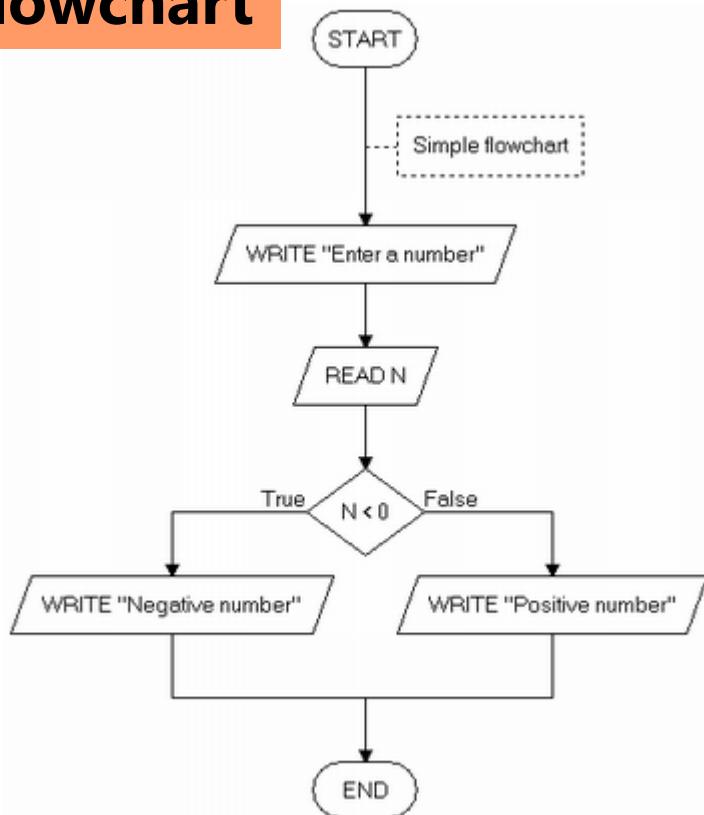


'Algorithm' stems from 'Algoritmi', the Latin form of al-Khwārizmī, a Persian mathematician, astronomer and geographer.
Source: <http://en.wikipedia.org/wiki/Algorithm>

Algorithm

- Ways of representing an algorithm:

Flowchart



Pseudocode

get a number

read the number and store it in N

if N is less than zero

 print negative number

else

 print positive number

end If

Algorithm Vs Program

Algorithm

- Ideas

get a number

read the number and store it in N

if N is less than zero

 print positive number

else

 print negative number

end If

Program

- The final code on a machine

```
x = input('Enter a number:')
```

```
N = int(x)
```

```
if N < 0:
```

```
    print('Negative Number')
```

```
else:
```

```
    print('Positive Number')
```

Writing a Program

- Requires
 - Understanding of language issues
 - Syntax and Semantics
 - Data Structures
 - Representation of the problem
 - Reasoning ability
 - Algorithms

An overview of



Why are we learning Python?

- Clear and readable syntax
- Intuitive
- Natural expression
- Powerful
- Popular & Relevant
- Example: Paypal
 - ASF XML Serialization
 - C++
 - 1580 lines
 - Python
 - 130 lines



Who uses Python?

- Google
- Red Hat
- Dropbox
- Rackspace
- Twitter
- Facebook
- Raspberry Pi
- NASA
- CERN
- ITA
- Yahoo!
- Walt Disney
- IBM
- Reddit
- YouTube

Python Program without Learning

```
a = 1  
b = 2  
c = a + b  
if c < 0:  
    print('Yes')  
  
else:  
    print('No')
```

Intuitive!



Pseudo Code to Program

Algorithm

get a number

read the number and store it in N

if N is less than zero

 print positive number

else

 print negative number

end If

Program

```
x = input('Enter a number:')
```

```
N = int(x)
```

```
if N < 0:
```

```
    print('Negative Number')
```

```
else:
```

```
    print('Positive Number')
```

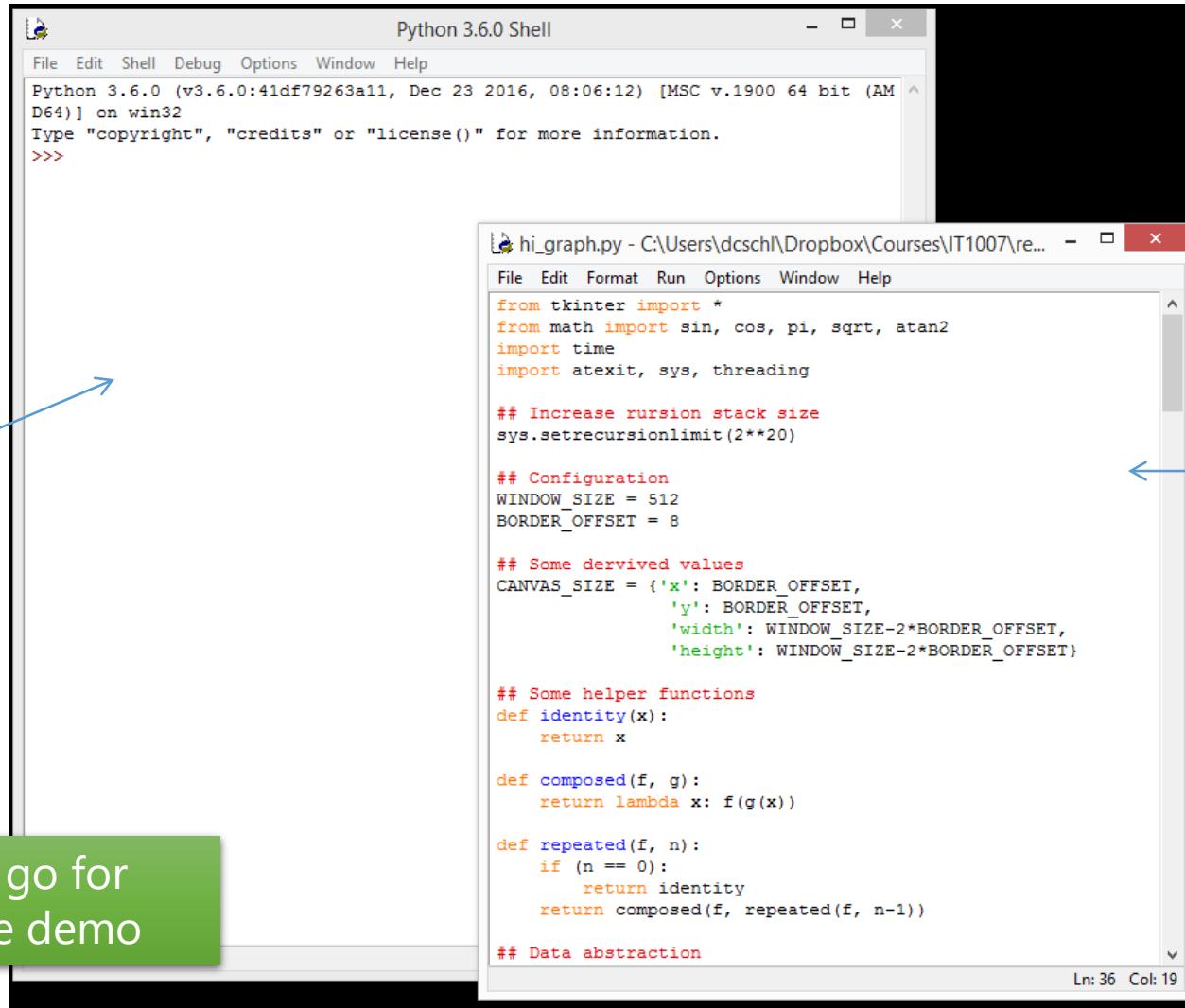
Automatic Vs. Manual Transmission: Which is the best choice for you?



The Environment: IDLE

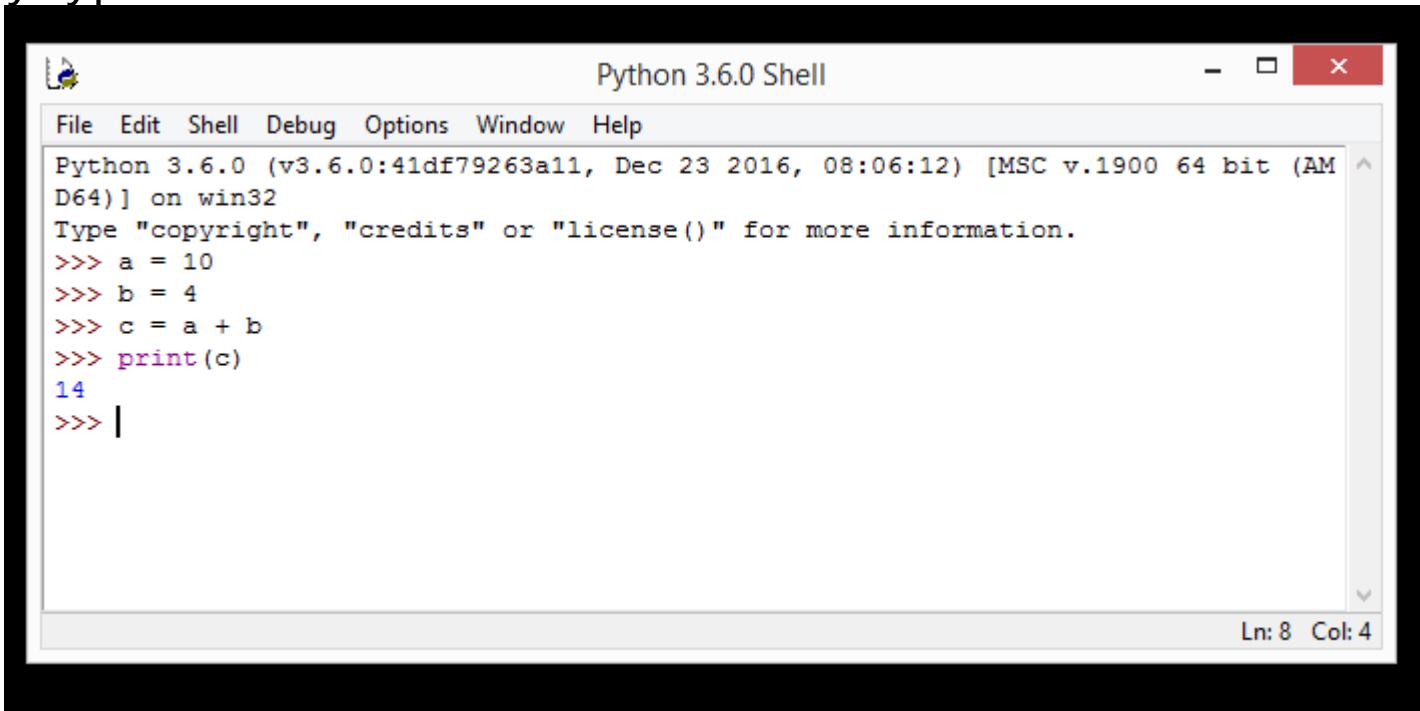
- **IDLE** as an IDE
 - IDLE:
 - Integrated development and learning environment
 - IDE:
 - Integrated development environment
 - Edit, run and debug
- Other tools
 - Jupyter notebook
 - PyCharm
 - Spyder
 - Visual Studio Code, etc.

A Screenshot of IDLE



You can

- Directly type into the console



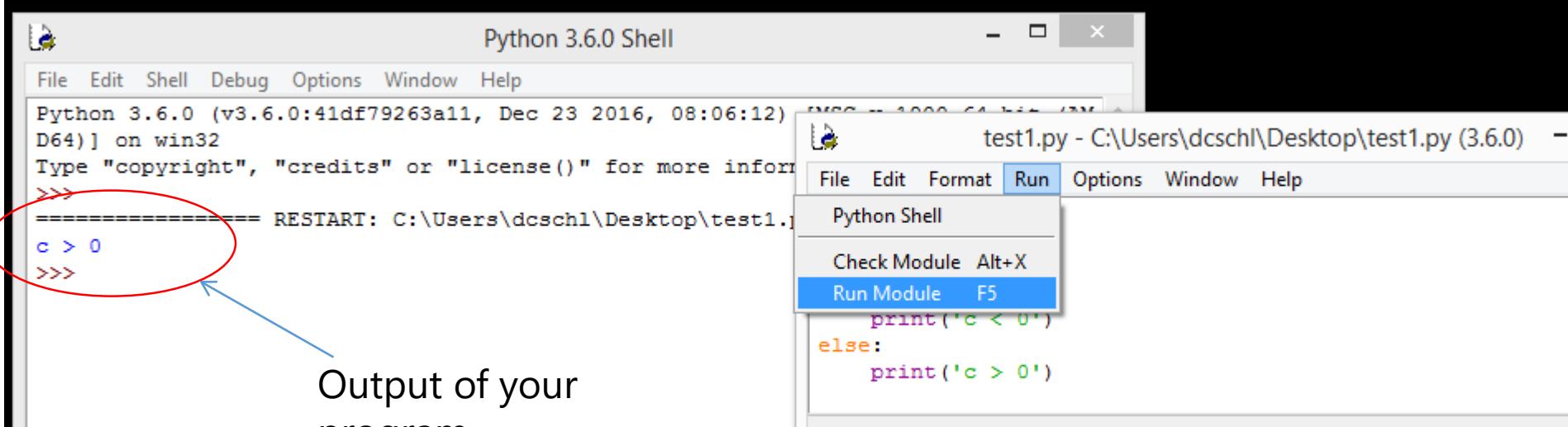
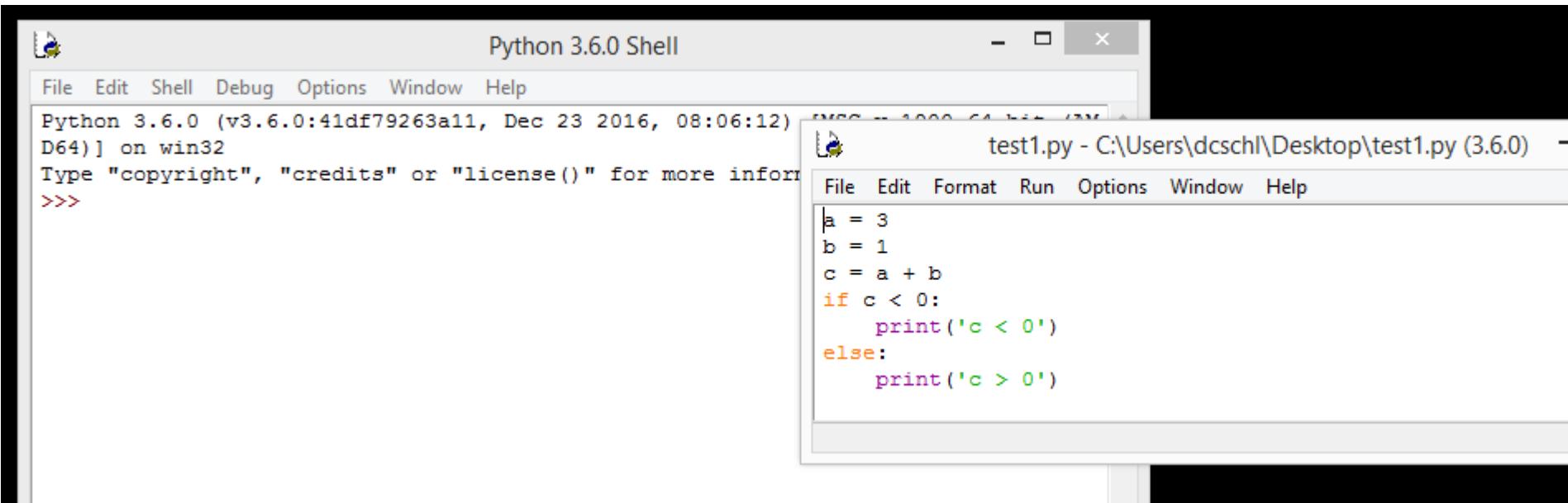
A screenshot of the Python 3.6.0 Shell window. The title bar reads "Python 3.6.0 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the Python interpreter's prompt and some code:

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit (AM  
D64)] on win32  
Type "copyright", "credits" or "license()" for more information.  
>>> a = 10  
>>> b = 4  
>>> c = a + b  
>>> print(c)  
14  
>>> |
```

The status bar at the bottom right shows "Ln: 8 Col: 4".

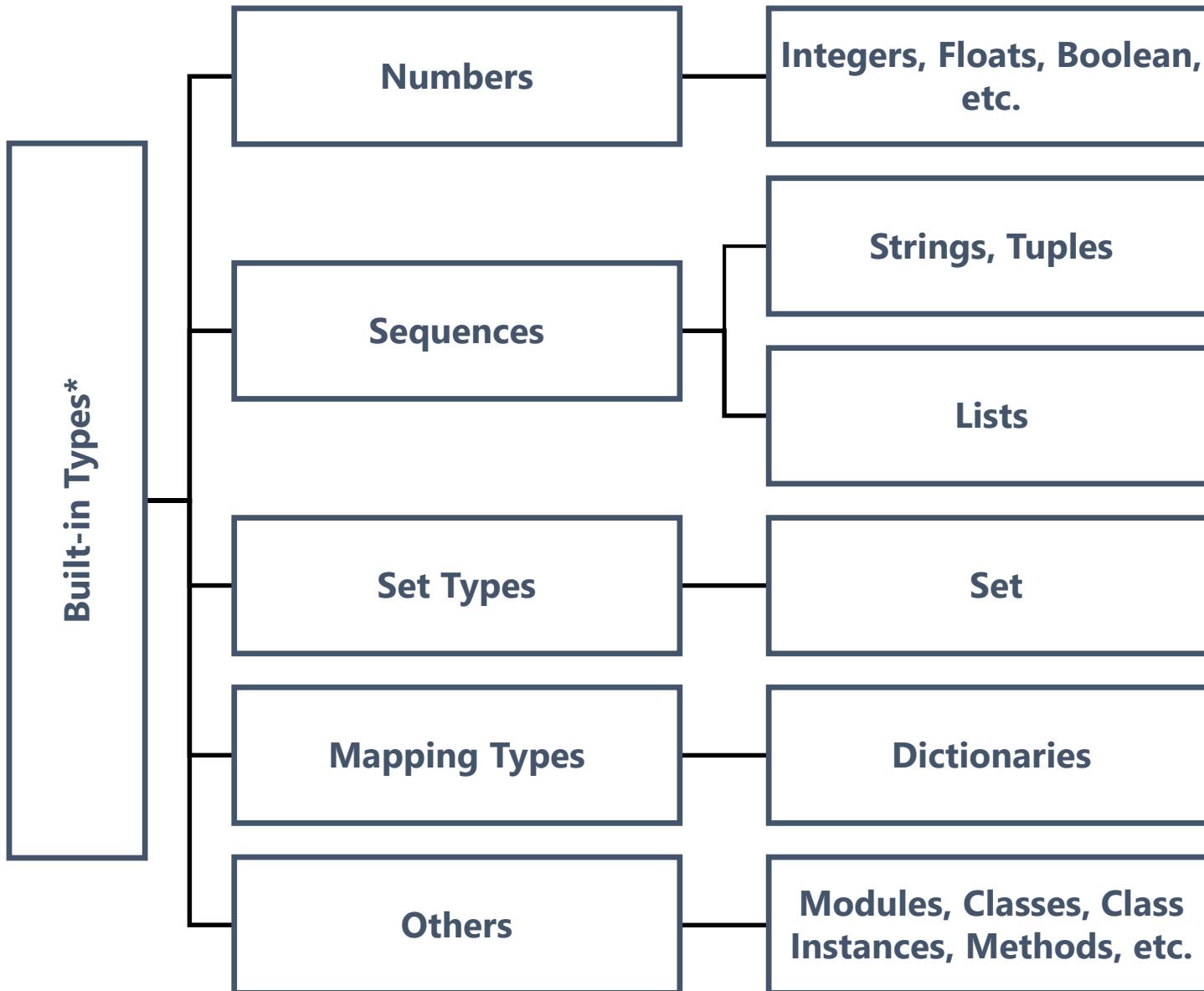
- In which, we **seldom** do this

Or Run a file



Output of your

Representation



*List is incomplete

Built-in Types

Type	Description	Immutable?
int	Integer	Yes Primitive types
float	Floating-point number	Yes
bool	Boolean value	Yes
string	Character String	Yes
list	Sequence of objects	No
tuple	Sequence of objects	Yes
set	Unordered set of distinct objects	No
dict	Associative Mapping (dictionary)	No

Immutable: Cannot modify

Numbers: Numeric Types

- Integers: *int*
- Floats: *float*
 - Stores real numbers as binary fractions
 - 64-bit double precision*
- Self Exercise:
 - Convert the decimal numbers 0.375 and 0.1 to binary. What do you learn from the conversion?

```
>>> 2  
2  
>>> type(2)  
<class 'int'>  
>>> 2.0  
2.0  
>>> type(2.0)  
<class 'float'>
```

Boolean Type

- Following are evaluated to **False**

- False : Keyword
- None : Keyword
- 0, 0.0, 0j : Value Zero (*int, float, complex*)
- "" : Empty String
- [] : Empty List
- {} : Empty Dictionary
- range(0) : Iterator
- set() : Empty set

Will learn them in
subsequent weeks

- Rest are evaluated to **True**

```
>>> bool(0.0)      >>> bool(10)
False                True
>>> bool(0)        >>> bool('hi')
False                True
>>> bool({})       >>> bool([1,2])
False                True
>>> bool(None)     >>> bool(1)
False                True
>>> bool(True)      >>> bool({1,2})
True                 True
>>> bool(False)    >>> bool(True)
False                True
>>> bool([])        >>> bool(int)
False                True
>>> bool('')        >>> bool('')
False                False
>>> bool("")
```

Identifiers

- User-defined names for objects
 - Can enhance readability

- Rules

- First character should be an alphabet or underscore (_)
- Other characters can be numbers and underscore
- Special characters not allowed
- Names are case sensitive

assignment operation

```
>>> int_var = 2
>>> _int_var = 2
>>> 2int_var = 2
SyntaxError: invalid syntax
>>> int@var = 2
SyntaxError: can't assign to operator
>>>
```

```
>>> x = 2
>>> X = 4
>>> print(x)
2
>>> print(X)
4
```



Multiple Assignments

```
>>> x, y = 1, 2  
>>> x  
1  
>>> y  
2  
>>> x, y = y, x  
>>> x  
2  
>>> y  
1
```

```
>>> x, y, z = 1, 2, 3  
>>> x  
1  
>>> y  
2  
>>> z  
3  
>>> x, y, z = z, y, x  
>>> x  
3  
>>> y  
2  
>>> z  
1
```

Python is Dynamically Typed

- No need to declare object type
- Interpreter automatically recognizes the type

```
>>> x = 2
>>> print(x)
2
>>> type(x)
<class 'int'>
>>> x = 2.0
>>> print(x)
2.0
>>> type(x)
<class 'float'>
>>> x = 2+0j
>>> print(x)
(2+0j)
>>> type(x)
<class 'complex'>
>>> x = True
>>> type(x)
<class 'bool'>
```

- Keywords cannot be used as identifiers
- Builtins can be used as variables
 - but don't do it



builtins

```
>>> import builtins
>>> dir(__builtins__)
['ArithmetError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundException', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', '__build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hash', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

```
>>> print = 2
>>> print('hi')
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    print('hi')
TypeError: 'int' object is not callable
```

Keyword Types

Type	Example
Value Keywords	<i>True, False, None</i>
Operator Keywords	<i>and, or, not, in, is</i>
Control Flow Keywords	<i>if, else, elif</i>
Iteration Keywords	<i>for, while, break, continue, else</i>
Structure Keywords	<i>def, class, with, as, pass, lambda</i>
Returning Keywords	<i>return, yield</i>
Import Keywords	<i>import, from, as</i>
Exception-handling Keywords	<i>try, except, raise, finally, else, assert</i>
Asynchronous Programming Keywords	<i>async, await</i>
Variable Handling Keywords	<i>del, global, nonlocal</i>

```
>>> import keyword  
>>> keyword.iskeyword('del')  
True
```

```
>>> del = 3  
SyntaxError: invalid syntax
```

Operators

- Arithmetic Operators
- Logical Operators
- Equality Operators
- Comparison Operators

Arithmetic Operators

Operation	Result
<code>x + y</code>	sum of x and y
<code>x - y</code>	difference of x and y
<code>x * y</code>	product of x and y
<code>x / y</code>	quotient of x and y
<code>x // y</code>	floored quotient of x and y
<code>x % y</code>	remainder of <code>x / y</code>
<code>-x</code>	x negated
<code>+x</code>	x unchanged
<code>x ** y</code>	x to the power y

```
>>> 2+3  
5  
>>> 2.0+3.0  
5.0  
>>> 2-3  
-1  
>>> 2*3  
6  
>>> 2.0*3.0  
6.0  
  
>>> 2/3  
0.6666666666666666  
>>> 3/2  
1.5  
>>> 3//2  
1  
>>> 3%2  
1  
>>> 3**2  
9  
  
>>> -2  
-2
```

Mixed mode arithmetic

- If operands are of different types?
- Narrower (less general) and Wider (more general) Types
 - Float is wider (more general) than integer
 - All integers are floats but not vice-versa
- Narrower type is promoted to wider type
 - Integer is promoted to float

```
>>> 2+3.0  
5.0  
>>> 2.0-3  
-1.0  
>>> 3.0/2  
1.5  
>>> 3/2.0  
1.5  
>>> 3//2.0  
1.0  
>>> 3.0//2  
1.0  
>>> 3.0**2  
9.0  
>>> 3**2.0  
9.0  
>>> 3.0%2  
1.0  
>>> 3%2.0  
1.0
```

Comparison Operators

Operation	Meaning
<	strictly less than
<=	less than or equal
>	strictly greater than
>=	greater than or equal
==	equal
!=	not equal
is	object identity
is not	negated object identity

```
>>> 2<3  
True  
>>> 3<2  
False  
>>> 2 <= 3  
True  
>>> 2 > 3  
False  
>>> 3 >= 3  
True  
>>> 2 == 2  
True  
>>> 2 != 3  
True  
>>> 2 != 2  
False  
>>> False == False  
True  
>>> False == True  
False
```

What is the difference between `==` and `is`?

is operator

```
>>> x = 2  
>>> y = 2  
>>> x is y  
True
```

```
>>> x = 2  
>>> y = 3  
>>> x is y  
False
```

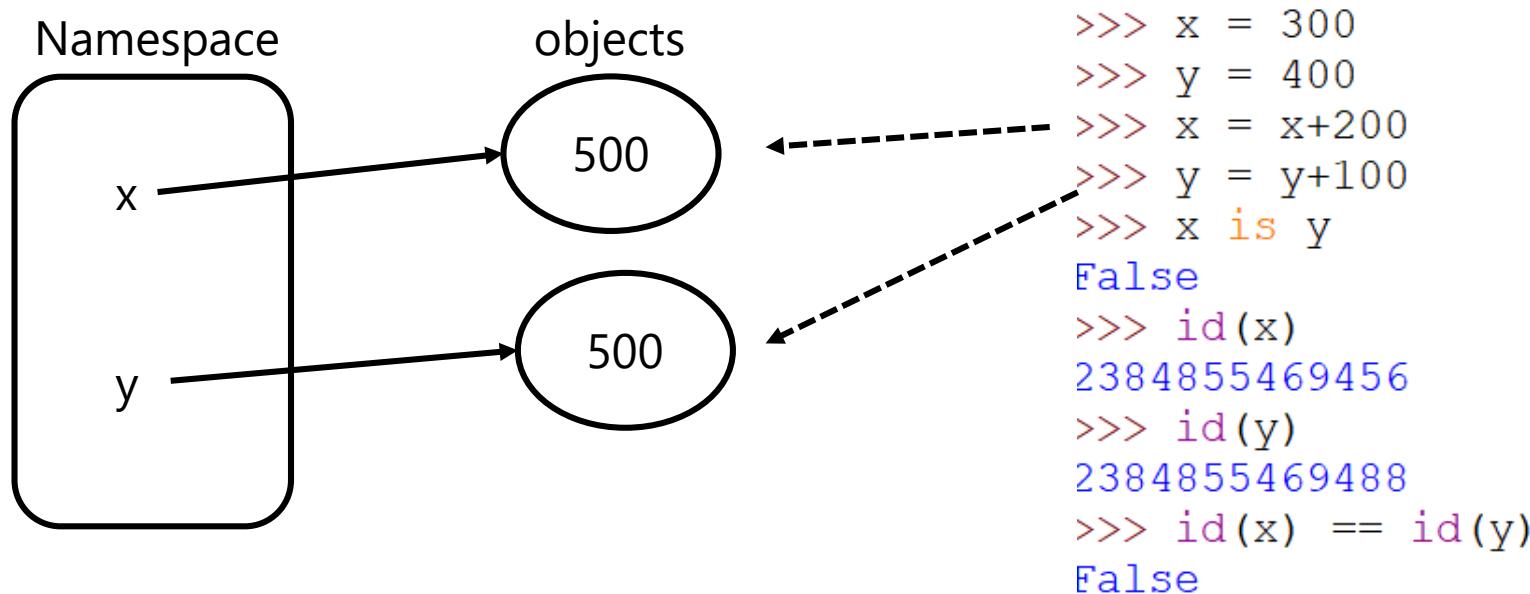
```
>>> x = 4  
>>> y = 3  
>>> x = x + 1  
>>> y = y + 2  
>>> x is y  
True
```

```
>>> x = 400  
>>> y = 300  
>>> x = x+100  
>>> y = y+200  
>>> x is y  
False
```

```
>>> x = 400  
>>> y = 300  
>>> x = x+100  
>>> y = x  
>>> x is y  
True
```

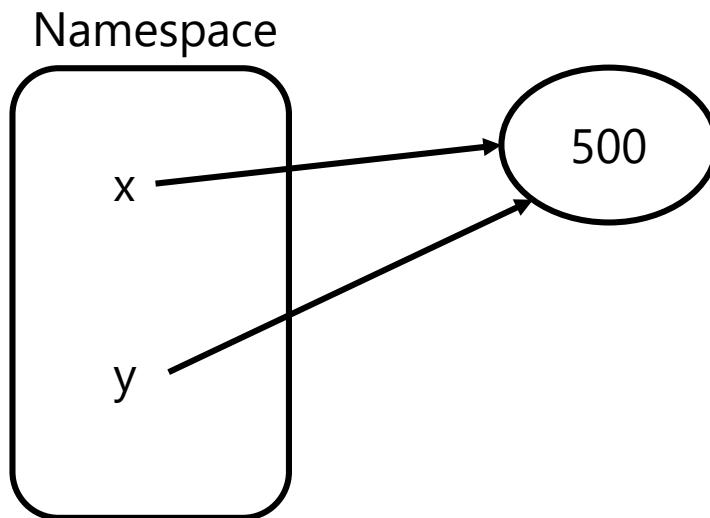
is operator

- Binary operator
 - Returns true if identity of both operands is same
- What is identity?



Keyword *is*

- Binary operator
 - Returns true if identity of both operands is same
- What is identity?



```
>>> x = 400
>>> y = 300
>>> x = x+100
>>> y = x
>>> x is y
True
>>> id(x)
2384855469456
>>> id(y)
2384855469456
>>> id(x) == id(y)
True
```

Logical/Boolean Operators

Operator	Operation	Result	Remark
and (conditional and)	x and y	If x is false, then x, else y	<ul style="list-style-type: none">• Short-circuit operator• Only evaluates the second argument if the first one is true
or (conditional or)	x or y	If x is false, then y, else x	<ul style="list-style-type: none">• Short-circuit operator• Only evaluates the second argument if the first one is false
not (unary negation)	not x	If x is false, then <i>True</i> , else <i>False</i>	<ul style="list-style-type: none">• Low priority than non-Boolean operators• Ex: <code>not a == b</code> means <code>not (a==b)</code>

and Operator

x **and** y: if x is false, then x, else y

```
>>> 1 and 0  
0  
>>> 0 and 1  
0
```

```
>>> x = 3  
>>> y = 2  
>>> x and y  
2
```

```
>>> x = 0  
>>> y = 2  
>>> x and y  
0  
>>> x = False  
>>> x and y  
False
```

```
>>> print and input  
<built-in function input>  
>>> bool(print)  
True
```

```
>>> False and True  
False  
>>> True and False  
False
```

or Operator

x **or** y: if x is false, then y, else x

```
>>> (1 or 0)  
1  
>>> 0 or 1  
1
```

```
>>> x = 3  
>>> y = 2  
>>> x or y  
3
```

```
>>> x = 0  
>>> y = 2  
>>> x or y  
2
```

```
>>> False or True  
True  
>>> True or False  
True
```

```
>>> x = 1  
>>> y = 2  
>>> (0 or 0) and (x or y)  
0
```

not Operator

not x: If x is false, then *True*, else *False*

```
>>> not 2  
False  
>>> not 0  
True  
,
```

Augmented Assignment Operators

Operation	Description
$x += y$	$x = x + y$
$x *= y$	$x = x * y$
$x /= y$	$x = x / y$
$x //= y$	$x = x //y$
$x **= y$	$x = x**y$

```
>>> x = 1  
>>> x+= 1  
>>> x  
2
```

Expressions

- Expressions
 - A piece of syntax evaluated to some value
 - Combination of operators and operands
 - Value is an expression
 - Variable is an expression
 - Combination of values, variables and operators is also an expression

```
>>> 1  
1  
>>> x = 1  
>>> x  
1  
>>> x + 1*2  
3  
.
```

Standard IO: Input

- Input

```
>>> input('Enter an integer: ')
Enter an integer: 2
'2'
```

- Type Casting

- Conversion of one type to other
- Example:

```
>>> x = input('Enter an integer: ')
Enter an integer: 2
>>> x
'2'
>>> type(x)
<class 'str'>
>>> x = int(x)
>>> x
2
>>> type(x)
<class 'int'>
```

```
>>> x = float(x)
>>> x
2.0
>>> type(x)
<class 'float'>
```

Standard IO: Output

```
>>> print()  
  
>>> print('IT 5001')  
IT 5001  
>>> x = 2  
>>> print(x)  
2  
  
  
>>> print('This is \nIT5001')  
This is  
IT5001
```

Precedence

Operator	Description
()	Parenthesis
**	Exponentiation
+x, -x	x unchanged, x negated
*, /, //, %	Multiplication, division, floor division, remainder
+, -	Addition, Subtraction
in, not, <, <=, >, >=, ==, !=	Membership, comparison and identity tests
not x	Boolean NOT
and	Boolean AND
or	Boolean OR

Precedence

(4-5) * 3 - 7 % 4 ** 2 / 3

-1 * 3 - 7 % 4 ** 2 / 3

-1 * 3 - 7 % 16 / 3

-3 - 7 % 16 / 3

-3 - 7 / 3

-3 - 2.333334

-5.233334

Operator

()

**

+x, -x

*, /, //, %

+, -

in, not, <, <=, >, >=, ==, !=

not x

and

or

Equal precedence:
Association is from left to right

Strings

- Strings are **indexed sequence of characters**
- Example Strings
 - It is IT5001
 - It's IT5001
 - "It is IT5001," said Alice
 - C:\new\IT5001

Strings

- Single quotes:

- Example

- It is IT5001
 - "It is IT5001," said Alice
 - It's IT5001

```
>>> 'It is IT5001'  
'It is IT5001'  
>>> 'It is IT5001'  
'It is IT5001'  
>>> '"It is IT5001," said Alice'  
'"It is IT5001," said Alice'  
  
>>> 'It\'s IT5001'  
"It's IT5001"
```

escape character



Strings

- Double quotes

- Example:

- It is IT5001
 - It's IT5001
 - "It's IT5001," said Alice.

```
>>> "It is IT5001"  
'It is IT5001'  
>>> "It's IT5001"  
'It's IT5001'  
  
>>> "\"It is IT5001, \" said Alice"  
'"It is IT5001," said Alice'
```

escape character

Strings

- Triple Quotes and Triple Double Quotes
 - Doesn't require escape character for single quote and double quotes within strings
 - Support multiline strings

```
>>> """It is IT5001," said Alice. So, it's IT5001."""
    "It is IT5001," said Alice. So, it's IT5001.'
```

String Manipulations

- String Operators
- Built-in String Functions
- String Indexing and Slicing
- Built-in String Methods
- String Formatting

String Operators

Operator	Operation	Result	Example
+	$x + y$	Concatenates strings x and y	'This is ' + 'IT5001' = 'This is IT5001'
*	$x * c$	A new string with 'c' copies of string x, where c is integer	'Hi'*2 = 'HiHi'
in	$x \text{ in } y$	Returns True if string x is in string y	'Hi' in 'Hi IT5001' → True
not in	$x \text{ not in } y$	Returns True if string x is not in string y	'Hi' not in 'Hi IT5001' → False

String Operators

```
>>> s = 'ba'  
>>> t = 'ck'  
>>> s+t  
'back'  
  
>>> t = s + 'na'*2  
>>> t  
'banana'
```

```
>>> w = 'banana'  
>>> s = (w + '')*2  
>>> print(s)  
bananabanana  
>>> s = (w + ' ') *2  
>>> s  
'banana banana '  
  
>>> 'b' in t  
True  
>>> 'z' in t  
False
```

Built-in String Functions

Function	Return Value	Example
len()	Length of the string	len('Hi') = 2
chr(<i>i</i>)	A string representing a character whose Unicode point is the integer <i>i</i> , $0 < i < 1114111$ - Returns a single character string for an input integer	chr(123) = '{'
ord()	ASCII value of character (string with) - Returns integer value for an input single character string	ord('{') = 123
str()	Returns string representation of an object	str(2.5) = '2.5'

Lexicographical Ordering

```
>>> 'apple' > 'banana'  
False  
>>> 'apple' > 'Banana'  
True  
>>> 'banana' > 'bananb'  
False  
>>> 'apple' > 'applee'  
False  
>>> t = 'banana'  
>>> 'c' < t  
False
```

Unicode of Characters

>>> ord('A')	>>> ord('9')
65	57
>>> ord('B')	>>> ord('0')
66	48
>>> ord('Z')	>>> ord('1')
90	49
>>> ord('a')	>>> ord('9')
97	57
>>> ord('b')	>>> chr(65)
98	'A'
>>> ord('z')	>>> chr(66)
122	'B'

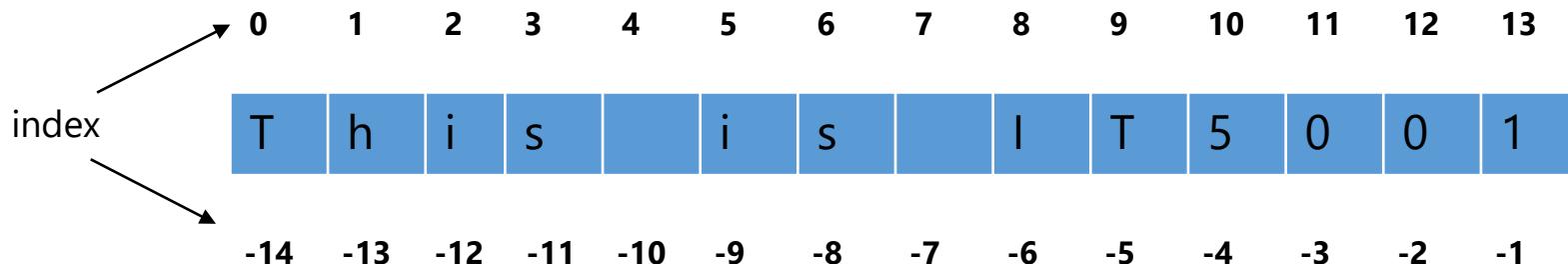
lexicographical ordering: first the first two letters are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two letters are compared, and so on, until either sequence is exhausted.

String Indexing and Slicing

- Strings are represented as compact arrays

```
string_example = 'This is IT5001'
```

Indexing:



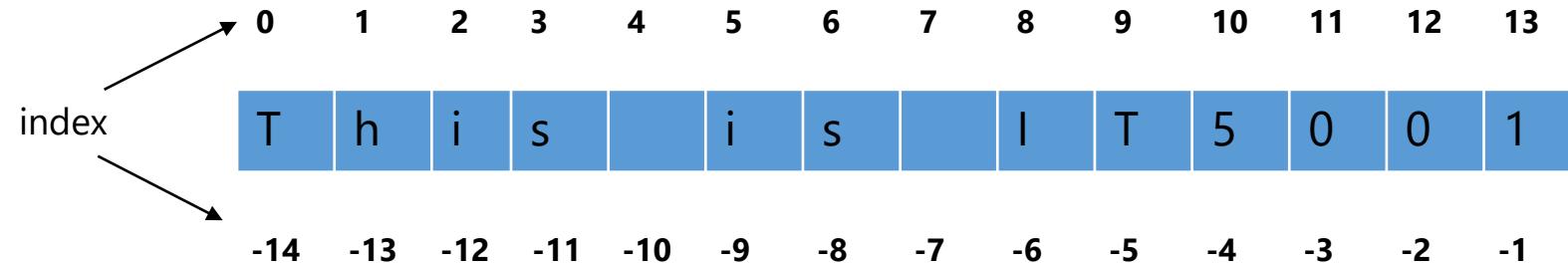
Slicing:

```
string_example[start : end : stride]
```

String Indexing and Slicing

```
string_example = 'This is IT5001'
```

Indexing:

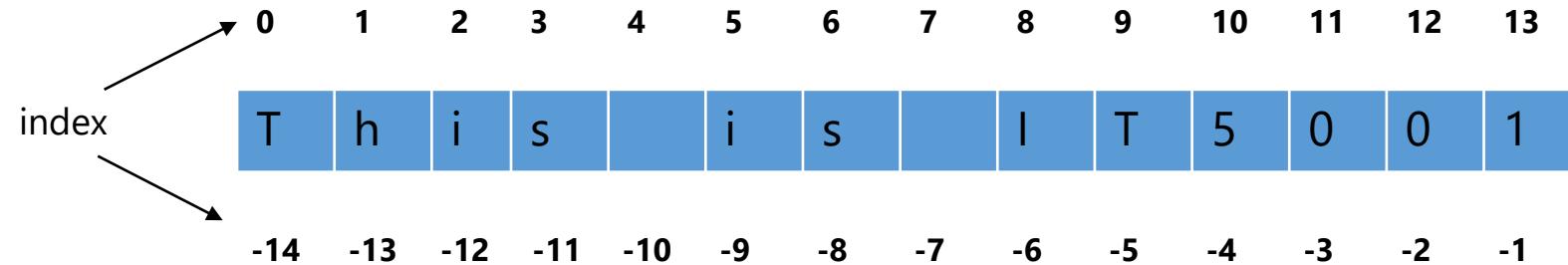


```
>>> string_example = 'This is IT5001'  
>>> string_example[0:3:2]  
'Ti'  
>>> string_example[-1]  
'1'  
>>> string_example[1:len(string_example)]  
'his is IT5001'
```

String Indexing and Slicing

```
string_example = 'This is IT5001'
```

Indexing:



```
>>> string_example[-12:-4]
'is is IT'
>>> string_example[-12:-4:2]
'i si'
```

Immutability of Strings

```
>>> string_example = 'This is IT5001'  
>>> string_example[1] = 'i'  
Traceback (most recent call last):  
  File "<pyshell#14>", line 1, in <module>  
    string_example[1] = 'i'  
TypeError: 'str' object does not support item assignment
```

String Methods

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

- Case Conversion
 - upper, lower, title, etc.

```
>>> 'abcd'.upper()
'ABCD'
>>> 'ABCD'.lower()
'abcd'
>>> 'abcd'.title()
'Abcd'
```

f-strings

- f-strings
 - Strings prefixed with 'f'
 - 'f' stands for formatted strings
 - Expressions can be embedded in strings
 - Expressions evaluated at run time.
 - Contains replacement fields, delimited by curly braces

```
>>> module_code = "IT5001"
>>> module_name = "Software Development Fundamentals"
>>> f"Welcome to {module_code} : {module_name}"
'Welcome to IT5001 : Software Development Fundamentals'

>>> print(f'23/2')
23/2
>>> print(f'{23/2}')
11.5
```

Raw Strings

- Raw Strings
 - Strings prefixed with literal 'r'

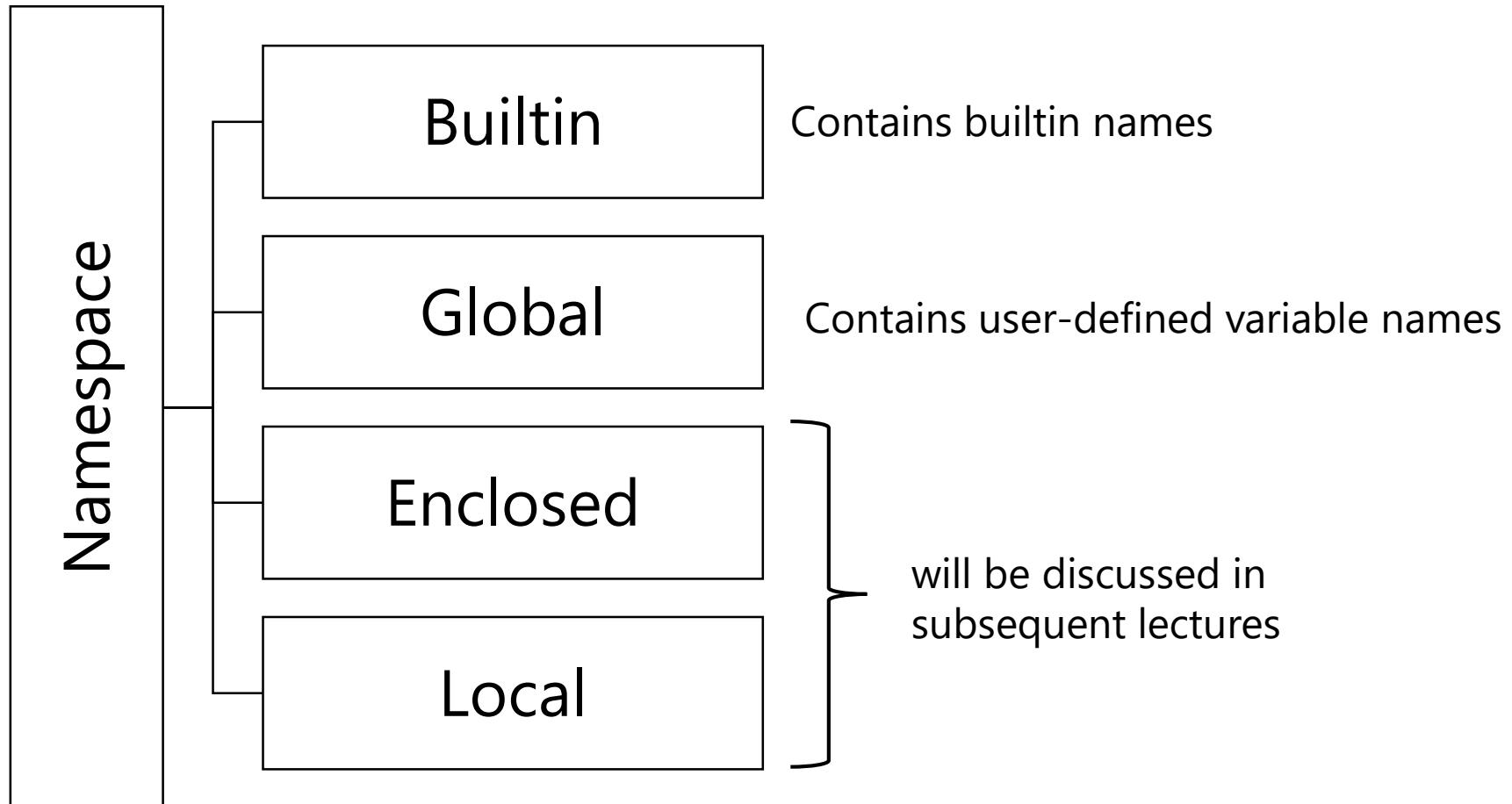
```
>>> print('This is \nIT5001')
This is
IT5001
>>> print(r'This is \nIT5001')
This is \nIT5001
```

Conclusion

- Numeric and Boolean Types
- Operators and Precedence
- Expressions and Statements
- Strings, String Operators, String Functions, and String Methods
- Immutability
- **Next Class:** Libraries and User-defined Functions

Miscellaneous Namespaces

Namespaces



Builtin Namespace

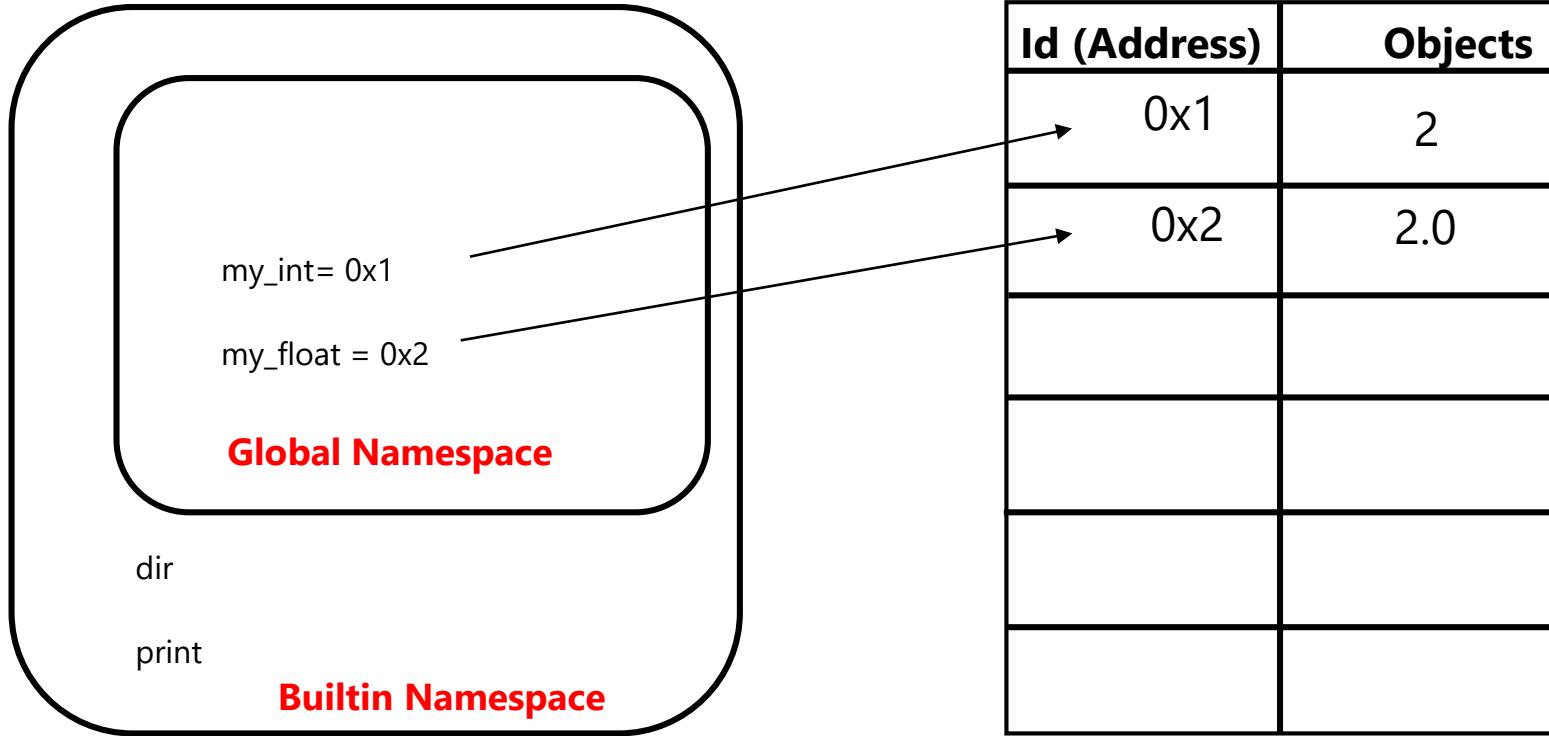
- Contains names of `_builtin_` module:
 - Datatypes
 - Int, float, etc.
 - Functions
 - print, input, etc.
 - Exceptions
 - NameError, SyntaxError, etc.
- Check `dir(__builtins__)`
- Will be created (destroyed) when Python interpreter starts (closes)
- What if you want to use a name in builtins?

Global Namespace

```
>>> print(globals())
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
<class '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__':
{}, '__builtins__': <module 'builtins' (built-in)>}
>>> my_int = 2
>>> print(globals())
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
<class '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__':
{}, '__builtins__': <module 'builtins' (built-in)>, 'my_int': 2}
>>> del my_int
>>> print(globals())
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
<class '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__':
{}, '__builtins__': <module 'builtins' (built-in)>}
```

How are objects stored?

Heap Memory



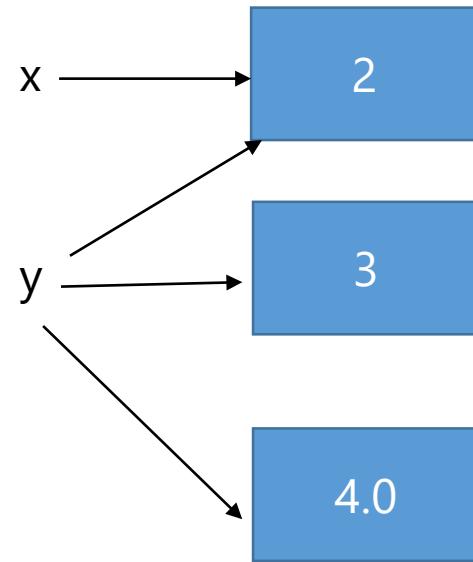
Interpreter first searches names in Global Namespace

If name is not there in global namespace, searches in builtin namespace

If name is not in builtin namespace, throws 'NameError'

How are objects stored?

- $x = 2$
- $y = 2$
- $y = 3$
- $y = 4.0$

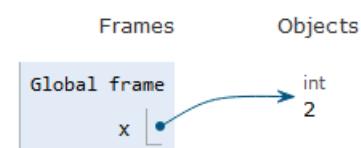


How are objects stored?

Python 3.6
(known limitations)

```
1 x = 2
2
3 y = 2
4
5 x = x+1
```

[Edit this code](#)



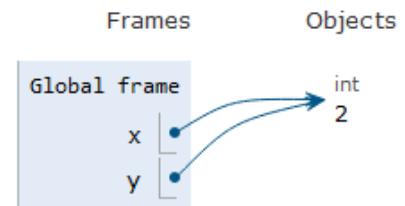
1

Demo: pythontutor.com

Python 3.6
(known limitations)

```
1 x = 2
2
3 y = 2
4
5 x = x+1
```

[Edit this code](#)

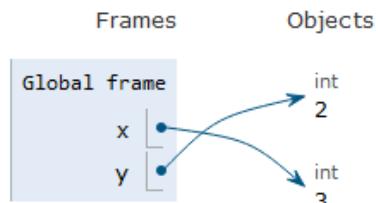


2

Python 3.6
(known limitations)

```
1 x = 2
2
3 y = 2
4
5 x = x+1
```

[Edit this code](#)



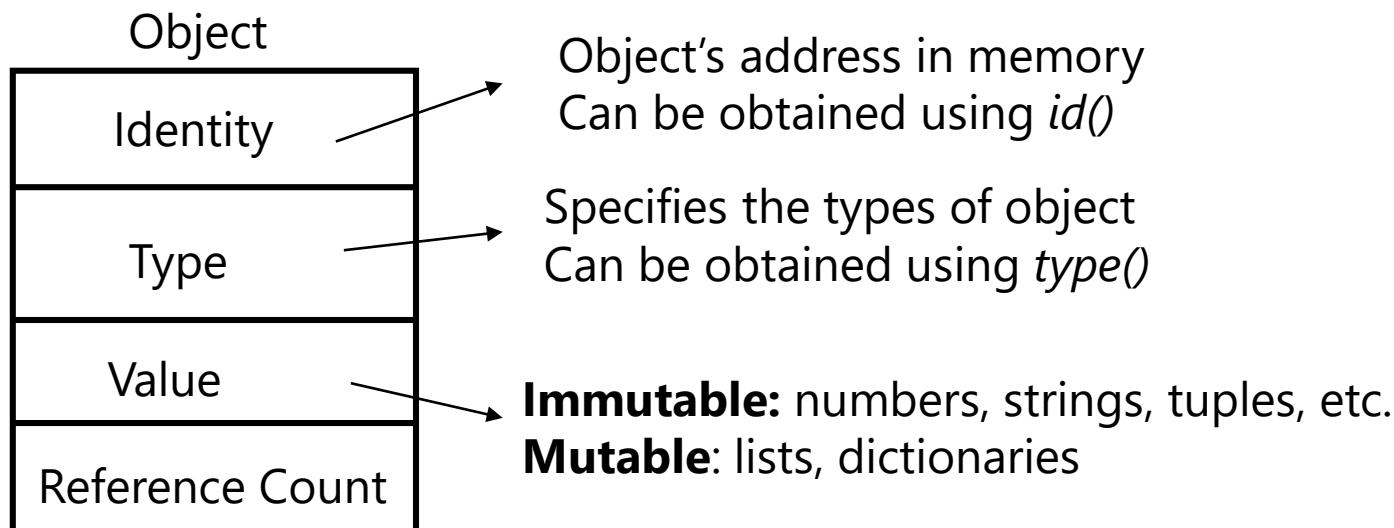
3

Memory Management

- Python does memory management automatically
- Private heap to store objects
- Memory management depends on object type

Data Model: Objects, Values, and Types

- Objects are Python's abstraction for data
- Data in program is represented by objects and relation between objects



```
>>> x = 2
>>> y = 2
>>> x is y
True
>>> x = 3
>>> y = x
>>> x is y
True
>>> x = 4
>>> y = 2
>>> x is y
False
>>> x = 400
>>> y = 300
>>> x += 100
>>> y += 200
>>> x is y
False
>>> x = 4
>>> y = 3
>>> x+= 1
>>> y+=2
>>> x is y
True
>>> |
```

Why is this behaviour?



NUS | Computing

National University
of Singapore

IT5001 Software Development Fundamentals

2. Functions (Callable Units)

Rajendra Prasad Sirigina

August-2023

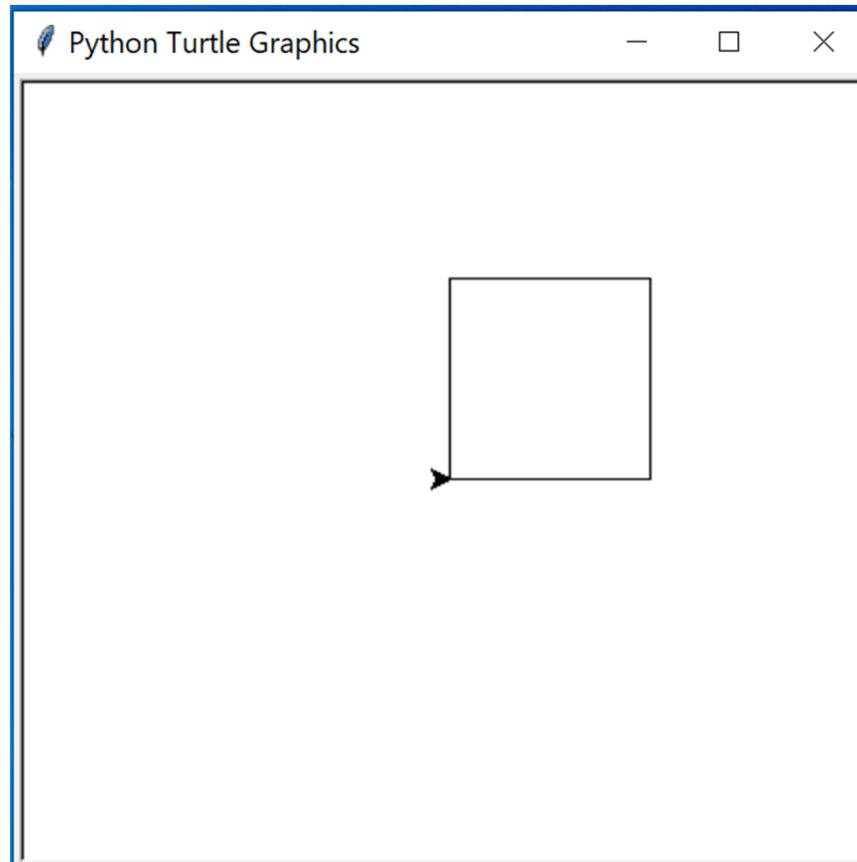
Agenda: Functions

- Python Modules and Packages
 - Namespaces
 - Scope
- User-defined functions
 - Arguments
 - Positional
 - Keyword
 - Default
 - Return Statement
 - print vs return
 - Function Tracing
 - Namespaces
 - Local namespace
- Pure Functions

Why Functions?

Example: Drawing a square

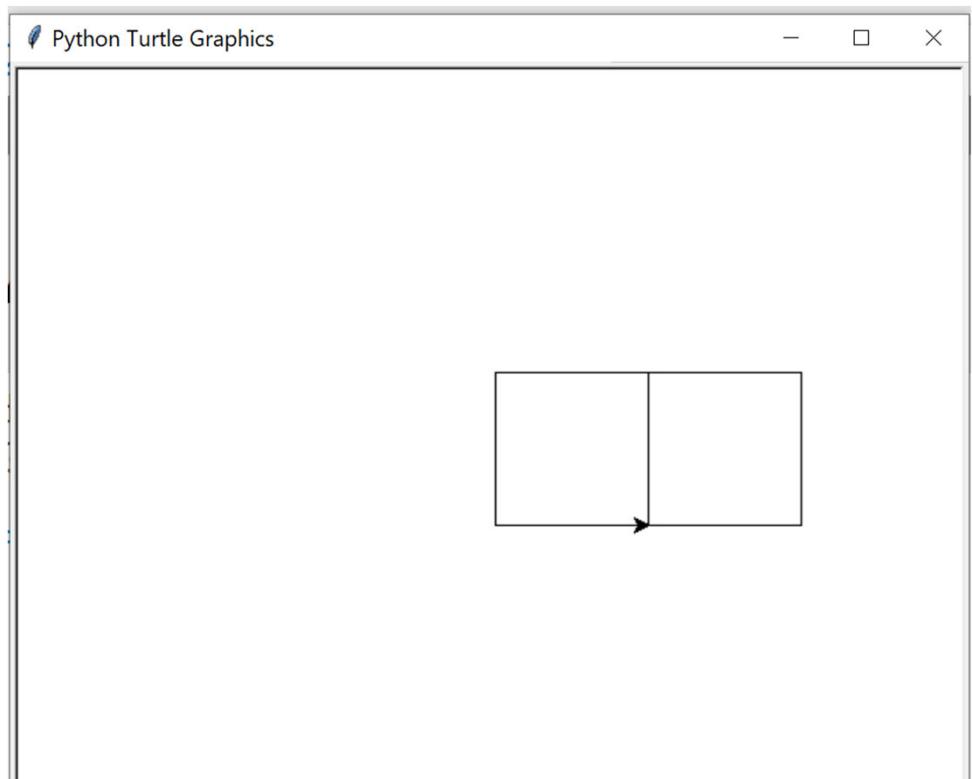
```
from turtle import *
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
```



What if we need to draw another square at a different location?

Drawing another square

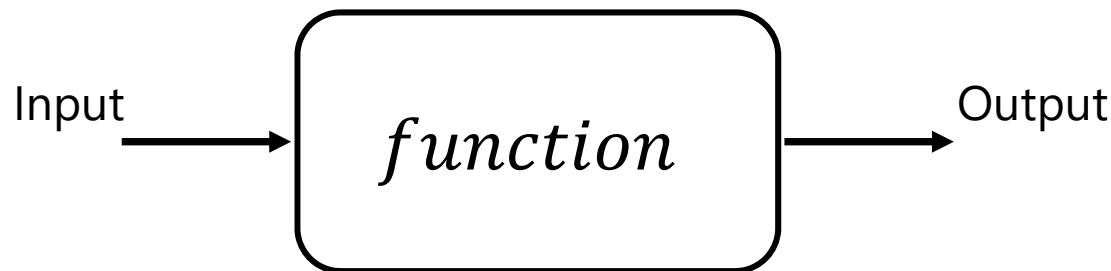
```
from turtle import *
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
```



What if you want more squares?

- Involves a lot of repetitions
- Instead write code to draw a square and reuse it
- How to do it?
 - Use functions
- Functions allows reusability
 - No need to rewrite the code for same functionality

Abstraction



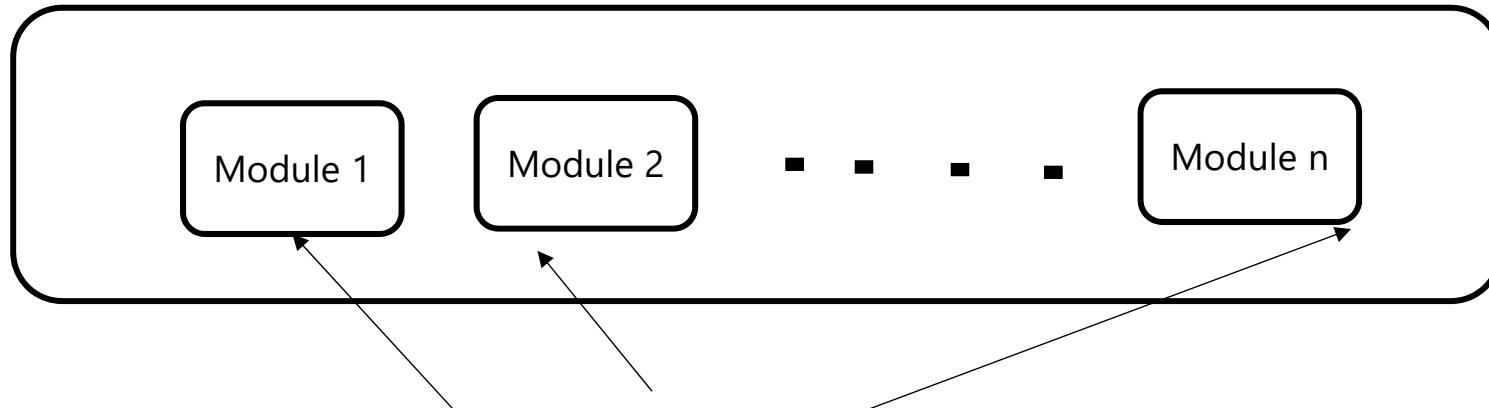
- Usability
 - User only needs to know input arguments and output data types
- Declarative style of programming
 - Tell the system what you want

Modularity

A Giant Module
(A program that do many things)

vs

Modular Architecture

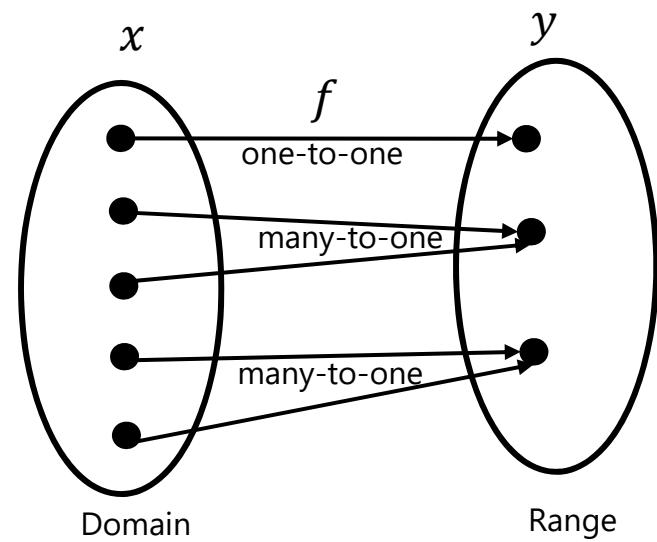


programs that only do one thing

Functions: Mathematics

- A mapping from input to output

name of the function
 $y = f(x)$
output argument

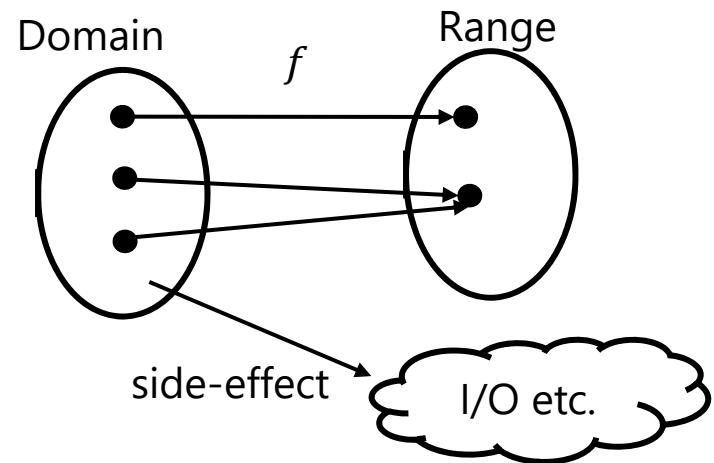


Functions in Programming

- A self-contained block of code
- Known as subroutines, methods, procedures, etc.

Types of Functions

- Pure functions
 - Functions without side-effects
 - Only mapping
 - Outputs depend only on inputs
- Functions with side-effects
 - Side-effects: I/O tasks
 - Taking inputs from keyboard, reading data from a file, etc.
 - Printing output to screen, writing data to a file, etc.
- Higher-order Functions (in Week 4)
 - Functions that accept functions as inputs and/or return functions as outputs



$$h(x) = f(g(x))$$

Functions in Python

- Built-in Functions
 - Builtin module is loaded automatically
 - No need to import
 - Examples: print(), input(), id(), etc.
- Functions from Python modules and packages
 - Not loaded automatically
 - Need to import them when needed
 - Example: function in modules like math, statistics.
- User-defined Functions
- Special function
 - main()

Built-In Functions

Built-in Functions			
A abs() aiter() all() any() anext() ascii()	E enumerate() eval() exec()	L len() list() locals()	R range() repr() reversed() round()
B bin() bool() breakpoint() bytearray() bytes()	F filter() float() format() frozenset()	M map() max() memoryview() min()	S set() setattr() slice() sorted() staticmethod() str() sum() super()
C callable() chr() classmethod() compile() complex()	G getattr() globals()	N next()	T tuple() type()
D delattr() dict() dir() divmod()	H hasattr() hash() help() hex()	O object() oct() open() ord()	V vars()
	I id() input() int() isinstance() issubclass() iter()	P pow() print() property()	Z zip()
			_ <u>import_</u> ()

Functions from Python Modules and Packages

- Import packages

➤ Example:

- math package

```
import math  
x = math.pi/2  
y = math.sin(x)  
print(y)
```

- We can import modules

➤ Syntax

- `import <module_name>`
- `from <module_name> import <name(s)>`

➤ Examples

- `import math`
- `from math import cos, sin`
- `from math import *`

```
from math import sin, pi  
x = pi/2  
y =sin(x)  
print(y)
```

Namespaces: Importing Modules

Syntax:

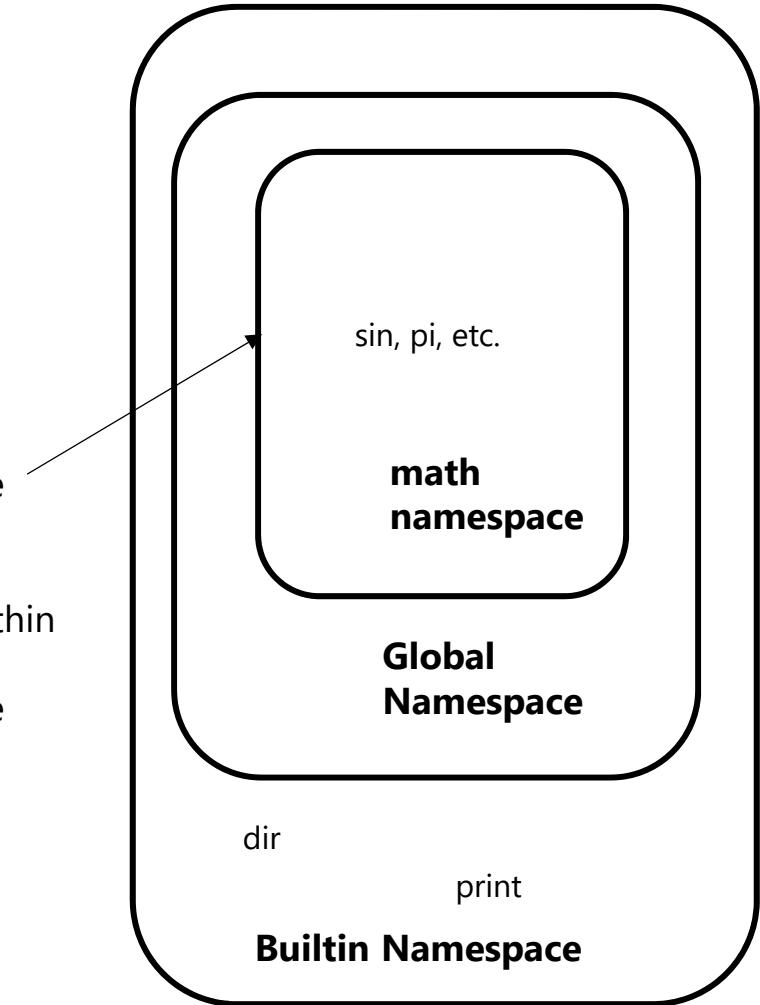
```
import <module_name>  
<module_name>.<object_name>
```

Example:

```
import math  
math.pi  
math.sin(math.pi/2)
```

```
>>> dir(__builtins__)  
>>> print(globals())  
>>> dir(math)
```

dedicated namespace for each imported module within global namespace



Importing Modules

```
>>> dir(math)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    dir(math)
NameError: name 'math' is not defined
>>> print(globals())
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>}
>>> import math
>>> print(globals())
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 'math': <module 'math' (built-in)>}
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod',
 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan',
 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

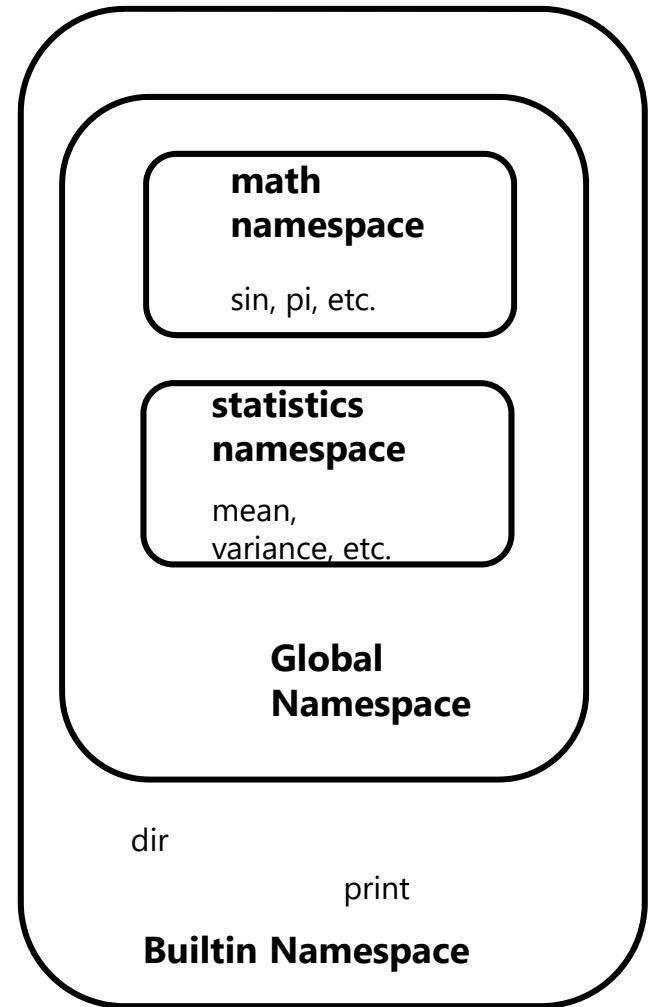
Namespaces: Importing Multiple Modules

Syntax:

```
import <module_1>, <module_2>  
<module_1>.<object_name>  
<module_2>.<object_name>
```

Example:

```
import math, statistics  
sin_pi_by_2 = math.sin(math.pi/2)  
mean_ = statistics.mean([1,2])  
print(f'sin_pi_by_2 is {sin_pi_by_2}')  
print(f'Mean of 1 and 2 is {mean_}')
```



Namespaces: Importing Objects from Modules

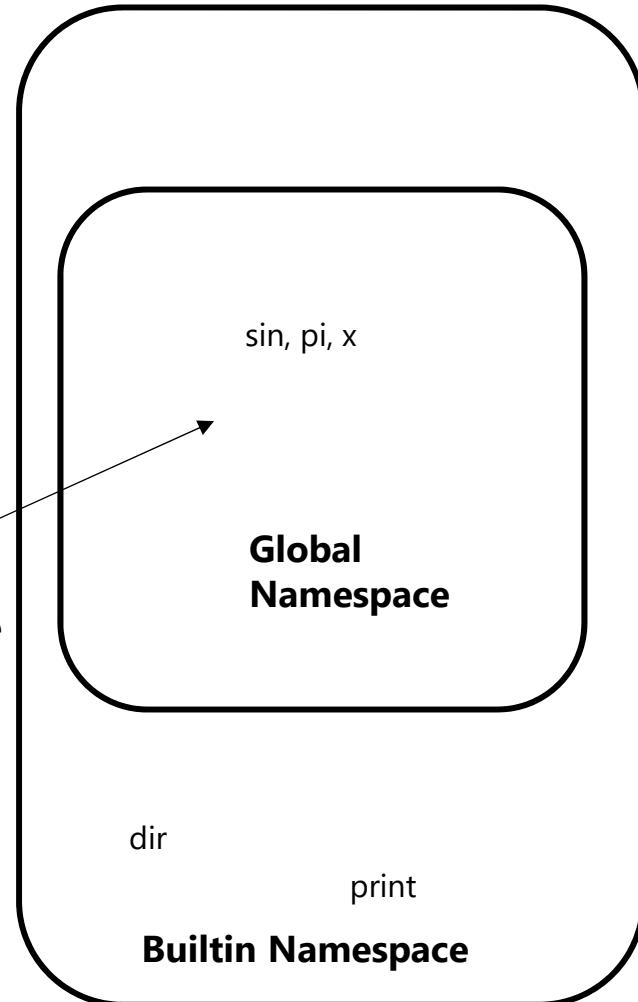
Syntax:

```
from <module_name> import <object_name>  
<module_name>.<object_name>
```

Example:

```
from math import sin, pi  
print(globals())  
x = sin(pi/2)  
print(f'sin_pi_by_2 is {x}')
```

Names from
math module
are imported
into global
namespace



You don't want to be like this in real life

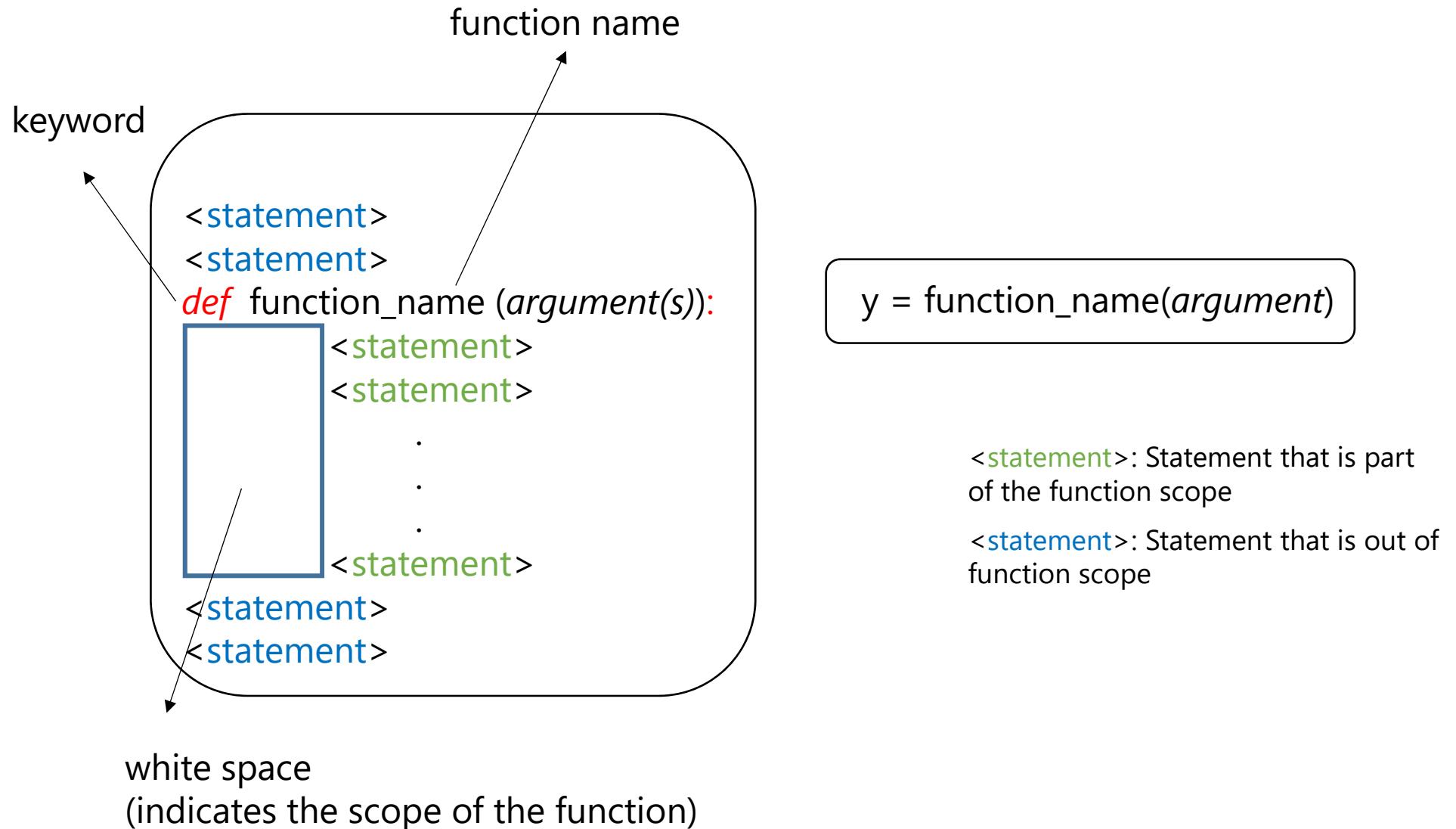


Try NOT to import
too many packages

User-defined Functions

- Simple functions
- Doc strings
- Arguments (Parameters)
 - Positional
 - Keyword
 - Default
- Parameter Passing
 - Pass-by Value
 - Pass-by Reference
 - Pass-by Assignment

User-Defined Functions



User-Defined Functions

```
print("Hello!")
def my_function_1():
    print("Hello Functions!")

    print("I am also a part of the function")
print("I am not a part of the function")

my_function_1()
```

Return Statement

- It does two things
 - Terminates the function
 - Return statement pass the output of function to the calling function

```
def function_name (arg_1, arg_2,..., arg_n):  
    <statement>  
    <statement>  
    .  
    .  
    .  
    return <statement>
```

Example

$$y = f(x_1, x_2) = \frac{x_1 + x_2}{2}$$

The diagram illustrates the components of a Python function definition:

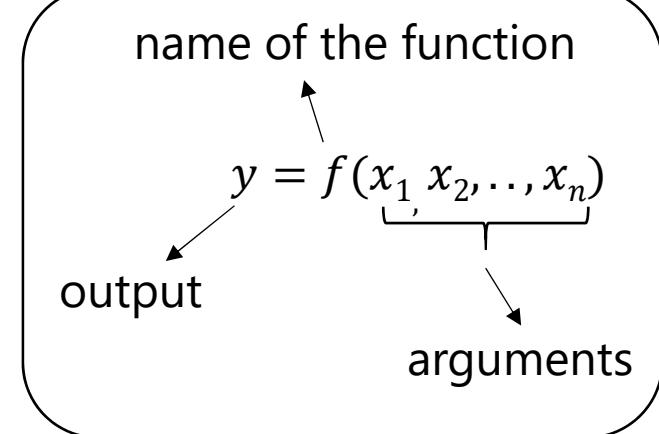
- name of function**: Points to the word `average`.
- arguments**: Points to the parameters `number_1` and `number_2`.
- Block of Code**: Points to the entire code block enclosed in a dashed box.
- keywords**: Points to the keyword `def`.
- white space**: Points to the indentation and newlines.
- Value returned by the function**: Points to the `return` statement and the expression `average_of_numbers`.

```
def average(number_1, number_2):
    sum_of_numbers = number_1+number_2
    average_of_numbers = sum_of_numbers/2
    return average_of_numbers
```

Arguments

- Provide input to functions

```
def function_name (arg_1, arg_2,...,arg_n):  
    <statement>  
    <statement>  
    .  
    .  
    .  
    <statement>
```



```
def my_module(arg_1):  
    print(f'Module: {arg_1}')  
my_module('IT5001')
```

Doc String

- Contains information about function
 - Describes how to use the function
 - Can access it using help/doc methods

```
def my_module(arg_1):
    """
    Parameters:
        arg_1 (str): ID of the module
                      Must be a string
    Returns:
        This function returns nothing

    Example:
        module_ID = 'IT5001'
        my_module(module_ID)
    Output:
        Module: IT5001
    ...
    print(f'Module: {arg_1}')
```

```
>>> help(my_module)
Help on function my_module in module __main__:

my_module(arg_1)
    Parameters:
        arg_1 (str): ID of the module
                      Must be a string
    Returns:
        This function returns nothing

    Example:
        module_ID = 'IT5001'
        function_2(module_ID)
    Output:
        Module: IT5001
```

Types of Arguments

- Positional arguments
- Keyword arguments
- Default/optional arguments

Positional Arguments

```
def module_info(module_code, module_name, module_type):  
    print(f"{module_code}: {module_name} is an {module_type} Module")  
  
>>> module_info("IT5001", "Software Development Fundamentals", 'Essential')  
IT5001: Software Development Fundamentals is an Essential Module
```

Positional Arguments

- Number of arguments are important

```
def module_info(module_code, module_name, module_type):  
    print(f"{module_code}: {module_name} is an {module_type} Module")
```

```
>>> module_info("IT5001", "Software Development Fundamentals")  
Traceback (most recent call last):  
  File "<pyshell#1>", line 1, in <module>  
    module_info("IT5001", "Software Development Fundamentals")  
TypeError: module_info() missing 1 required positional argument: 'module_type'
```

- Should take care of order of arguments

```
>>> module_info("IT5001", "Essential", "Software Development Fundamentals")  
IT5001: Essential is an Software Development Fundamentals Module
```

Keyword Arguments

- Order is not important

```
def module_info(module_code, module_name, module_type):  
    print(f"{module_code}: {module_name} is an {module_type} Module")  
  
>>> module_info(module_code = "IT5001", module_type = 'Essential', module_name =  
"Software Development Fundamentals")  
IT5001: Software Development Fundamentals is an Essential Module
```

- Names of the arguments should be the same as in function definition

```
>>> module_info(module_code = "IT5001", module_type = 'Essential', module_name =  
"Software Development Fundamentals")  
Traceback (most recent call last):  
  File "<pyshell#1>", line 1, in <module>  
    module_info(module_code = "IT5001", module_type = 'Essential', module_name =  
"Software Development Fundamentals")  
TypeError: module_info() got an unexpected keyword argument 'module_name'
```

Default/optional arguments

- Assign default values in function definition

default argument should always be at the end

```
def module_info(module_code, module_name, module_type = 'Essential'):  
    print(f"{module_code}: {module_name} is an {module_type} Module")
```

- Default arguments can be omitted while calling a function

```
>>> module_info("IT5001", "Software Development Fundamentals")  
IT5001: Software Development Fundamentals is an Essential Module
```

- Default argument can be assigned a different value

```
>>> module_info("CS522", " Advanced Computer Architecture", module_type = 'Elective')  
CS522: Advanced Computer Architecture is an Elective Module
```

Return vs Print

```
def sum_two_numbers(arg_1, arg_2):  
    return arg_1 + arg_2
```

Vs

Returns None

```
def sum_two_numbers(arg_1, arg_2):  
    print(arg_1 + arg_2)
```

```
x = sum_two_numbers(2, 3)
```

Revisit Drawing Squares

- How to generalize the code?

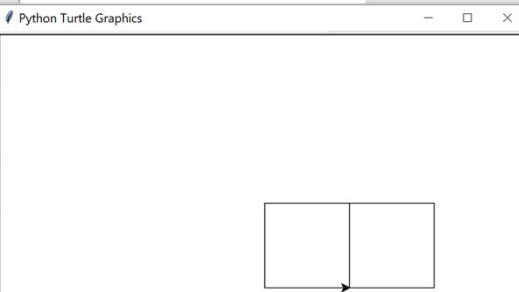
```
from turtle import *
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
```

- Retain similarities
- Parametrize differences

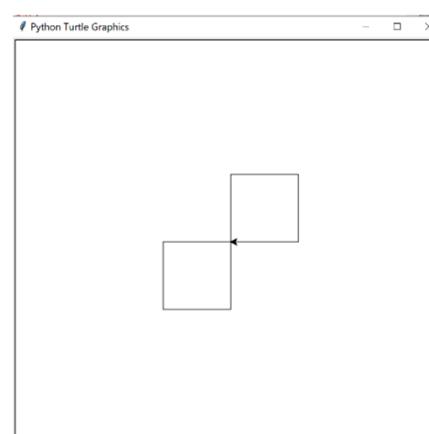
Drawing Squares using Functions

```
from turtle import *
def draw_square(side_length):
    forward(side_length)
    left(90)
    forward(side_length)
    left(90)
    forward(side_length)
    left(90)
    forward(side_length)
    left(90)
```

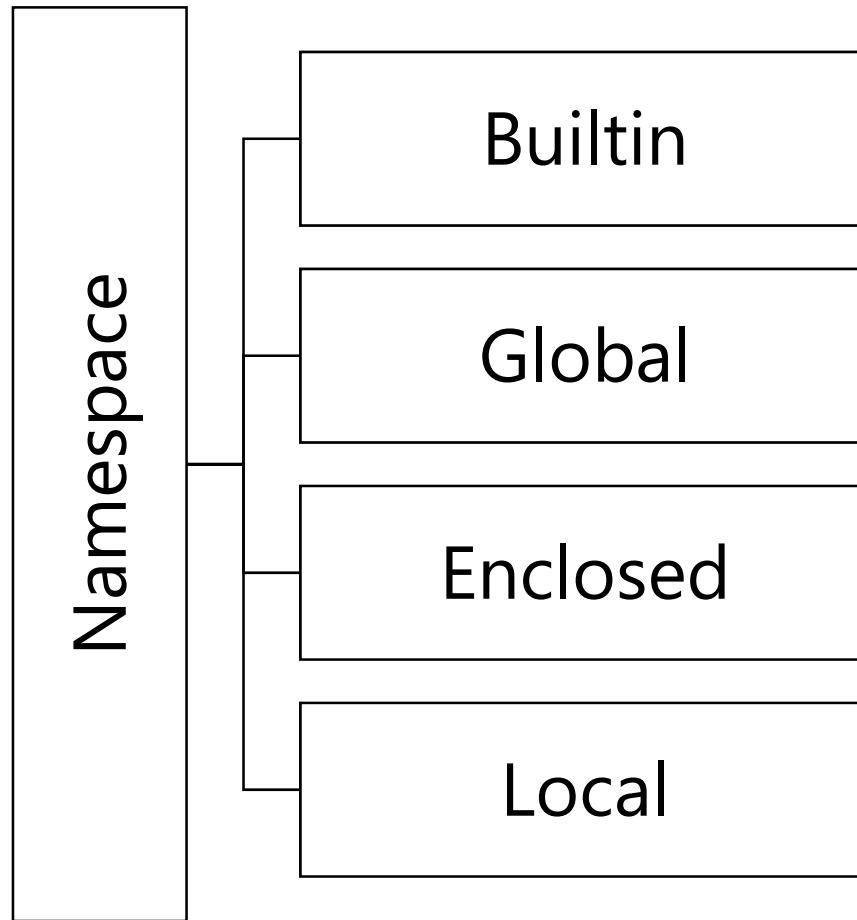
```
draw_square(100)
forward(100)
draw_square(100)
```



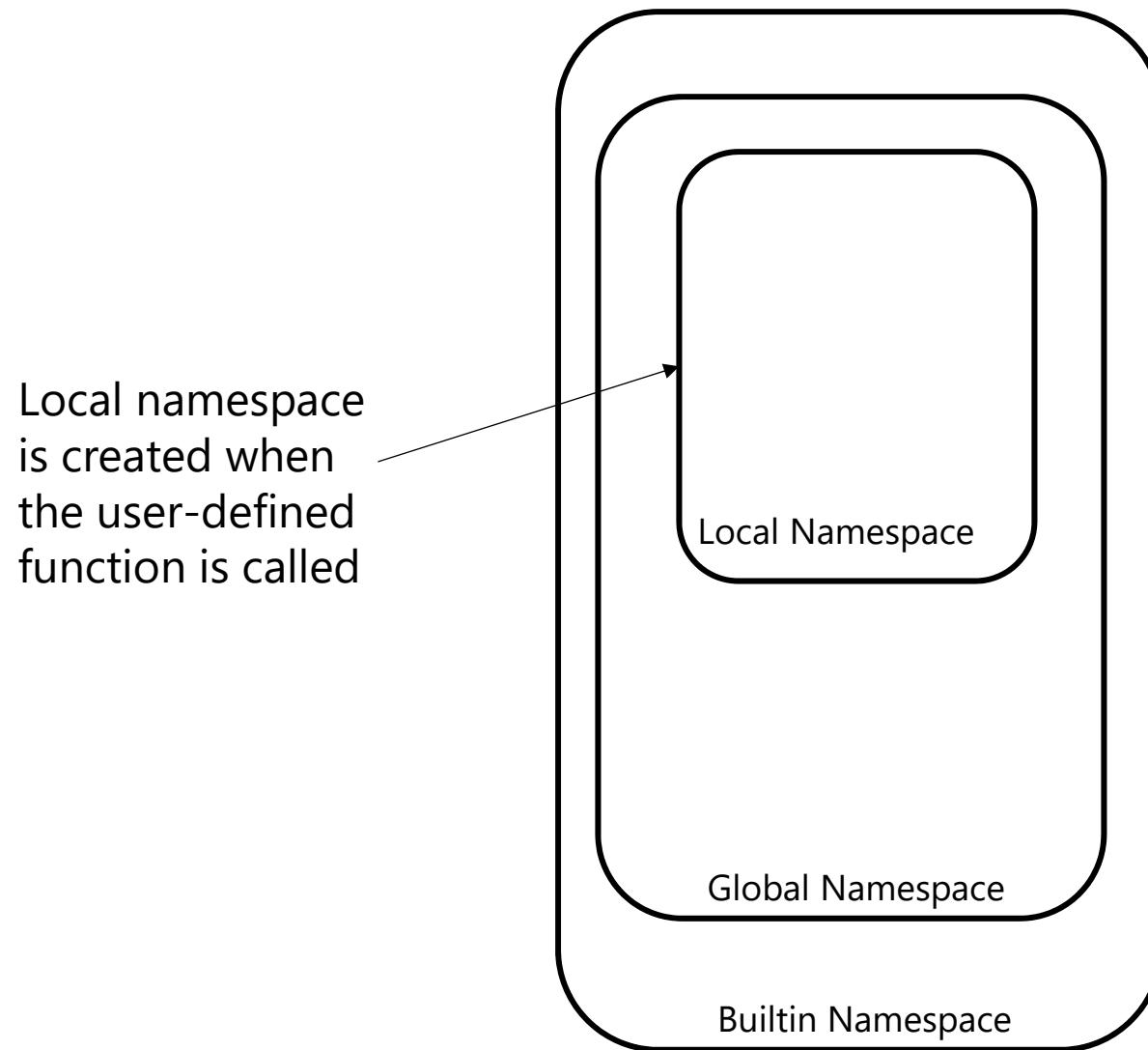
```
draw_square(100)
left(180)
draw_square(100)
```



Namespaces



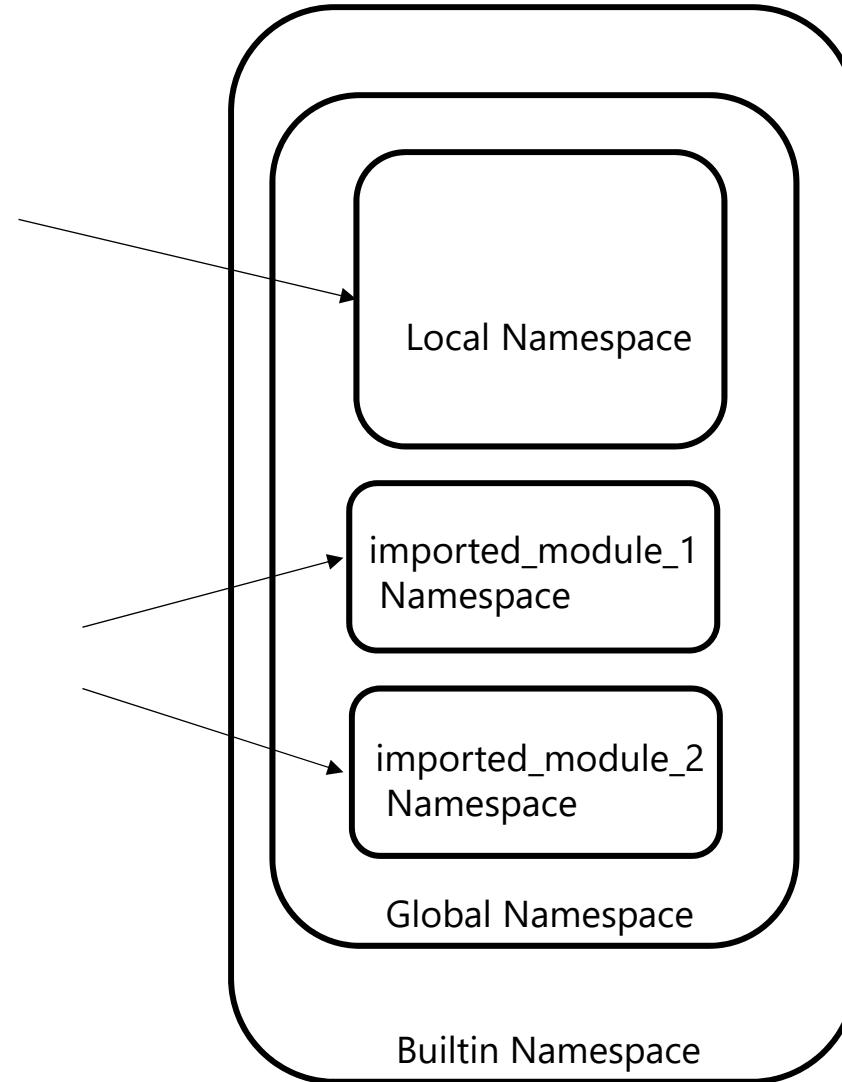
Namespaces: User-defined functions



Namespaces: Imported Modules and User-defined functions

Local namespace is created with in global namespace when the user-defined function is called

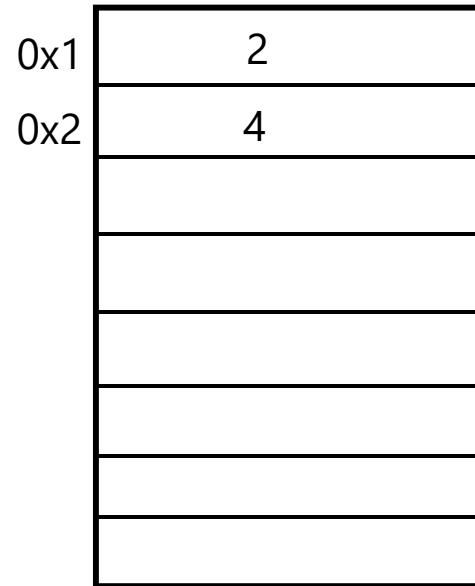
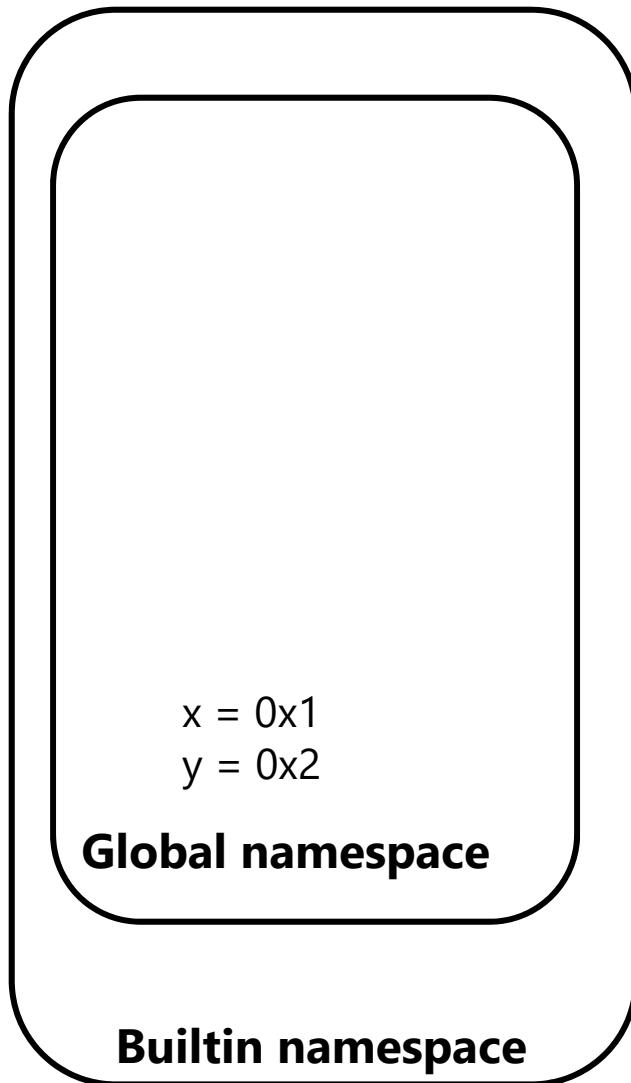
Dedicated namespace for each imported module



Functions: Tracing 1

```
x = 2
y = 4
def sum_two_numbers(x, y):
    return x + y
z = sum_two_numbers(3, 6)
print(z)
```

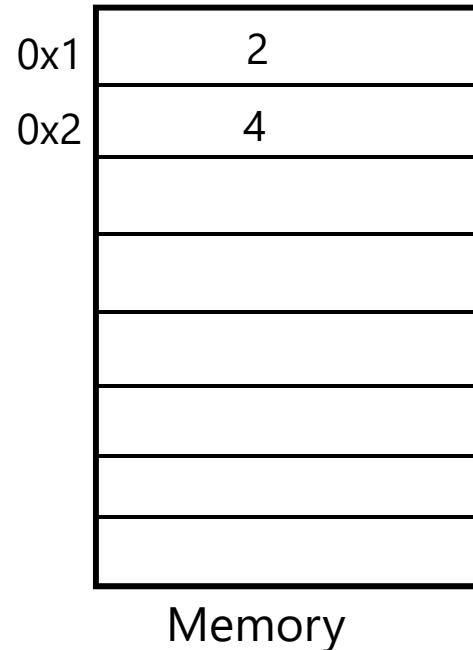
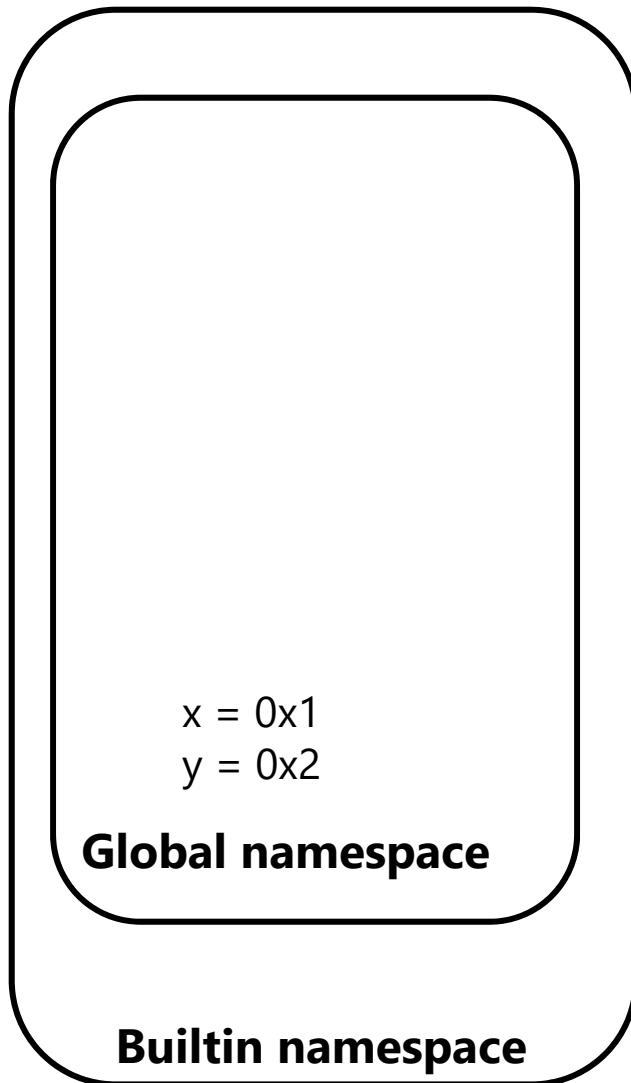
Functions: Tracing 1



Memory

```
1 x = 2
2 y = 4
3 def sum_two_numbers(x, y):
4     return x + y
5 z = sum_two_numbers(3, 6)
6 print(z)
7
```

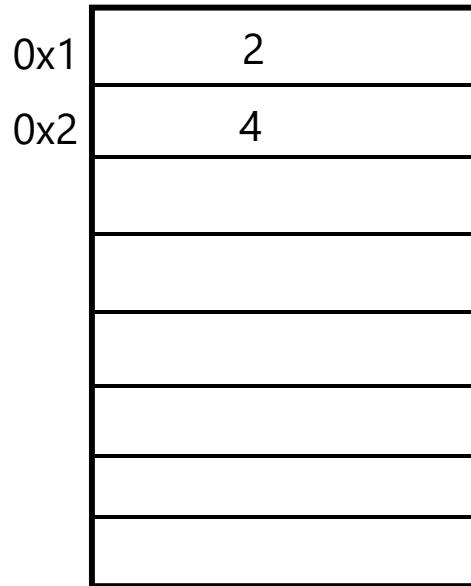
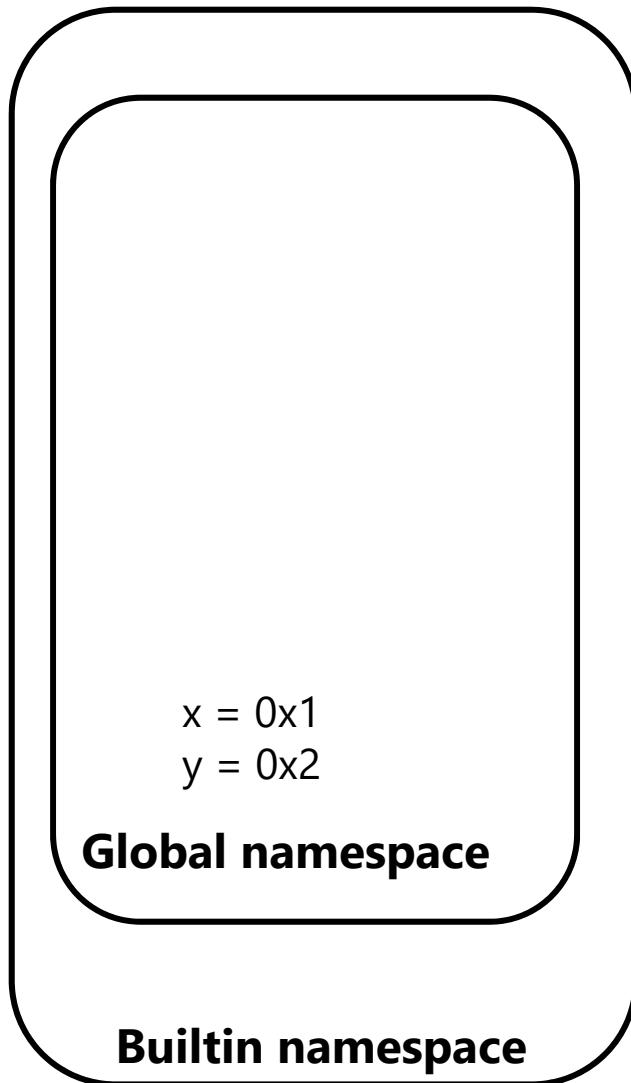
Functions: Tracing 1



Memory

```
1 x = 2
2 y = 4
3 def sum_two_numbers(x, y):
4     return x + y
5 z = sum_two_numbers(3, 6)
6 print(z)
7
```

Functions: Tracing 1

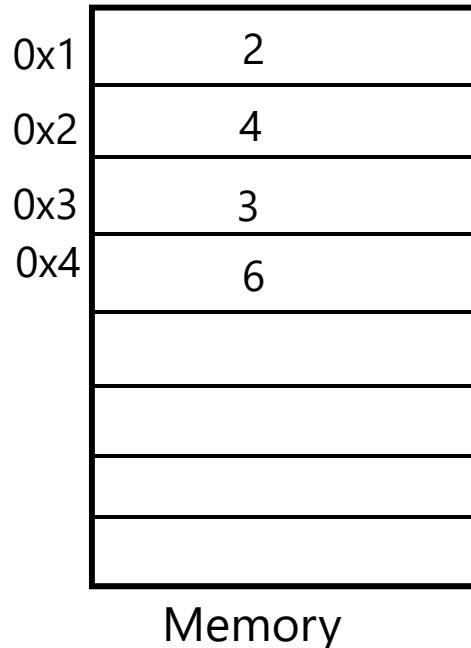
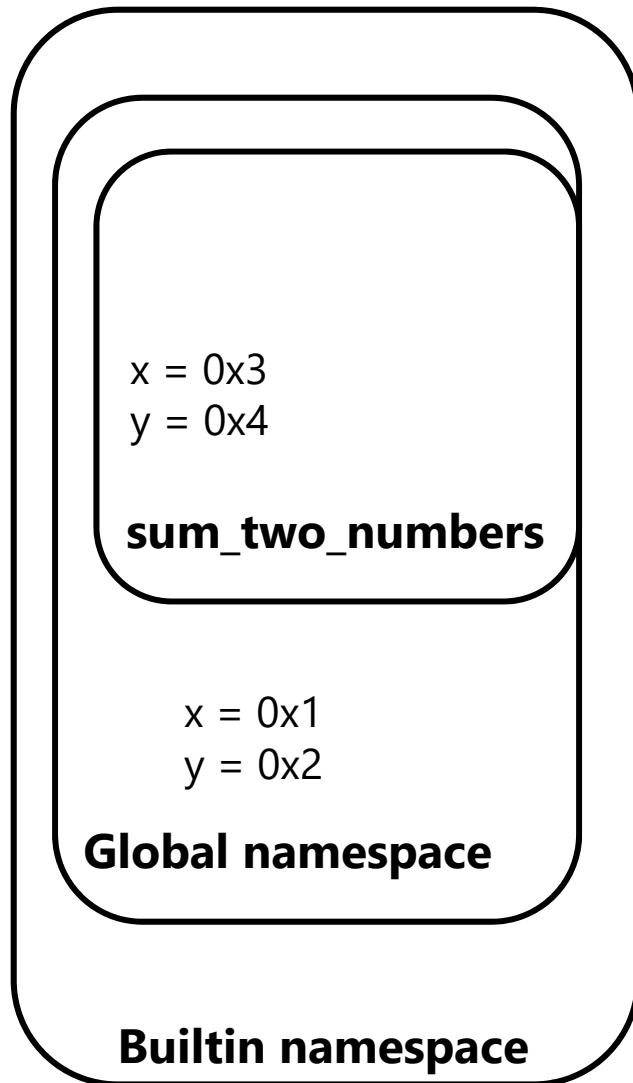


Memory

```
1 x = 2
2 y = 4
3 def sum_two_numbers(x, y):
4     return x + y
5 z = sum_two_numbers(3, 6)
6 print(z)
7
```

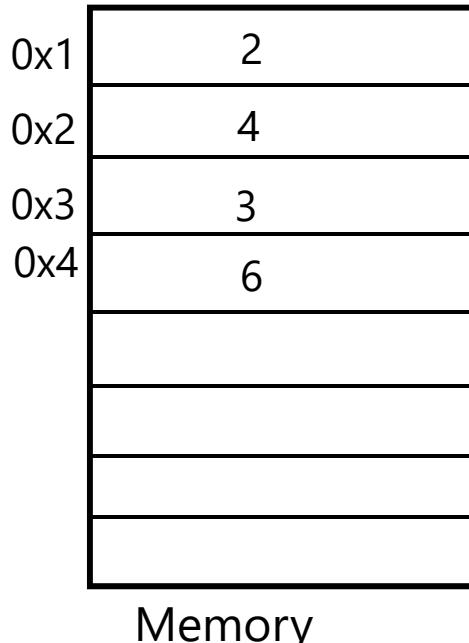
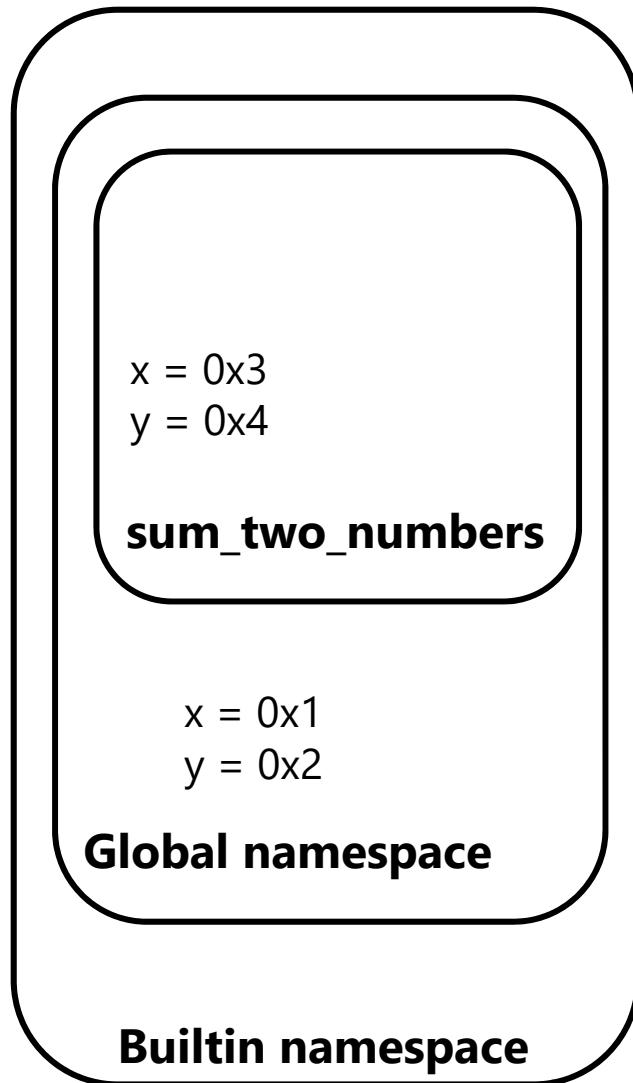
The code shows a function definition for `sum_two_numbers` and its call. The line `z = sum_two_numbers(3, 6)` is highlighted with a red dashed box.

Functions: Tracing 1



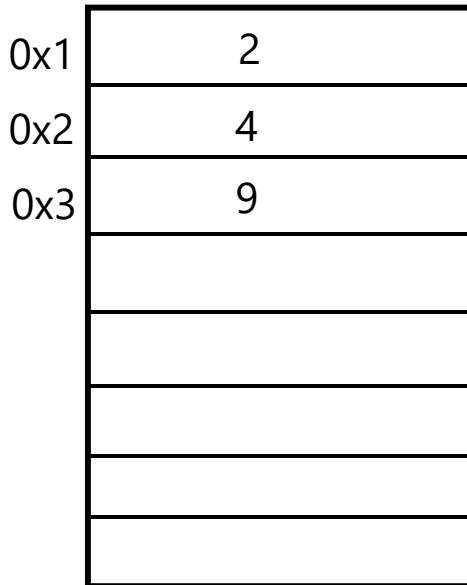
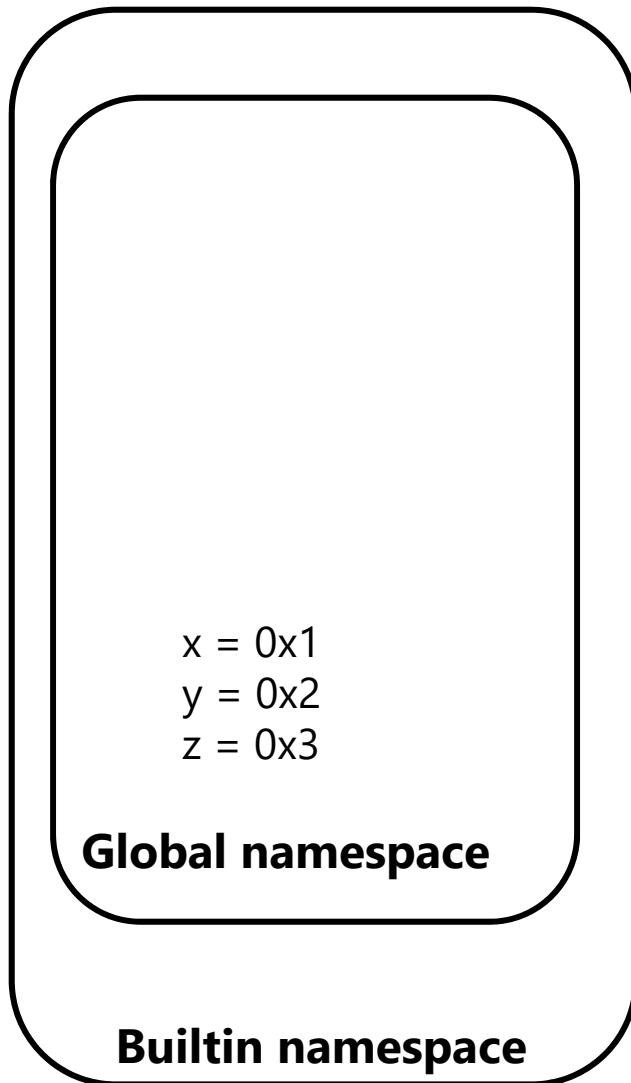
```
1 x = 2
2 y = 4
3 def sum_two_numbers(x, y):
4     return x + y
5 z = sum_two_numbers(3, 6)
6 print(z)
7
```

Functions: Tracing 1



```
1 x = 2
2 y = 4
3 def sum_two_numbers(x, y):
4     return x + y
5 z = sum_two_numbers(3, 6)
6 print(z)
7
```

Functions: Tracing 1



Memory

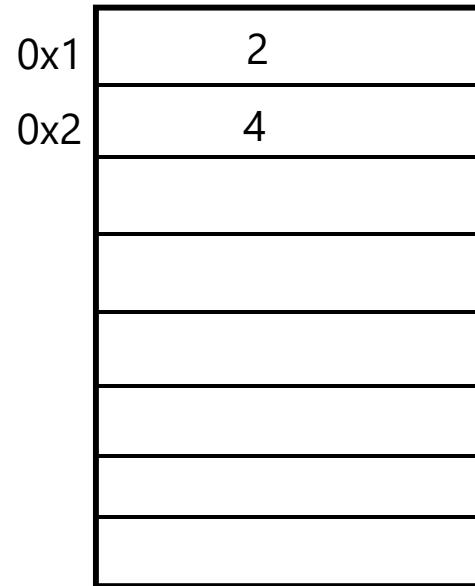
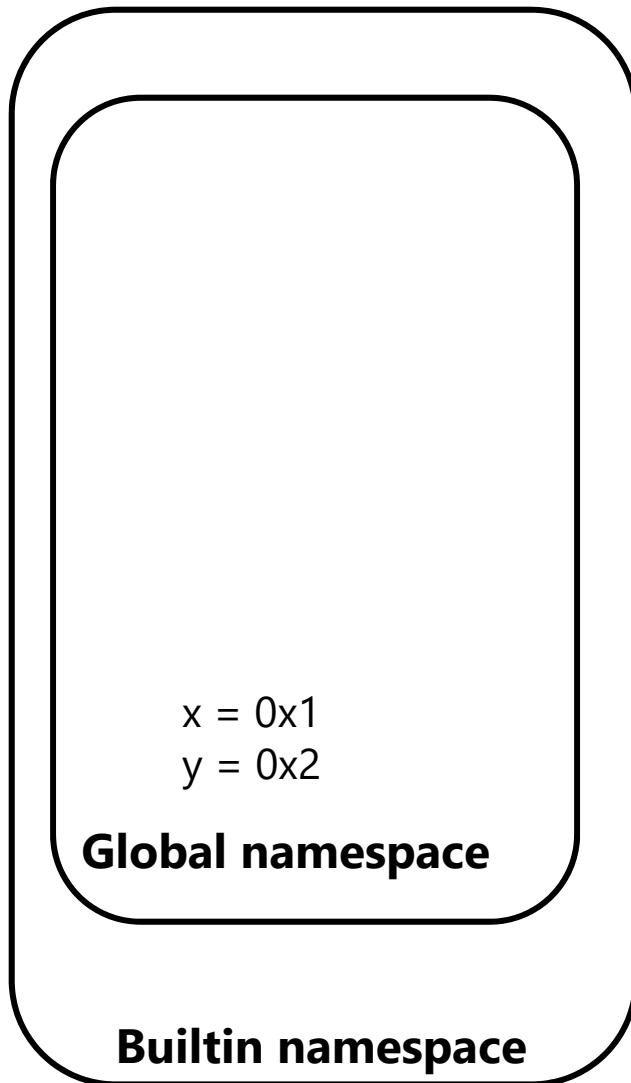
```
1 x = 2
2 y = 4
3 def sum_two_numbers(x, y):
4     return x + y
5 z = sum_two_numbers(3, 6)
6 print(z)
7
```

The code shows a function definition for `sum_two_numbers` that takes two parameters `x` and `y` and returns their sum. It then calls this function with arguments `3` and `6`, storing the result in `z` and printing it. The line `z = sum_two_numbers(3, 6)` is highlighted with a red dashed box.

Functions: Tracing 2

```
1 x = 2
2 y = 4
3 def sum_two_numbers(x):
4     return x+y
5
6 z = sum_two_numbers(3)
7 print(z)
o
```

Functions: Tracing 2



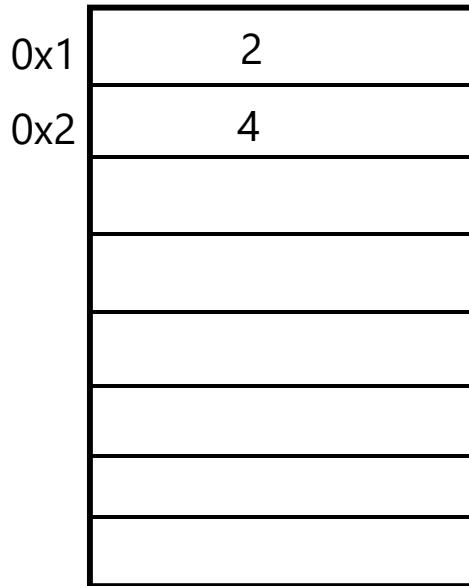
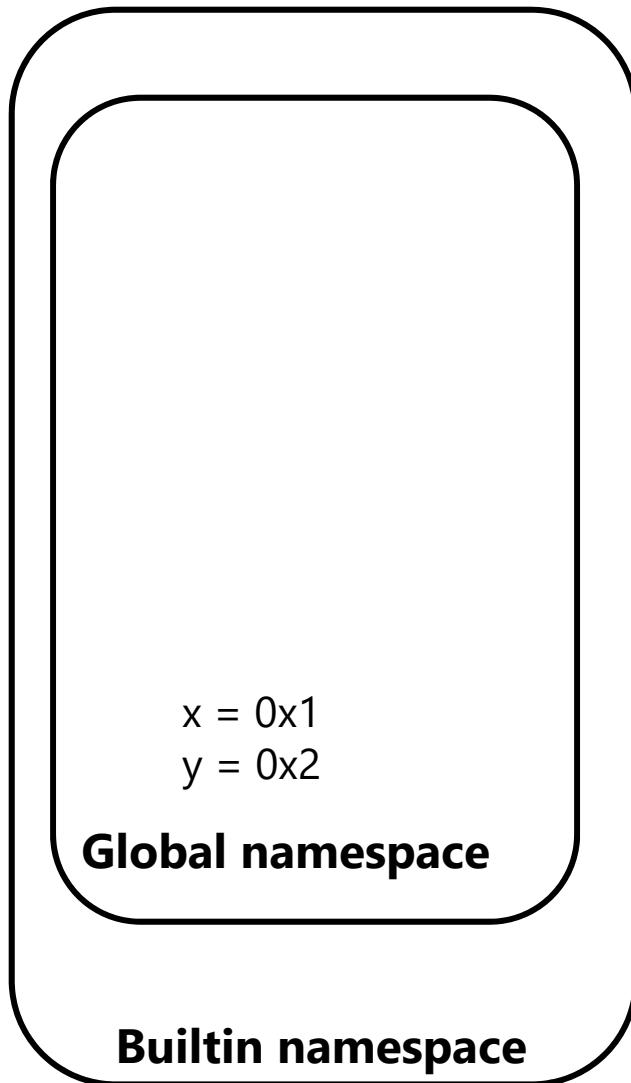
Heap Memory

A screenshot of a Python script editor showing a script named `script.py`. The code is:

```
1 x = 2
2 y = 4
3 def sum_two_numbers(x):
4     return x+y
5
6 z = sum_two_numbers(3)
7 print(z)
```

A red dashed rectangular box highlights the function definition from line 3 to line 7.

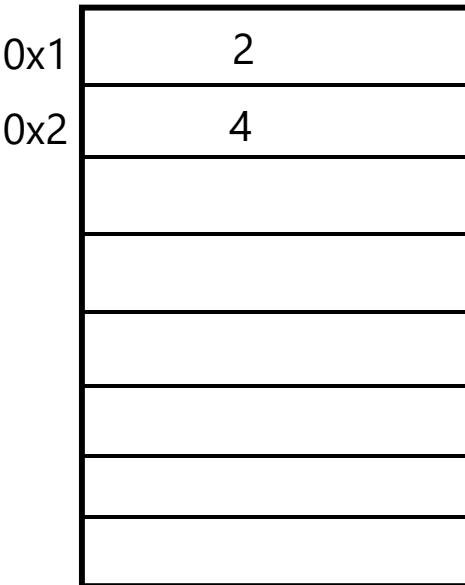
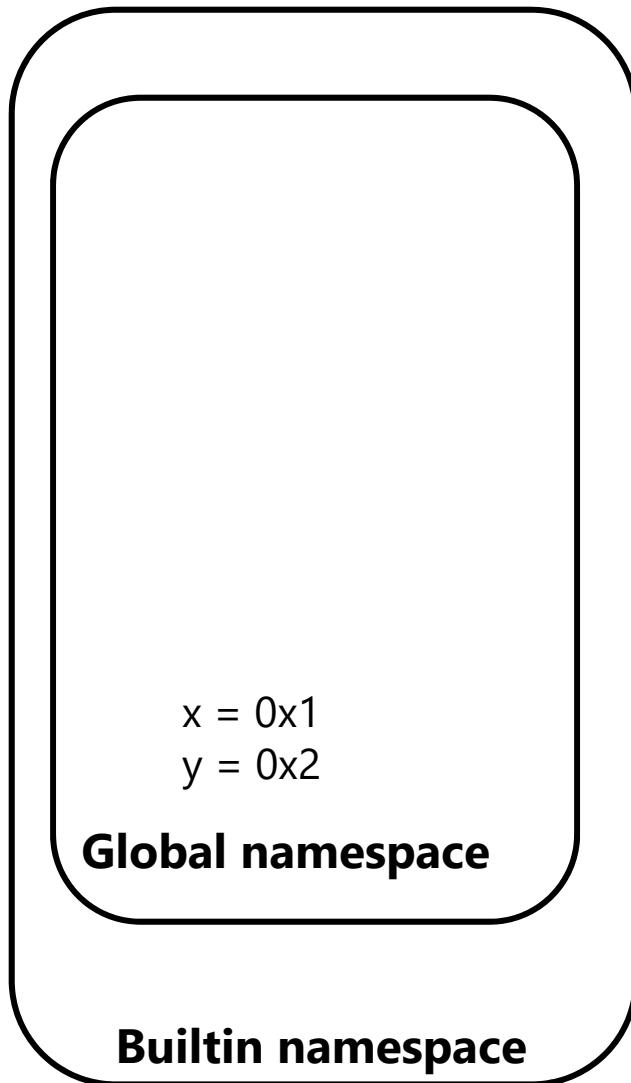
Functions: Tracing 2



Heap Memory

```
1 x = 2
2 y = 4
3 def sum_two_numbers(x):
4     return x+y
5
6 z = sum_two_numbers(3)
7 print(z)
```

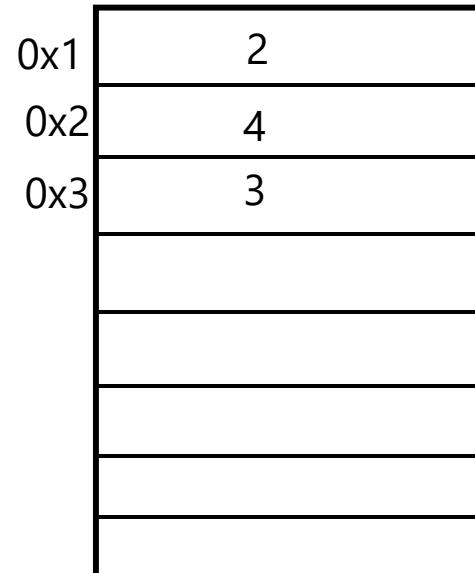
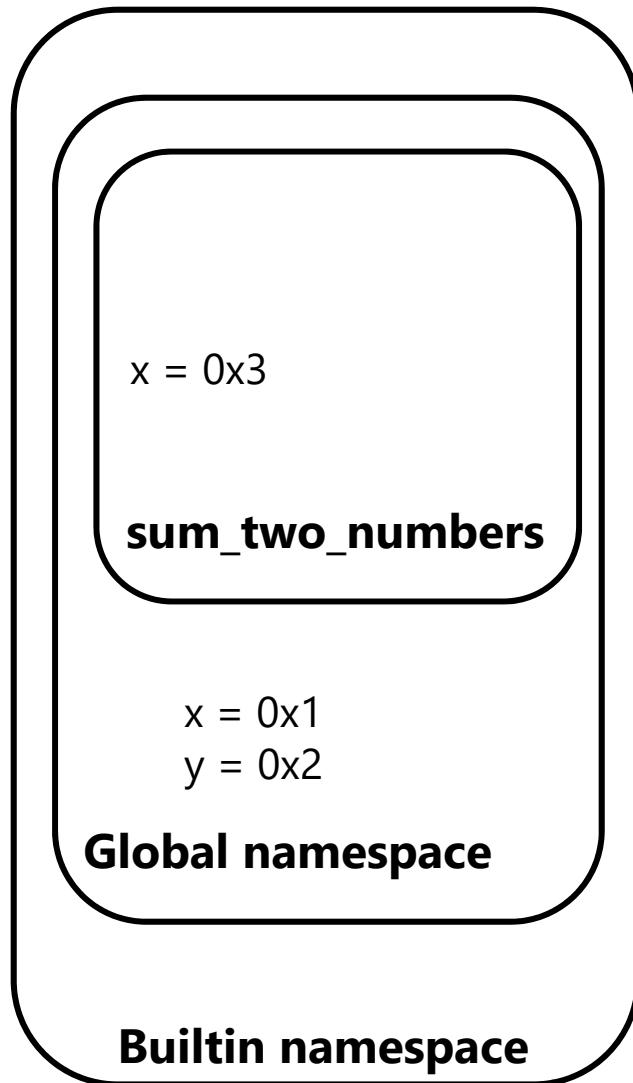
Functions: Tracing 2



Heap Memory

```
1 x = 2
2 y = 4
3 def sum_two_numbers(x):
4     return x+y
5
6 z = sum_two_numbers(3)
7 print(z)
```

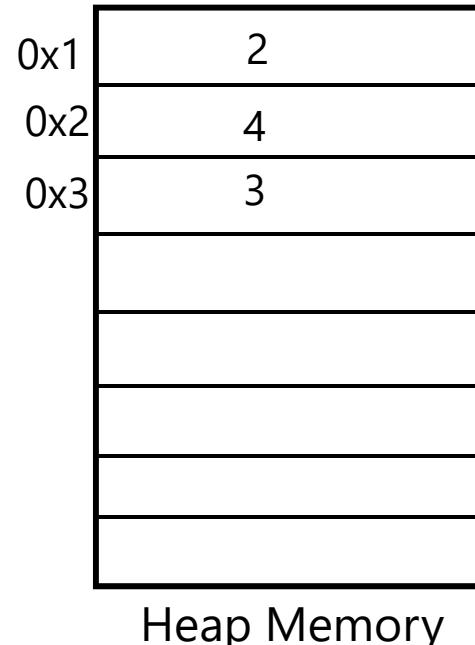
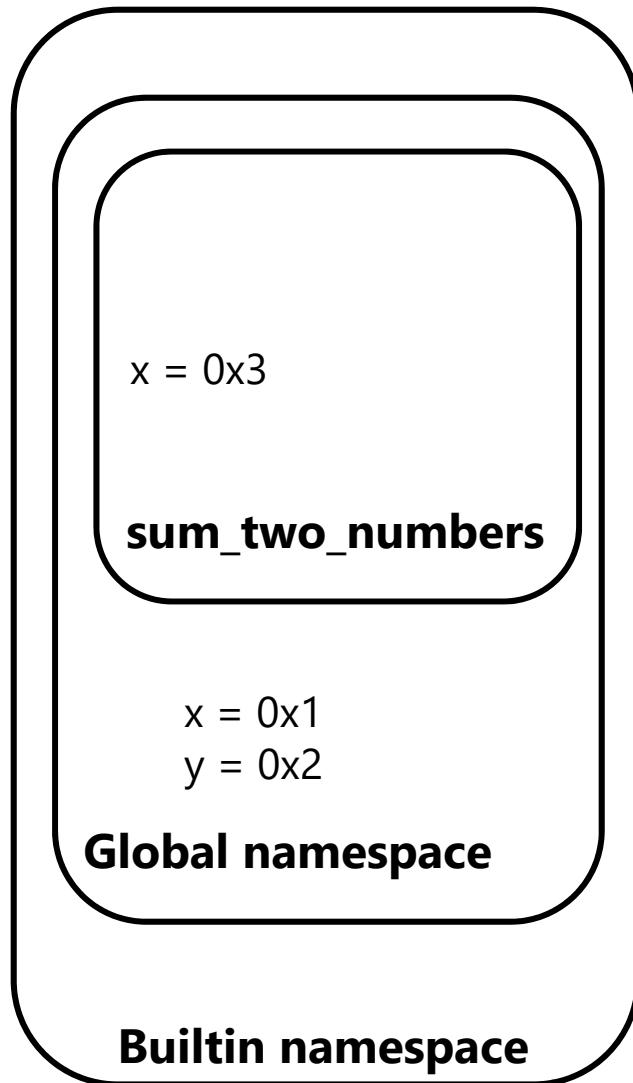
Functions: Tracing 2



Heap Memory

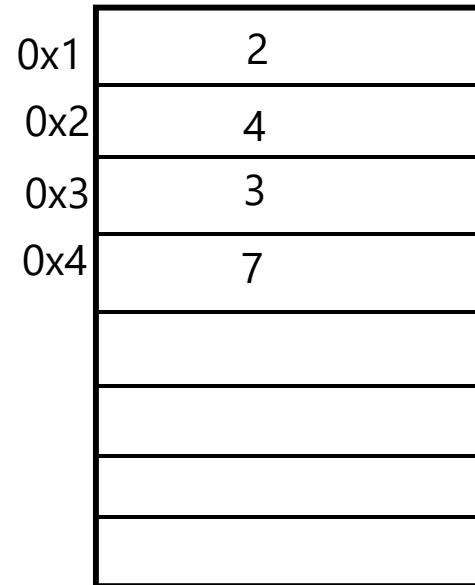
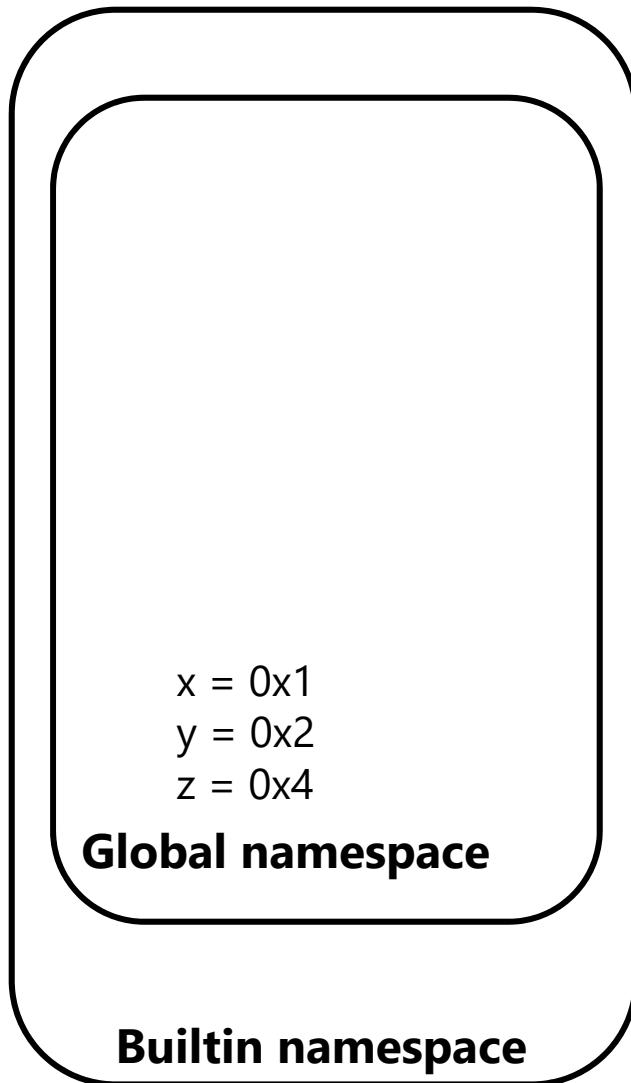
```
1 x = 2
2 y = 4
3 def sum_two_numbers(x):
4     return x+y
5
6 z = sum_two_numbers(3)
7 print(z)
```

Functions: Tracing 2



```
1 x = 2
2 y = 4
3 def sum_two_numbers(x):
4     return x+y
5
6 z = sum_two_numbers(3)
7 print(z)
```

Functions: Tracing 2

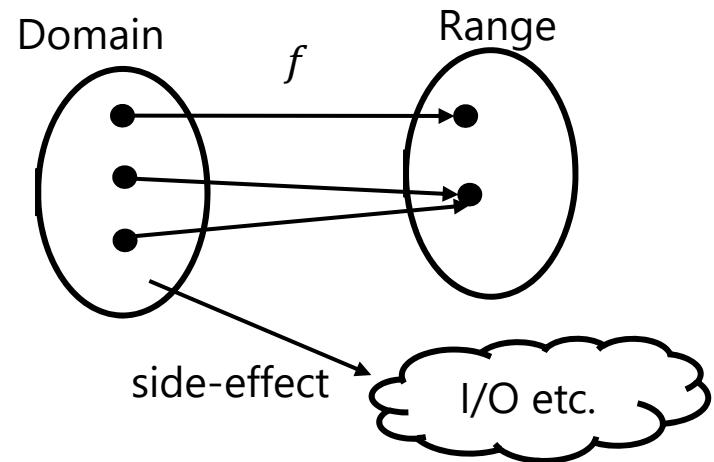


Heap Memory

```
1 x = 2
2 y = 4
3 def sum_two_numbers(x):
4     return x+y
5
6 z = sum_two_numbers(3)
7 print(z)
```

Side-effects

- Functions with side-effects
 - Side-effects: I/O tasks
 - Taking inputs from keyboard, reading data from a file, etc.
 - Printing output to screen, writing data to a file, etc.
- Pure functions
 - Functions without side-effects
 - Only mapping
 - Outputs depend only on inputs



Pure Function

Pure function: Yes/No?

```
x = 2  
y = 4  
def sum_two_numbers(x, y):  
    return x + y  
z = sum_two_numbers(3, 6)  
print(z)
```

Pure function: Yes/No?

```
x = 2  
y = 4  
def sum_two_numbers(x):  
    return x+y  
  
z = sum_two_numbers(3)  
print(z)
```

Summary

- Modules and Packages
- User-defined functions
- Argument Types
- Print vs Return
- Pure Function



IT5001 Software Development Fundamentals

3. Control Structures
Sirigina Rajendra Prasad

(Slide Credit: Prof. Alan, SoC, NUS)

Example: Solving a Quadratic Equation

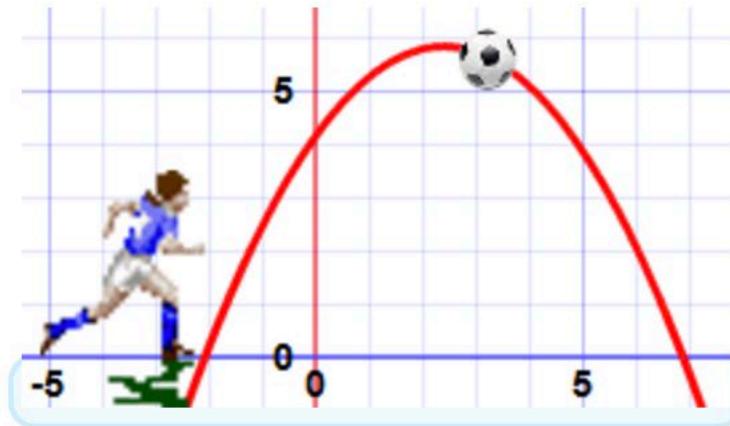
An example of a **Quadratic Equation**:

$$5x^2 + 3x + 3 = 0$$

this makes it Quadratic



Quadratic Equations make nice curves, like this one:



Example: Solving a Quadratic Equation

Standard Form

The **Standard Form** of a Quadratic Equation looks like this:

$$ax^2 + bx + c = 0$$

- **a**, **b** and **c** are known values. **a** can't be 0.
- "x" is the **variable** or unknown (we don't know it yet).

Example: Solving a Quadratic Equation

- Remember what we learned in high school...

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- Let's try to implement it in Python

```
from math import sqrt

def solve_qe(a,b,c):
    delta = b**2 - 4*a*c
    ans1 = (-b + sqrt(delta))/(2*a)
    ans2 = (-b - sqrt(delta))/(2*a)
    print("The two solutions are " + str(ans1)
          + " and " + str(ans2))
```

```
>>> solve_qe(1,5,6)
The two solutions are -2.0 and -3.0
>>> solve_qe(1,4,4)
The two solutions are -2.0 and -2.0
>>>
```

However...

```
>>> solve_qe(1,-5,6)
The two solutions are 3.0 and 2.0
>>> solve_qe(1,1,8)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    solve_qe(1,1,8)
  File "C:\Users\dcschl\Google Drive\Courses\YSC22
21\Lectures\solve_qe1.py", line 5, in solve_qe
    ans1 = (-b + sqrt(delta))/(2*a)
ValueError: math domain error
```

- Why?

```
from math import sqrt

def solve_qe(a,b,c):
    delta = b**2 - 4*a*c
    ans1 = (-b + sqrt(delta))/(2*a)
    ans2 = (-b - sqrt(delta))/(2*a)
    print("The two solutions are " + str(ans1)
          + " and " + str(ans2))
```



```
>>> solve_qe(1,-5,6)           delta = 25-24 = 1 > 0
The two solutions are 3.0 and 2.0
>>> solve_qe(1,1,8)           delta = 1 - 32 = -31 < 0
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    solve_qe(1,1,8)
  File "C:\Users\dcschl\Google Drive\Courses\YSC22
21\Lectures\solve_qe1.py", line 5, in solve_qe
```

Example: Solving a Quadratic Equation

- Remember what we learned in high school...

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

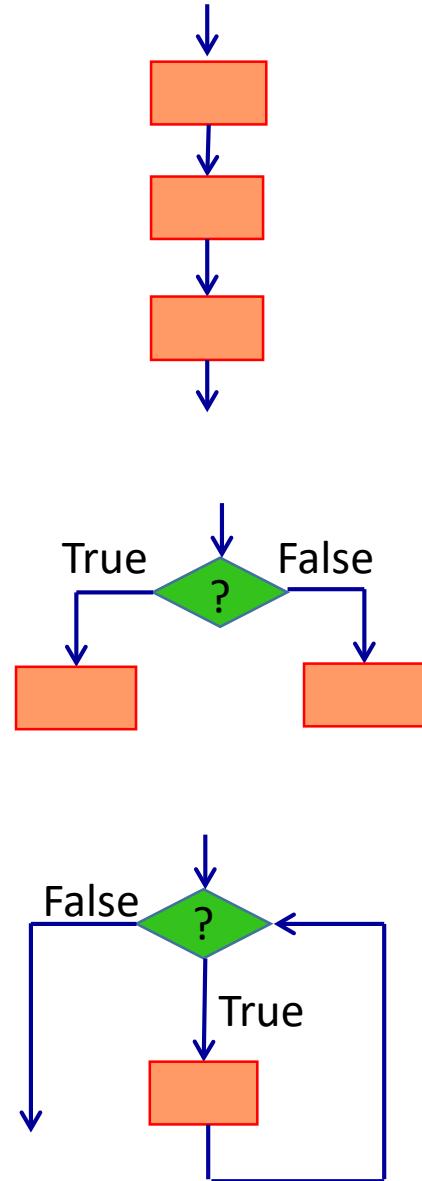
- If $\Delta < 0$
 - The equation has no real solution
- So we cannot call `sqrt()` if Δ is negative

Control Structures

The basic building blocks of programming

Control Structures

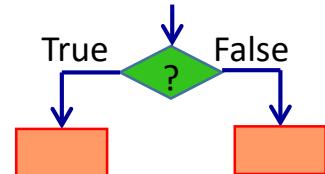
Sequence	• Default
Selection	• Also called branching
Repetition	• Also called loop



Making Choices



Control Structure: Selection



If (a condition is true)

Do A

Else

Do B

Can be **MORE THAN** one single instruction

- For example:

If (I have \$1000000000000000)

Buy a car

Eat a lot of buffets

Go travel

Quit NUS!

Else

Be good and study

Control Structure: Selection

If (a condition is true)

Do A

Else

Do B

- For example:

If (I have \$1000000000000000)

If (I am heartless)

Buy a car

Eat a lot of buffets

Go travel

Quit NUS!

Else

 donate all the money to charity

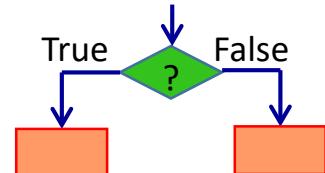
Else

 Be good and study

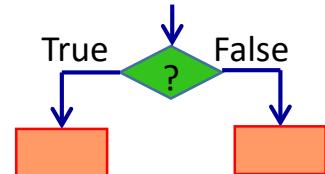
Can be **MORE THAN** one single instruction



Nested "if"



Control Structure: Selection



If (a condition is true)

Do A

Else

Do B

Can be **WITHOUT** "else"

- For example:

If (I have \$1000000000000000)

Buy a car

Eat a lot of buffets

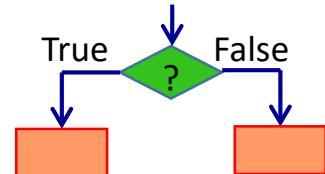
Go travel

Quit NUS!

Else

Be good and study

Control Structure: Selection



If (a condition is true)

Do A

Else

Do B

Can be **MORE THAN** one single instruction

- For example:

If (I have \$1000000000000000)

Buy a car

Eat a lot of buffets

Go travel

Quit NUS!

Else

Be good and study

Condition

```
def solve_qe(a,b,c):
    delta = b**2 - 4*a*c
    if delta >= 0:
        ans1 = (-b + sqrt(delta)) / (2*a)
        ans2 = (-b - sqrt(delta)) / (2*a)
        print("The two solutions are " + str(ans1)
              + " and " + str(ans2))
    else:
        print("The equation has no real root")
```

condition

If the condition is True

```
def solve_qe(a,b,c):
    delta = b**2 - 4*a*c
    if delta >= 0:
        ans1 = (-b + sqrt(delta)) / (2*a)
        ans2 = (-b - sqrt(delta)) / (2*a)
        print("The two solutions are " + str(ans1)
              + " and " + str(ans2))
    else:
        print("The equation has no real root")
```

If the condition is False

```
def solve_qe(a,b,c):
    delta = b**2 - 4*a*c
    if delta >= 0:
        ans1 = (-b + sqrt(delta)) / (2*a)
        ans2 = (-b - sqrt(delta)) / (2*a)
        print("The two solutions are " + str(ans1)
              + " and " + str(ans2))
    else:
        print("The equation has no real root")
```

Conditional

Syntax

```
if <expr>:  
    statement(s)
```

Example

```
>>> my_money = 1000  
>>> if my_money > 0:  
        print('Good')
```

'Good'



indentation

Conditional

Syntax

```
if <expr>:  
    statement(s)
```

Example

```
>>> my_money = 1000  
>>> if my_money > 0:  
        print('Good')  
        print('Good')  
        print('Good')
```

indentation

```
'Good'  
'Good'  
'Good'
```

Conditional

Syntax

```
if <expr>:  
    statement(s)  
  
else:  
    statement(s)
```

Example

```
>>> my_account = 1000  
>>> if my_account > 0:  
        print('rich')  
  
else:  
    print('broke')  
'rich'
```

Conditional (Nested)

Syntax

```
if <expr>:  
    if <expr>:  
        statement(s)
```

Example

```
a = 4  
if a < 10:  
    if a < 1:  
        print('Here')
```

Print nothing

Conditional

Syntax

```
if <expr>:  
    statement(s)  
  
else:  
    statement(s)
```

Example

```
>>> my_account = 1000  
>>> if my_account < 0:  
        print('poor')  
  
else:  
    if my_account > 1:  
        print('v rich')
```

Clumsy

v rich

Conditional

Syntax

```
if <expr>:  
    statement(s)  
elif <expr>:  
    statements(s)  
else:  
    statement(s)
```

Example

```
>>> a = -3  
>>> if a > 0:  
        print('yes')  
    elif a == 0:  
        print('no')  
    else:  
        print('huh')  
'huh'
```

Conditional

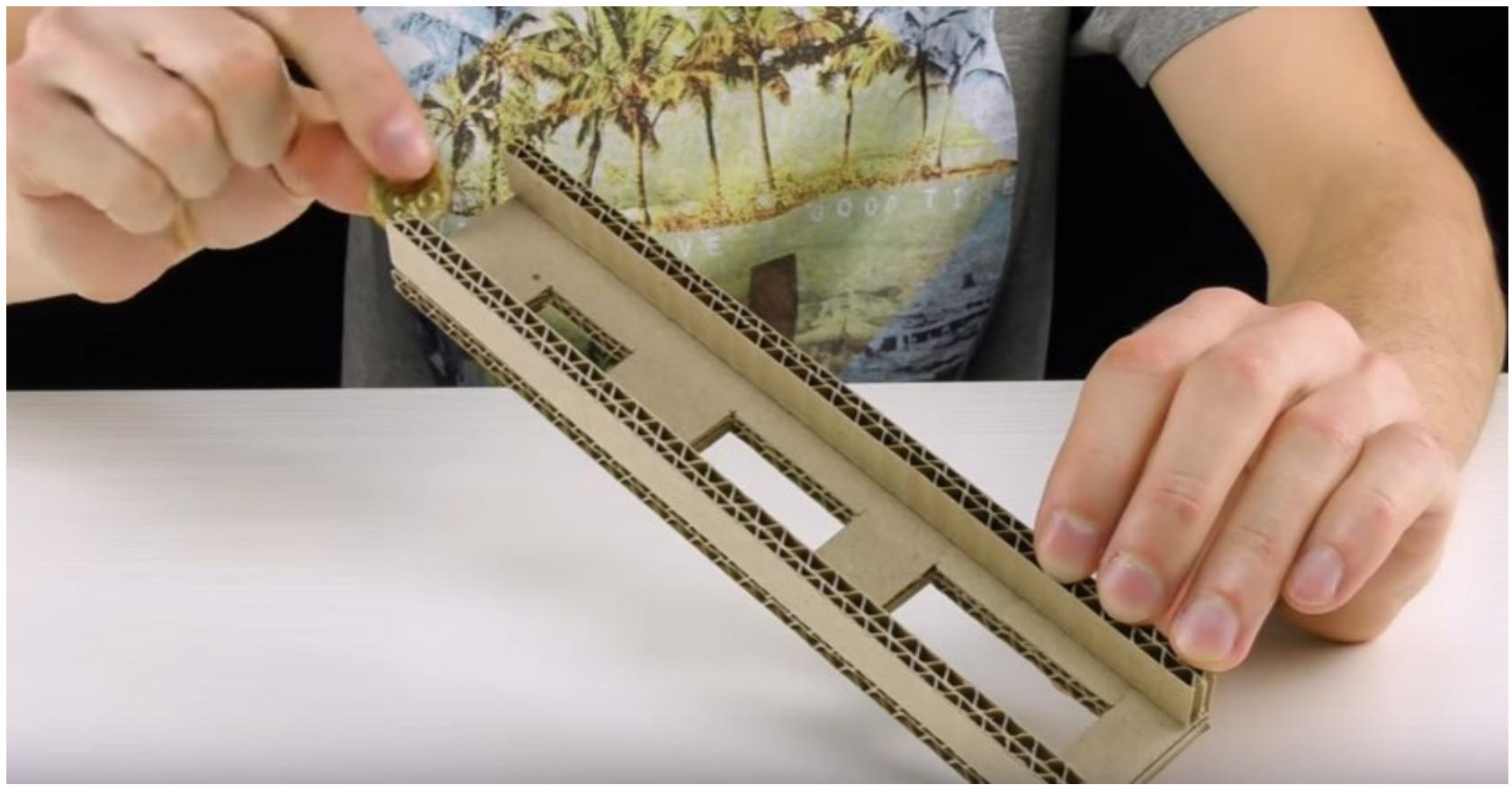
Syntax

```
if <expr>:  
    statement(s)  
elif <expr>:  
    statements(s)  
elif <expr>:  
    statements(s)  
else:  
    statement(s)
```

Can be many

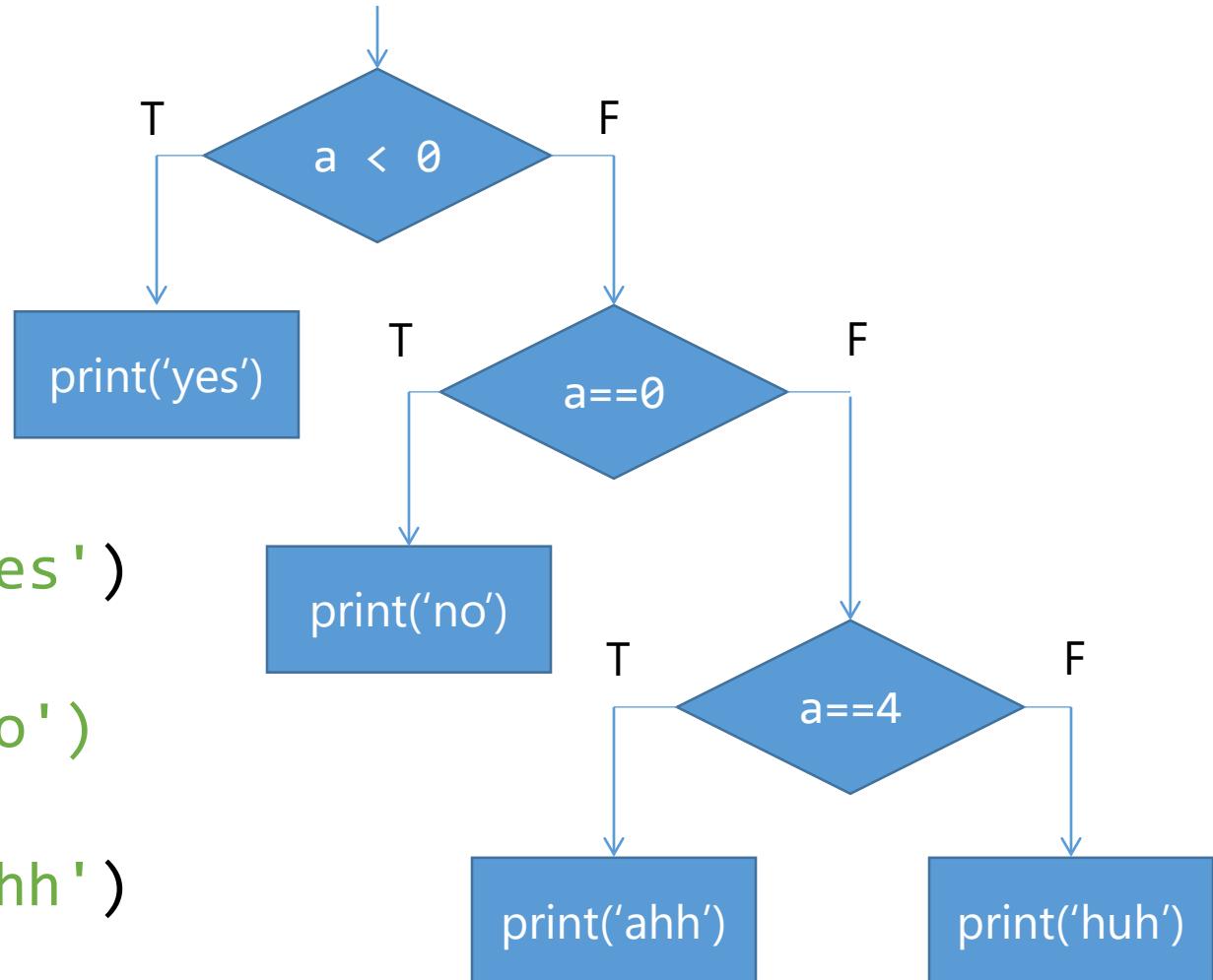
Example

```
>>> a = 4  
>>> if a > 0:  
            print('yes')  
elif a == 0:  
            print('no')  
elif a == 4:  
            print('ahh')  
else:  
    print('huh')  
  
'yes'
```



e.g.

```
>>> a = 4
>>> if a < 0:
        print('yes')
elif a == 0:
    print('no')
elif a == 4:
    print('ahh')
else:
    print('huh')
'ahh'
```



Homework: Figure out ALL conditions

```
def solve_qe(a,b,c):
    delta = b**2 - 4*a*c
    if delta >= 0:
        ans1 = (-b + sqrt(delta)) / (2*a)
        ans2 = (-b - sqrt(delta)) / (2*a)
        print("The two solutions are " + str(ans1)
              + " and " + str(ans2))
    else:
        print("The equation has no real root")
```

condition

Repetition



Control Structure: Repetition

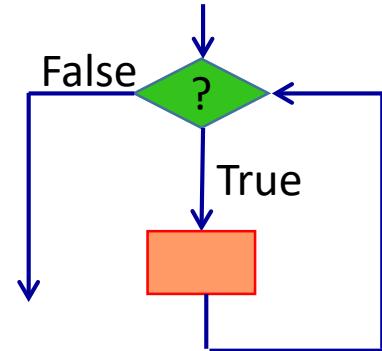
- While (a condition)
 - Do something

- For example

```
While (I am hungry)
    Eat a bun
```

- Again, can be more than one single instruction

```
While(I have money in bank)
    Take money out from bank
    Eat an expensive meal
    While(I have money in my wallet)
        Go Shopping
```



Iteration

the act of repeating a process with the aim of approaching a desired goal, target or result.

- Wikipedia

Three Types of Loops

- Must run exactly N times
- Run any number of times
- Run at most N times
 - Check all True (or check all False)
 - Find any True (or False)

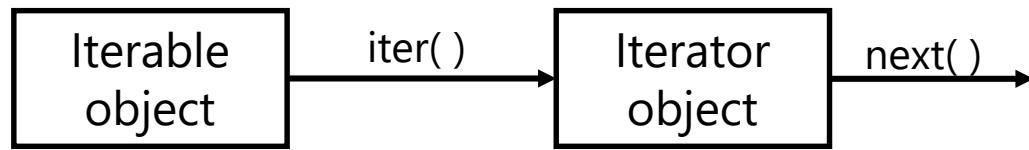
Three Types of Loops

- **Must run exactly N times**
- Run any number of times
- Run at most N times
 - Check all True (or check all False)
 - Find any True (or False)

For loop

- Uses iterable objects to repeatedly execute a series of tasks
- Number of repetitions are equal to number of items provided by iterable object
- Let us first look at iterable objects

Iterable and Iterators



- Can be looped over
- Any object with `__iter__` method is iterable
- Builtin Iterables:
 - range, strings, lists, tuples
- User-defined iterable objects
 - Generator functions
- Repeated calls of `next()` method return next item from iterator object
- If no further items in iterator, `next()` method raises `StopIteration` exception

Iterables

- This week:
 - `range()` - builtin iterable
- Subsequently
 - strings
 - user-defined iterators
 - lists/tuples/dictionary

Builtin Iterable: range()

- `range(start, stop, step)`
 - Generate sequence of numbers from *start* (inclusive) to *stop* (exclusive), incremented by *step*
 - *start* and *step* are optional arguments
 - Default Values:
 - *start* = 0
 - *step* = 1

Repetition Flow Control: “For”

Syntax

```
for i in range(n,m):  
    statement(s)
```

Example

```
for i in range(0,5):  
    print(i)
```

0
1
2
3
4

Exclusive

Repetition Flow Control: “For”

Example

```
for i in range(0,5):  
    print(i)
```

0
1
2
3
4

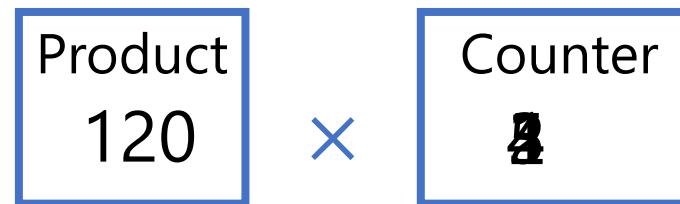
Interpreted as

```
i=0  
print(i)  
i=1  
print(i)  
i=2  
print(i)  
i=3  
print(i)  
i=4  
print(i)
```

Iterative Factorial

Idea

- Start with 1, multiply by 2, multiply by 3, ... , multiply by n.
- $n! = 1 \times 2 \times 3 \cdots \times n$



Iterative Factorial

- $n! = 1 \times 2 \times 3 \cdots \times n$
- **Computationally**
- Starting:
 - product = 1
 - counter = 1
- Iterative (repeating) step:
 - product \leftarrow product \times counter
 - counter \leftarrow counter + 1
- End:
 - product contains the result

Computing Factorial

- $n! = 1 \times 2 \times 3 \cdots \times n$
- Factorial rule:
 - product \leftarrow product \times counter
 - counter \leftarrow counter + 1

```
def factorial(n):  
    product = 1  
    for counter in range(2, n+1):  
        product = product * counter  
    return product
```

non-inclusive.

Up to n.
↓

```
factorial(6)
```

product counter

1	2
1x2 = 2	3
1x2x3=6	4
1x2x3x4=24	5
120	6
720	7

for loop

- **for** <var> **in** <sequence>:
- <body>
- **sequence**
 - a sequence of values
- **var**
 - variable that take each value in the sequence
- **body**
 - statement(s) that will be evaluated for each value in the sequence

range function

- `range([start,] stop[, step])`
 - creates a **sequence** of integers
 - from start (inclusive) to stop (non-inclusive)
 - incremented by step
- May omit

Examples

```
for i in range(10):  
    print(i)
```

```
for i in range(3, 10):  
    print(i)
```

```
for i in range(3, 10, 4):  
    print(i)
```

Example

Flipping coins

Flipping a coin

- A coin is “fair” if the probability of getting a head is equal to a tail
 - $P(\text{head}) == P(\text{tail}) == 0.5$
- How to test a coin is fair?
- Flip 1000 times!

Write a Pseudo Code for the Experiment

- I will flip a coin 1000 times and FOR EACH FLIP
 - I will record how many times I had flipped
 - If it is a head, I will record the number of heads



What you
repeat for
EACH time

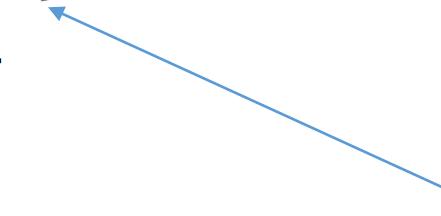


Flipping Coins

```
import random

def flipCoins():
    print('I will flip a coin 1000 times. ')
    print('Guess how many times it will come up heads. ')
```

```
heads = 0
for flip in range(0,1000):
    if random.randint(0, 1) == 1:
        heads = heads + 1
```



Randomly
generate
either 0 or 1

while loop

- **while** <expression>:
 - <body>
- expression
 - Predicate (condition) to stay within the loop
- body
 - Statement(s) that will be evaluated if predicate is True

Repetition (Infinite)

```
while True:
```

```
    print('Ah...')
```

```
a = 0
```

```
while a > 0:
```

```
    a = a + 1
```

```
    print(a)
```

Repetition

Syntax

```
while <expr>:  
    statement(s)
```

Example

```
>>> a = 0  
>>> while a < 5:  
        a = a + 1  
        print(a)
```

1
2
3
4
5

indentation

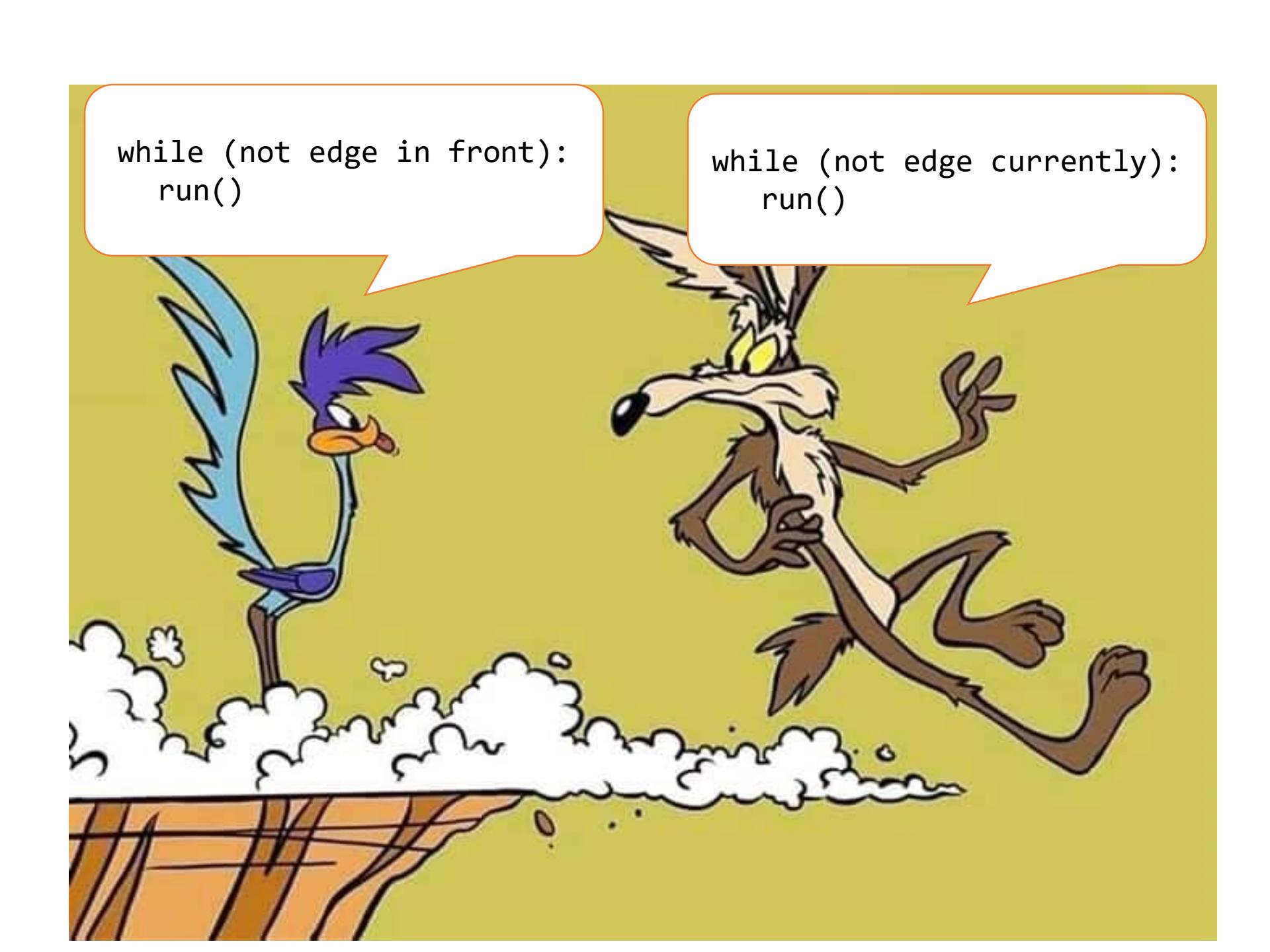
Another Iterative process

```
def factorial(n):
    product, counter = 1, 1
    while counter <= n:
        product = (product *
                    counter)
        counter = counter + 1
    return product

factorial(6)
```

product	counter
1	1
1	2
2	3
6	4
24	5
120	6
720	7

counter > n (7 > 6)
return product (720)



```
while (not edge in front):  
    run()
```

```
while (not edge currently):  
    run()
```

Another Iterative process

```
def factorial(n):  
    product, counter = 1, 1  
    while counter <= n:  
        product = (product *  
                   counter)  
        counter = counter + 1  
    return product  
  
factorial(6)
```

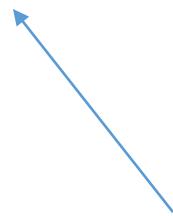
“`<=`” or “`<`” ?

product	counter
1	1
1	2
2	3
6	4
24	5
120	6
720	7

counter > n (7 > 6)
return product (720)

```
import random

def flipCoins():
    print('I will flip a coin 1000 times. ')
    print('Guess how many times it will come up heads. ')
flips = 0
heads = 0
while flips < 1000:
    if random.randint(0, 1) == 1:
        heads = heads + 1
    flips = flips + 1
```



Randomly
generate
either 0 or 1

Repetition (nested)

indentation

Syntax

```
while <expr>:  
    while <expr>:  
        statement(s)
```

Example

```
def nestedWhile():  
    i = 0  
    while i < 5:  
        i += 1  
        j = 0  
        while j < 3:  
            j += 1  
            print ('#' * j)
```

Repetition, a Very Common Pattern

9 out of 10 times you will

do

```
>>> a = 0
```

```
>>> while a < N:
```

```
    a = a + 1
```

do something

For loop

```
for i in range(0,N):
```

```
    do something
```

Three Types of Loops

- Must run exactly N times
- **Run any number of times**
- Run at most N times
 - Check all True (or check all False)
 - Find any True (or False)

Sum Some Numbers

```
Please enter a number or type 'bye' to sum:  
12  
Please enter a number or type 'bye' to sum:  
99  
Please enter a number or type 'bye' to sum:  
123  
Please enter a number or type 'bye' to sum:  
2  
Please enter a number or type 'bye' to sum:  
bye  
The sum of all numbers is 236
```

Sum Some Numbers

- You do not know how many numbers will the user enter

```
def sumNumbers():
    sumSoFar = 0
    print('Please enter a number or type \'bye\' to sum:')
    num = input()
    while num != 'bye':
        sumSoFar += int(num)
        print('Please enter a number or type \'bye\' to sum:')
        num = input()
    print(f'The sum of all numbers is {sumSoFar}')

sumNumbers()
```

Why do we need to repeat these?

```
def sumNumbers():
    sumSoFar = 0
    print('Please enter a number or type \'bye\' to sum:')
    num = input()
    while num != 'bye':
        sumSoFar += int(num)
        print('Please enter a number or type \'bye\' to sum:')
        num = input()
    print(f'The sum of all numbers is {sumSoFar}')

sumNumbers()
```

Loop Terminating Condition

- Must run exactly N times
- **Run any number of times**
- Run at most N times
 - Check all True (or check all False)
 - Find any True (or False)
- If we do not know how many times do we need, when do we know we finish looping?

Loop Terminating Condition

- When will the loop terminate?

```
def sumNumbers():
    sumSoFar = 0
    print('Please enter a number or type \'bye\' to sum:')
    num = input()
    while num != 'bye':
        sumSoFar += int(num)
        print('Please enter a number or type \'bye\' to sum:')
        num = input()
    print(f'The sum of all numbers is {sumSoFar}')
```

- When

- `num == 'bye'`? Or
- `num != 'bye'` ?

Loop Terminating Condition

- When will the loop terminate?

```
def sumNumbers():
    sumSoFar = 0
    print('Please enter a number or type \'bye\' to sum:')
    num = input()
    while num != 'bye':
        sumSoFar += int(num)
        print('Please enter a number or type \'bye\' to sum:')
        num = input()
    print(f'The sum of all numbers is {sumSoFar}')
```

- The loop body will keep repeating if the condition is true
- You break the loop if the condition is not true anymore

Three Types of Loops

- Must run exactly N times
- Run any number of times
- **Run at most N times**
 - **Check all True** (or check all False)
 - Find any True (or False)

Check if a String is all Alphabets

- Given a string, example, ‘abc123’
- Check if all the characters are alphabets
 - Return True or False
- In real life, how do you check?
- For example, if you are the teacher with a lot of test scripts, how do you check if “all are marked”?

Goal: All are Alphabet

- I just need one of the answers:
 - Yes: if all are **alphabet**
 - No: if there exists one not **alphabet**
- Combining
 - You check the **character** one-by-one
 - If the current one is **alphabet**, do nothing, check the next
 - Else return "No"!
 - Until finishing all **character** all checked, return "Yes"

Which line you
repeat a lot of
times?

Goal: All are Alphabet

- I just need one of the answers:
 - Yes: if all are **alphabet**
 - No: if there exists one not **alphabet**
- Combining
 - You check the **character** one-by-one
 - If the current one is **alphabet**, do nothing, check the next
 - Else return "No"!
 - Until finishing all **character** all checked, return "Yes"

In Python, you
indent the
statements
needed to be
loop

Goal: All are Alphabets

- Combining
 - You check the **character** one-by-one
 - If the current one is NOT **alphabet**, return “No”!
 - Until finishing all **character** all checked, return “Yes”

```
def checkAllAlpha(string) :  
    l = len(string)  
    for i in range(l) :  
        if not isAlphabet(string[i]) :  
            return False  
    return True
```

Goal: All are Alphabets

- Combining
 - You check the **character** one-by-one
 - If the current one is NOT **alphabet**, return "No"!
 - Until finishing all **character** all checked, return "Yes"

```
def checkAllAlpha(string):  
    l = len(string)  
    for i in range(l):  
        if not isAlphabet(string[i]):  
            return False  
    return True
```

Goal: All are Alphabets

- Combining
 - You check the **character** one-by-one
 - If the current one is NOT **alphabet**, return "No!"
 - Until finishing all **character** all checked, return "Yes"

```
def checkAllAlpha(string) :  
    l = len(string)  
    for i in range(l) :  
        if not isAlphabet(string[i]) :  
            return False  
    return True
```

Goal: All are Alphabets

- Combining
 - You check the **character** one-by-one
 - If the current one is NOT **alphabet**, return "No"!
 - Until finishing all **character** all checked, return "Yes"

```
def checkAllAlpha(string):  
    l = len(string)  
    for i in range(l):  
        if not isAlphabet(string[i]):  
            return False  
    return True
```

How many times?

- How many times we reach this line if $\text{len}(\text{string}) = N$?
 - You check the **character** one-by-one
 - If the current one is NOT **alphabet**, return "No"!
 - Until finishing all **character** all checked, return "Yes"

```
def checkAllAlpha(string) :  
    l = len(string)  
    for i in range(l) :  
        if not isAlphabet(string[i]) :  
            return False  
    return True
```

- Worst case: N times
- But maybe less than N

Provided that we have a function

- To check if a character is an alphabet

```
def isAlphabet(s):  
    if s >= 'a' and s <= 'z':  
        return True  
    if s >= 'A' and s <= 'Z':  
        return True  
    return False
```

Three Types of Loops

- Must run exactly N times
- Run any number of times
- **Run at most N times**
 - Check all True (**or check all False**)
 - Find any True (or False)
- Check all True similar to check all False
 - E.g. check if all characters are **NOT** alphabet?

```
def checkAllNotAlpha(string):  
    l = len(string)  
    for i in range(l):  
        if isAlphabet(string[i]):  
            return False  
    return True
```

Three Types of Loops

- Must run exactly N times
- Run any number of times
- **Run at most N times**
 - Check all True (or check all False)
 - **Find any True (or False)**
- Check any True?
 - Return the reverse of “check all False”

```
def checkAnyAlpha(string):  
    return not checkAllNotAlpha(string)
```

“For” vs “While”

- When to use “for” and when to use “while”?
- “For”
 - You know how many times before hand
 - Namely, anything in the body of the loop will NOT change the number of times you repeat the loop
 - E.g. printing out all the data in a spreadsheet
- “While”
 - You may not know how many times you need to repeat
 - The number of times is depended on the “condition”, in which, may change unpredictably inside the loop
 - E.g. while the player haven’t guess the right answer, keep guessing

Lastly: break & continue

```
for j in range(10):  
    print(j)  
    if j == 3:  
        break  
print("done")
```

0
1
2
3 Break out
done
of loop

```
for j in range(10):  
    if j % 2 == 0:  
        continue  
    print(j)  
print("done")
```

1 Continue with
3 next value
5
7
9
done

Let's play a game

```
>>> guessANum()
```

```
I have a number in mind between 0 and 99
```

```
Guess a number: 50
```

```
Too big
```

```
Guess a number: 25
```

```
Too big
```

```
Guess a number: 12
```

```
Too big
```

```
Guess a number: 6
```

```
Too small
```

```
Guess a number: 9
```

```
Too big
```

```
Guess a number: 7
```

```
Bingo!!!
```

```
>>>
```

guessANum.py

```
import random

def guessANum():
    secret = random.randint(0,99)      # 0 <= secret <= 99
    guess = -1
    print('I have a number in mind between 0 and 99')
    while guess != secret:
        guess = int(input('Guess a number: '))
        if guess == secret:
            print('Bingo!!! You got it! ')
        elif guess < secret:
            print('Your number is too small')
        else:
            print('Your number is too big')

guessANum()
```

Repeat
until the
condition
is **False**

guessANum.py

```
import random

def guessANum():
    secret = random.randint(0,99)      # 0 <= secret <= 99
    guess = -1
    print('I have a number in mind between 0 and 99')
    while guess != secret:
        guess = int(input('Guess a number: '))
        if guess == secret:
            print('Bingo!!! The answer is ' + str(secret))
        elif guess < secret:
            print('Your number is too small')
        else:
            print('Your number is too big')

guessANum()
```

Repeat
until the
condition
is False

How to write a love letter in Python

```
def show_my_love():
    everything = True
    you = everything
    my_mind = you
    while(my_mind == True):
        print('I love you')
```

How to write a love letter in Python

```
def show_my_love():
    everything = True      #Everything I say is true
    you = everything       #You are everything to me
    my_mind = you          #All my mind is filled with you

    # No 'if' in my love because it's unconditional

    while(my_mind == True): # My love is eternal
        print('I love you')

    # And there is no 'return' in my love
    # because I do not expect any
```

Tips

- A “while” or “if” block starts with a colon “:”
- Remember
 - When there is a colon, there are indentations
 - When there are indentations, before these there is a colon
- The inclusive/exclusive range is a pain



IT5001 Software Development Fundamentals

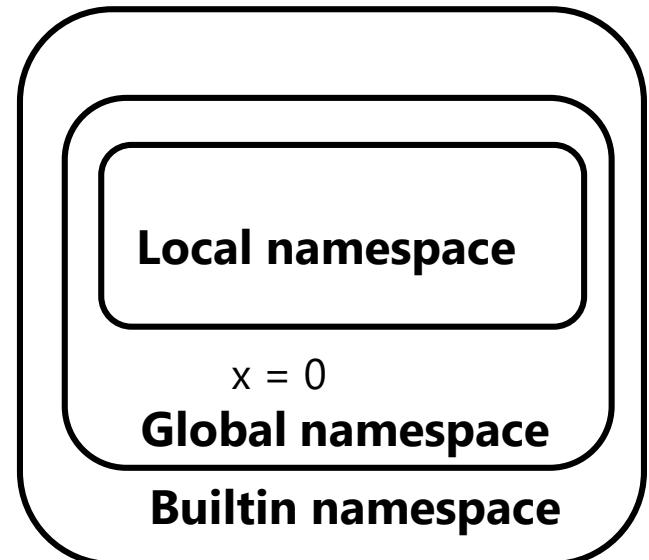
4. Functions, Scope, and Recursion

Sirigina Rajendra Prasad

Scope

Global vs Local Variables

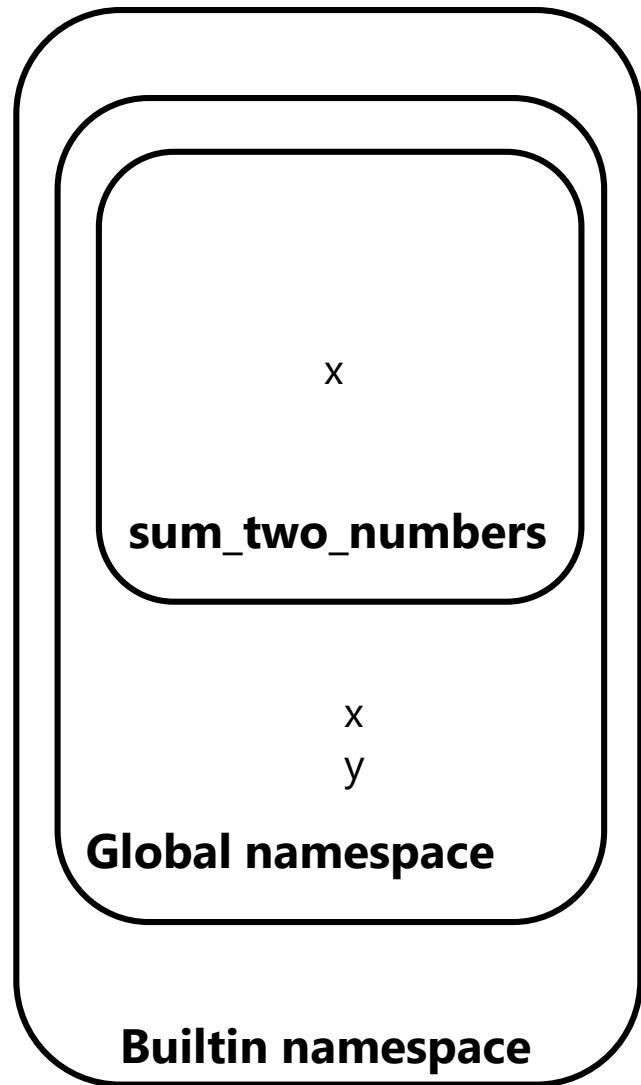
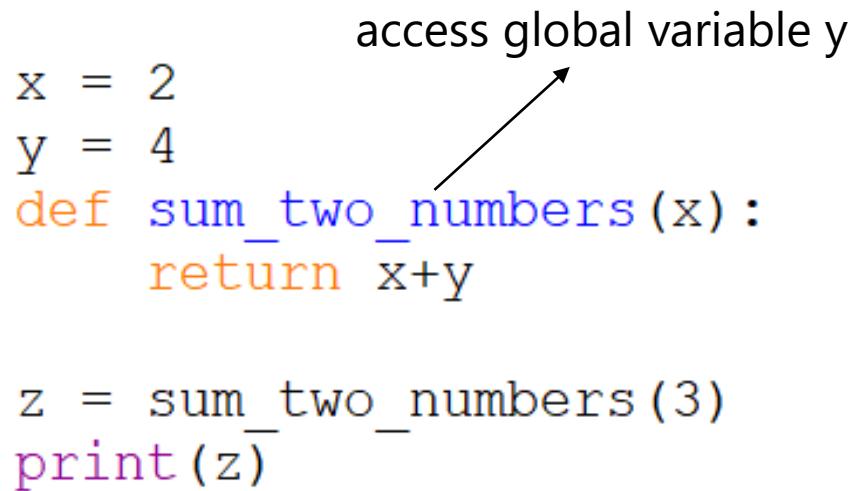
- A variable which is defined in the main body of a file is called a ***global*** variable. It will be **visible throughout the file**, and also inside any file which imports that file. EXCEPT...
- A variable which is defined inside a function is ***local*** to that function. It is accessible **from the point at which it is defined until the end of the function**, and exists for as long as the function is executing.
- The parameter names in the function definition behave like local variables, but they contain the values that we pass into the function when we call it.



Example

```
x = 2  
y = 4  
def sum_two_numbers(x):  
    return x+y  
  
z = sum_two_numbers(3)  
print(z)
```

access global variable y



Global Variable

```
x = 0  
def foo_printx():  
    print(x)  
  
foo_printx()  
print(x)
```

x = 0 Refers to

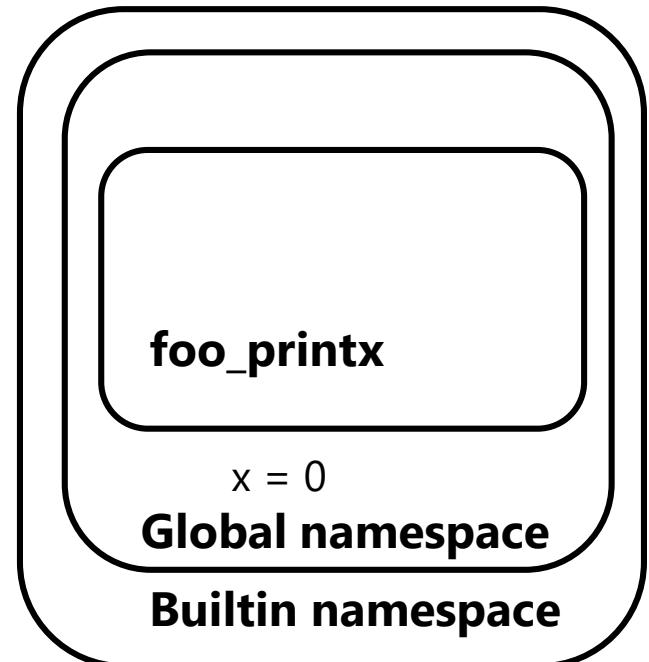
def foo_printx():
 print(x)

foo_printx()
print(x)

- This code will print

0

0



Global vs Local Variables

```
x = 0
```

```
def foo_printx():
```

```
    x = 999
```

```
    print(x)
```

```
foo_printx()
```

```
print(x)
```

Because, a new 'x'
is born here!

- This code will print

999

0

- The first '999' makes sense
- But why the second one is '0'?

Global vs Local Variables

```
x = 0
```

A Global 'x'

```
def foo_printx():
```

```
    x = 999
```

```
    print(x)
```

```
foo_printx()
```

```
print(x)
```

- This code will print
999

0

Scope of the local 'x'

Scope of the global 'x'

A local 'x' that is created within the function
foo_printx() and will 'die' after the function
exits

Global vs Local Variables

```
x = 0
```

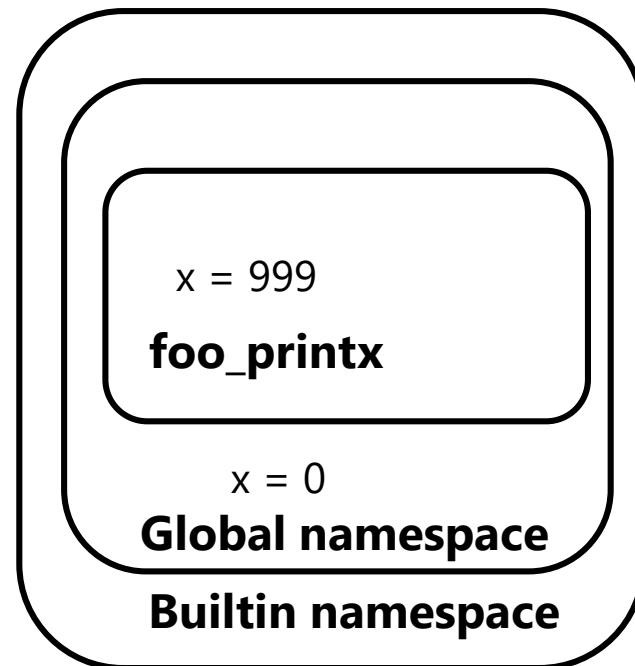
```
def foo_printx():
```

```
    x = 999
```

```
    print(x)
```

```
foo_printx()
```

```
print(x)
```



Crossing Boundary

- What if we really want to modify a global variable from inside a function?
- Use the “global” keyword
- (No local variable x is created)

```
x = 0
```

```
def foo_printx():  
    global x  
    x = 999  
    print(x)
```

```
foo_printx()  
print(x)
```

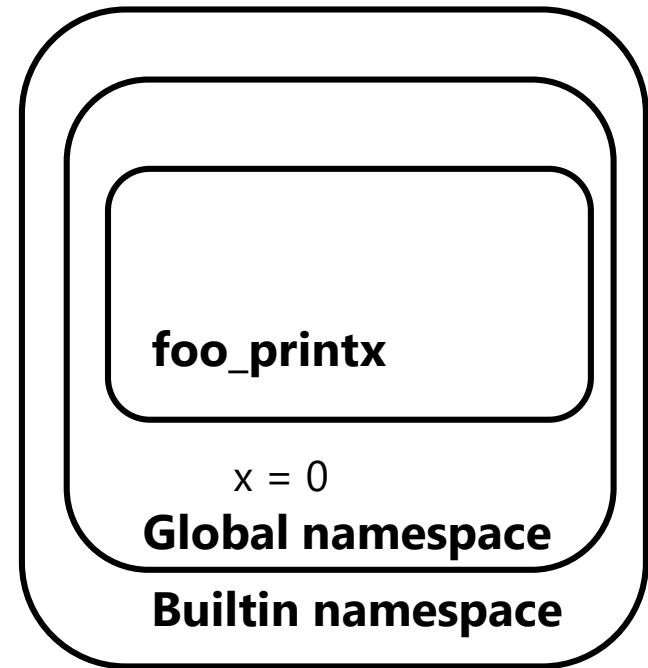
Output:
999
999

Crossing Boundary

```
x = 0
```

```
def foo_printx():
    global x
    x = 999
    print(x)
```

```
foo_printx()
print(x)
```



Output:
999
999

How about... this?

```
x = 0
```

```
def foo_printx():
    print(x)
    x = 999
    print(x)
```

```
foo_printx()
```

- Local or global?
- Error!
- Because the line “x=999” creates a local version of ‘x’
- Then the first print(x) will reference a **local** x that is not assigned with a value
- The line that causes an error

Parameters are LOCAL variables

Scope of x in p1

```
def p1(x):  
    print('Entering function p1')  
    output = p2(x)  
    print('Line before return in p1')  
    return output
```

Scope of x in p2

```
def p2(x):  
    print('Entering function p2')  
    output = p3(x)  
    print('Line before return in p2')  
    return output
```

Scope of x in p3

```
def p3(x):  
    print('Entering function p3')  
    output = x * x  
    print('Line before return in p3')  
    return output
```

Does not refer to

```
print(p1(3))
```

Practices (Convention)

- Global variables are VERY **bad**, especially if modification is allowed
- 99% of time, global variables are used as **CONSTANTS**
 - Variables that every function could access
 - But not expected to be modified

Convention:
Usually in all CAPs

```
POUNDS_IN_ONE_KG = 2.20462

def kg2pound(w):
    return w * POUNDS_IN_ONE_KG

def pound2kg(w):
    return w / POUNDS_IN_ONE_KG
```

Generator Functions

Generator Functions

```
def function(arg_1, arg_2,..., arg_n):  
    <statement>  
    <statement>  
  
    .  
  
    .  
  
    .  
  
    return <statement>
```

```
def generator_function(arg_1, arg_2,..., arg_n):  
    <statement>  
    <statement>  
  
    .  
  
    .  
  
    .  
  
    yield <statement>
```

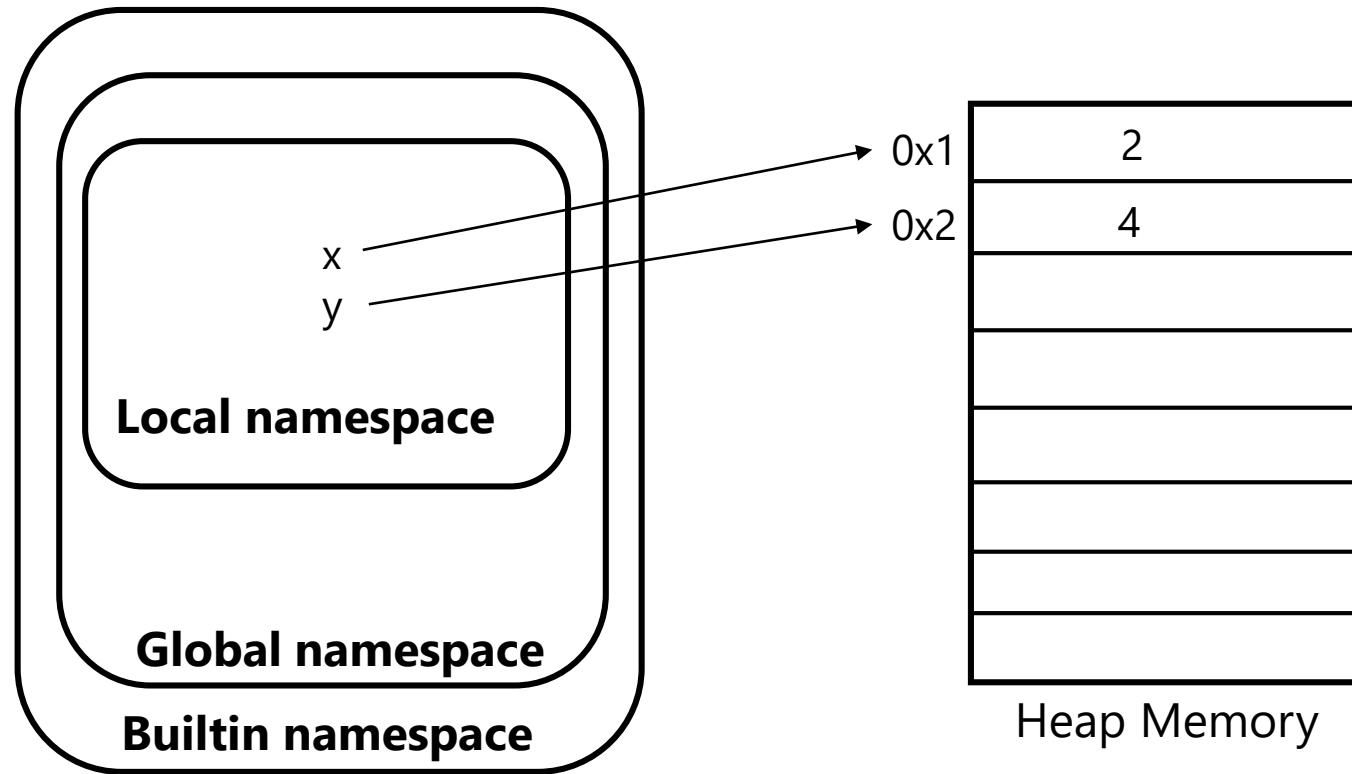
only difference

Return vs Yield

- With **return** statement:
 - State is not retained after the function returns the value
- With **yield** statement:
 - State of the function is retained between the calls
 - Can have many **yield** statements in sequence

What is the state of a function?

- Namespace and the objects that are referred by names



Generator Functions: Examples

```
def my_range(start = 0, stop = None, step = 1):  
    element = start  
    while element <= stop:  
        yield element  
        element = element + step
```

```
from math import inf  
for item in my_range(0, stop = inf):  
    print(item)
```

Generator Functions: Examples

```
def even_range(start = 0, stop = None, step = 1):
    element = start
    while element <= stop:
        if element%2 == 0:
            yield element
        element = element + step
```

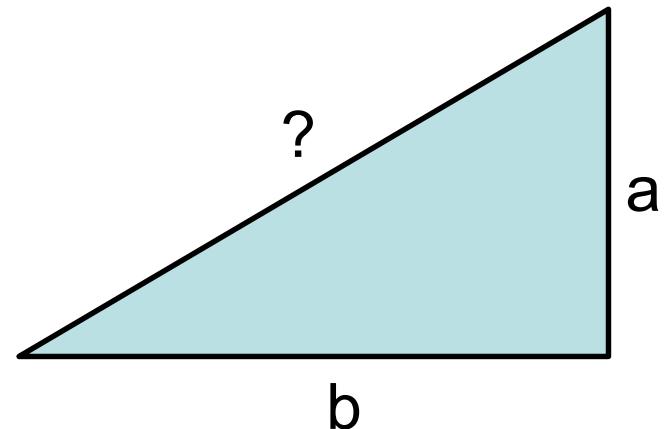
Calling Other Functions

Compare:

```
def hypotenuse(a, b):  
    return sqrt(sum_of_squares(a, b))
```

```
def sum_of_squares(x, y):  
    return square(x) + square(y)
```

```
def square(x):  
    return x * x
```



Versus:

```
def hypotenuse(a, b):  
    return sqrt((a*a) + (b*b))
```

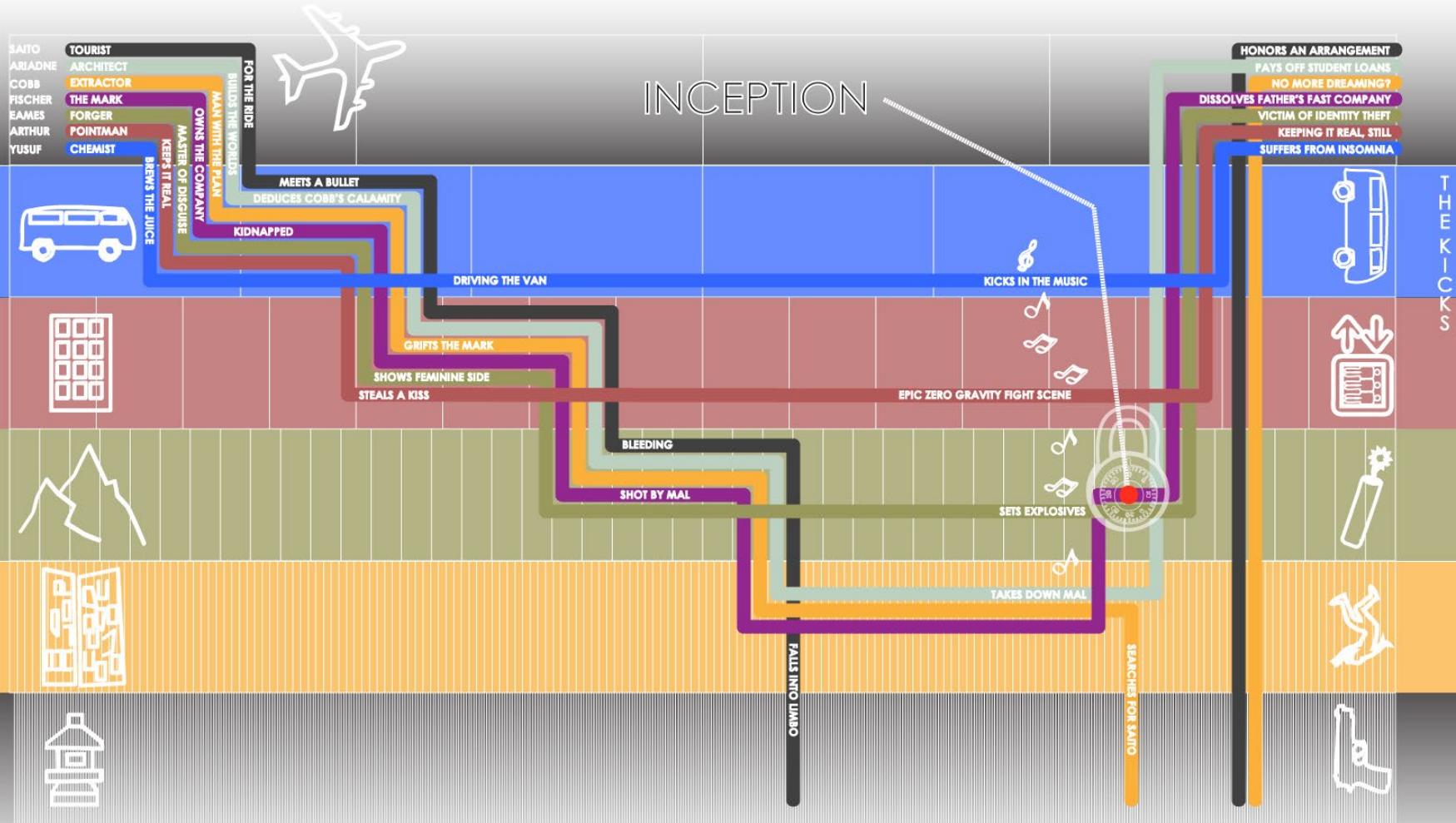


The Call Stack



reality

DEPTH OF DREAM

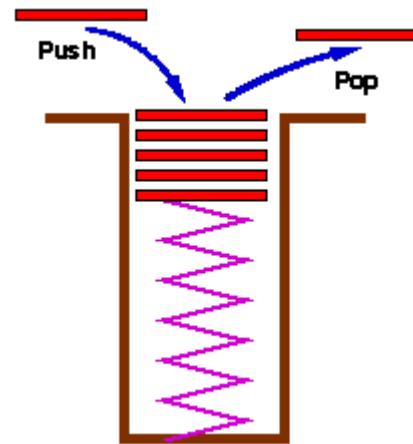


2010 INFOGRAPHIC
BY DANIEL WANG

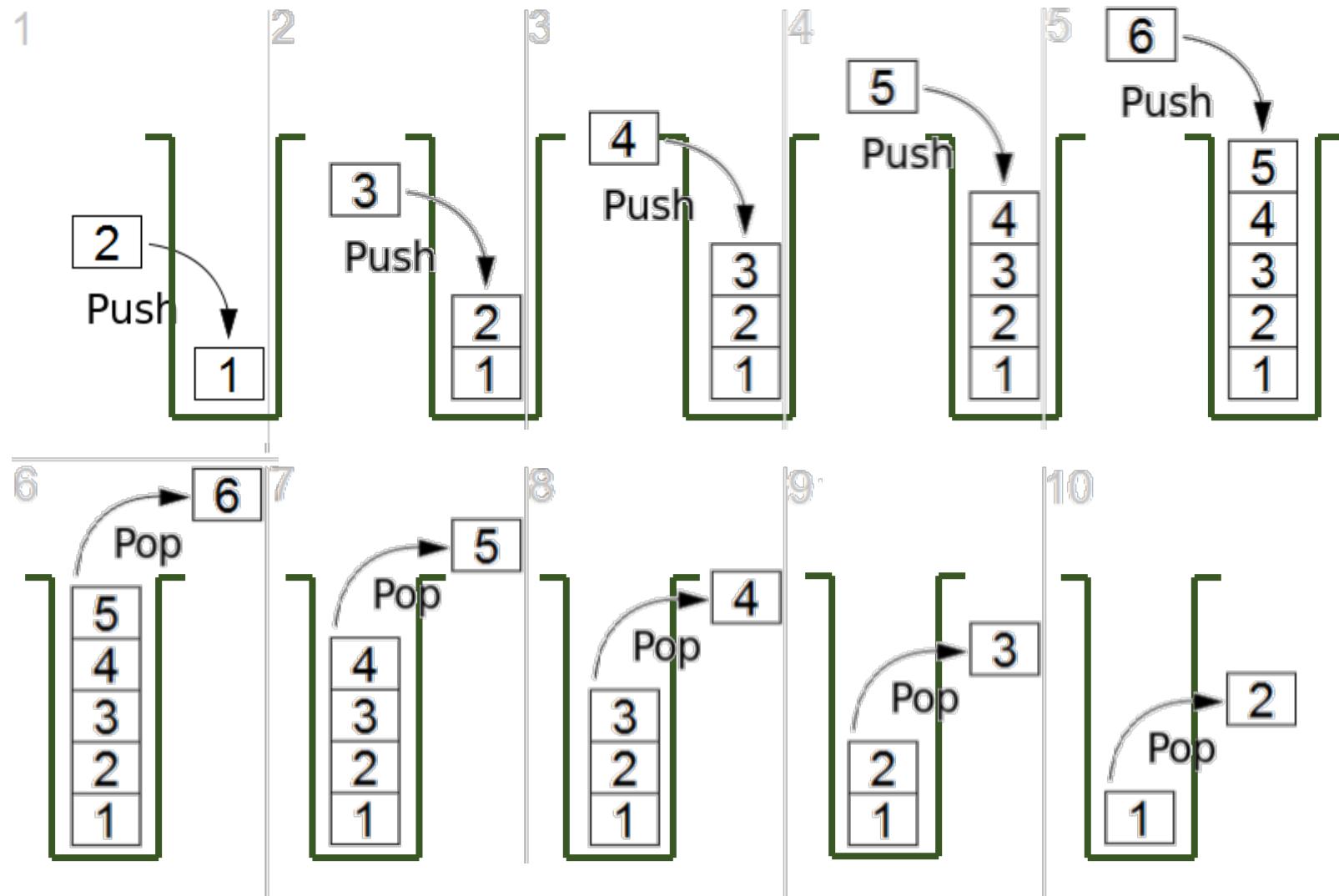
Michael Caine Just Ended An Eight Year Long Debate Over The Ending Of "Inception"

Stack

- First in last out order



First in Last Out



The Stack (or the Call Stack)

```
def p1(x):
    print('Entering function p1')
    output = p2(x)
    print('Line before return in p1')
    return output

def p2(x):
    print('Entering function p2')
    output = p3(x)
    print('Line before return in p2')
    return output

def p3(x):
    print('Entering function p3')
    output = x * x
    print('Line before return in p3')
    return output

print(p1(3))
```

The Stack (or the Call Stack)

```
>>> p1(3)
```

Entering function p1

Entering function p2

Entering function p3

Line before return in p3

Line before return in p2

Line before return in p1

9

FILO!

```
print(p1(3))
```

→ Going in
→ Exiting a
function

```
def p1(x):  
    print('Entering function p1')  
    output = p2(x)  
    print('Line before return in p1')  
    return output
```

```
→ def p2(x):
```

```
    print('Entering function p2')
```

```
    output = p3(x)
```

```
    print('Line before return in p2')
```

```
    return output
```

```
→ def p3(x):
```

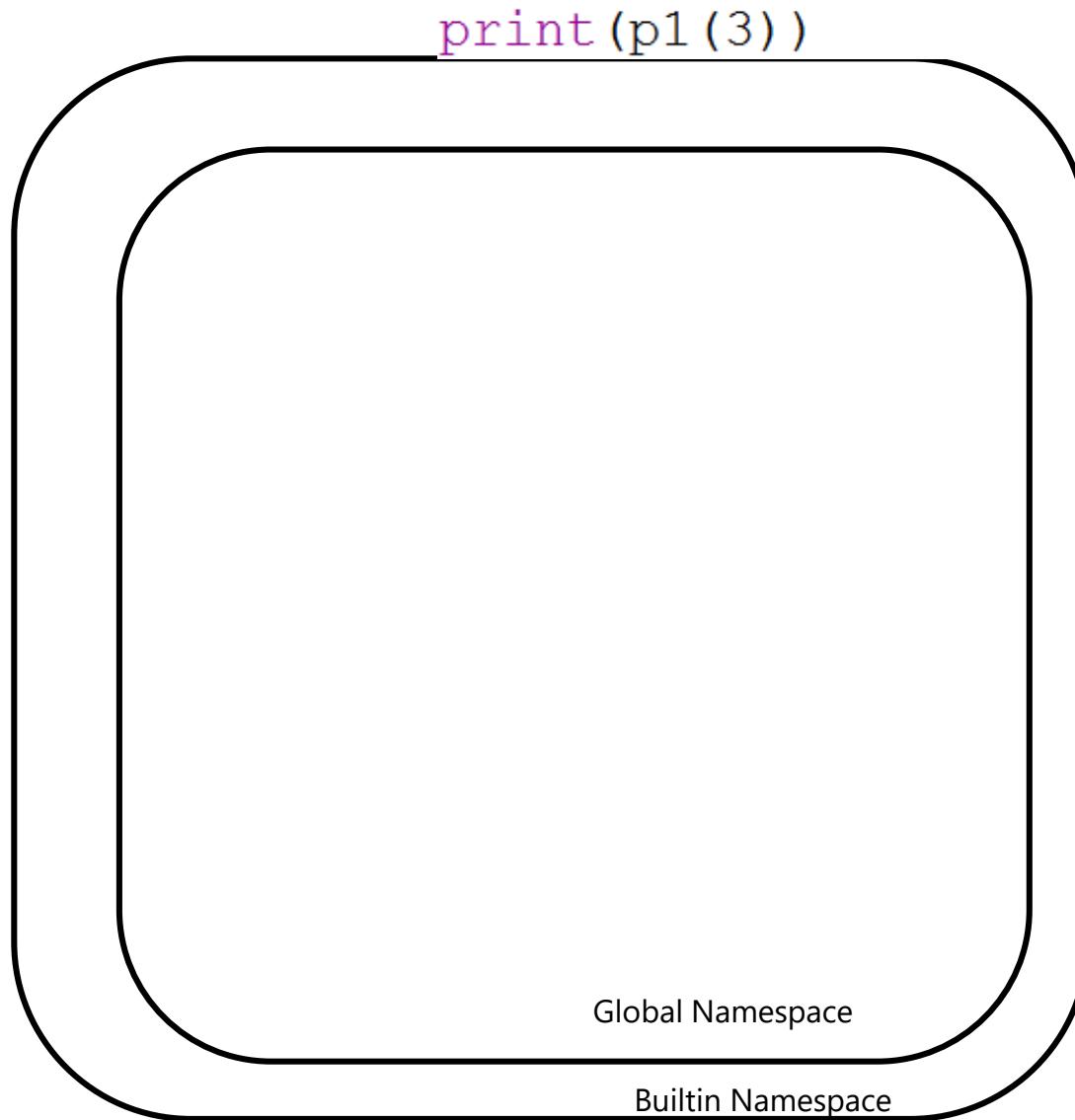
```
    print('Entering function p3')
```

```
    output = x * x
```

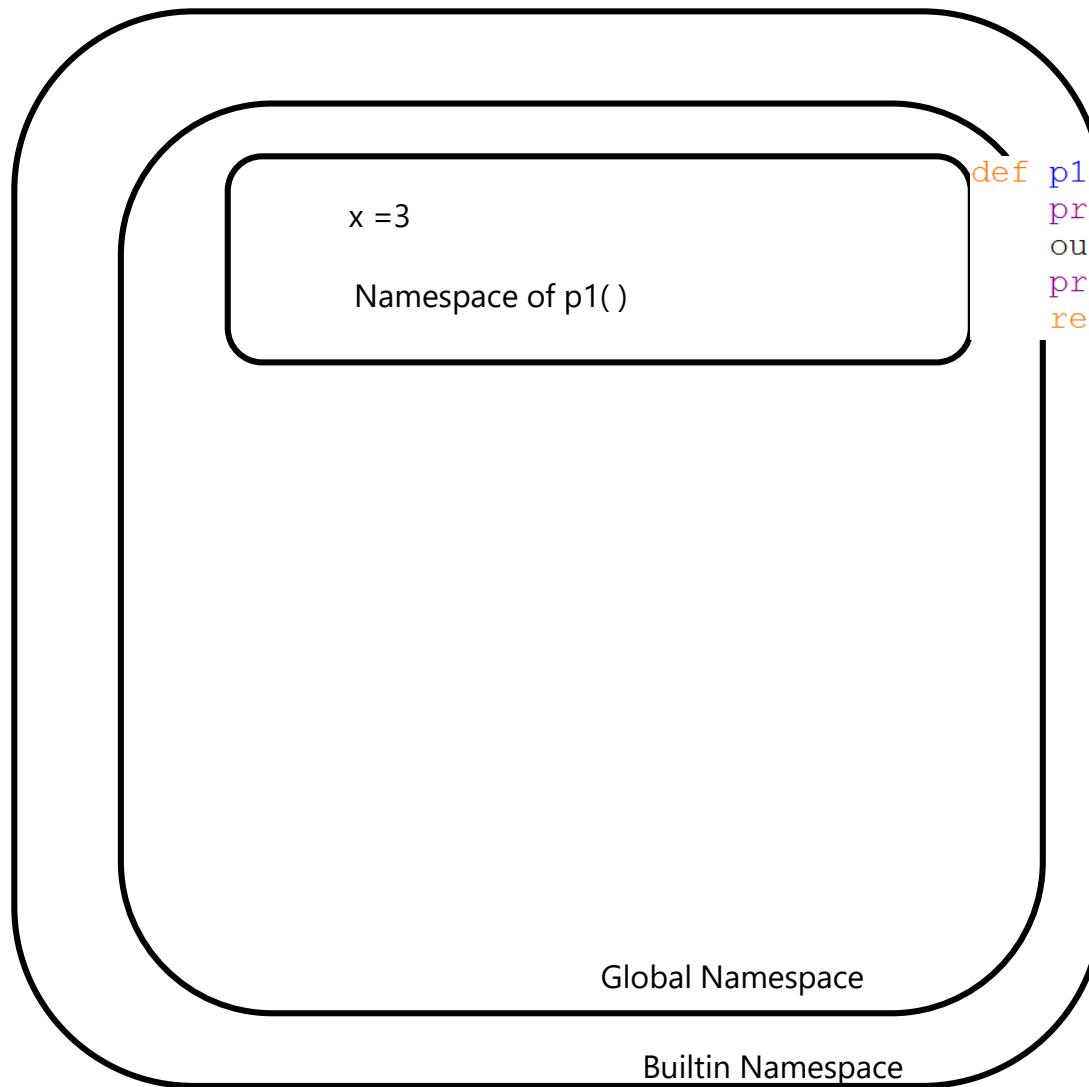
```
    print('Line before return in p3')
```

```
    return output
```

Namespaces: Calling Other Functions

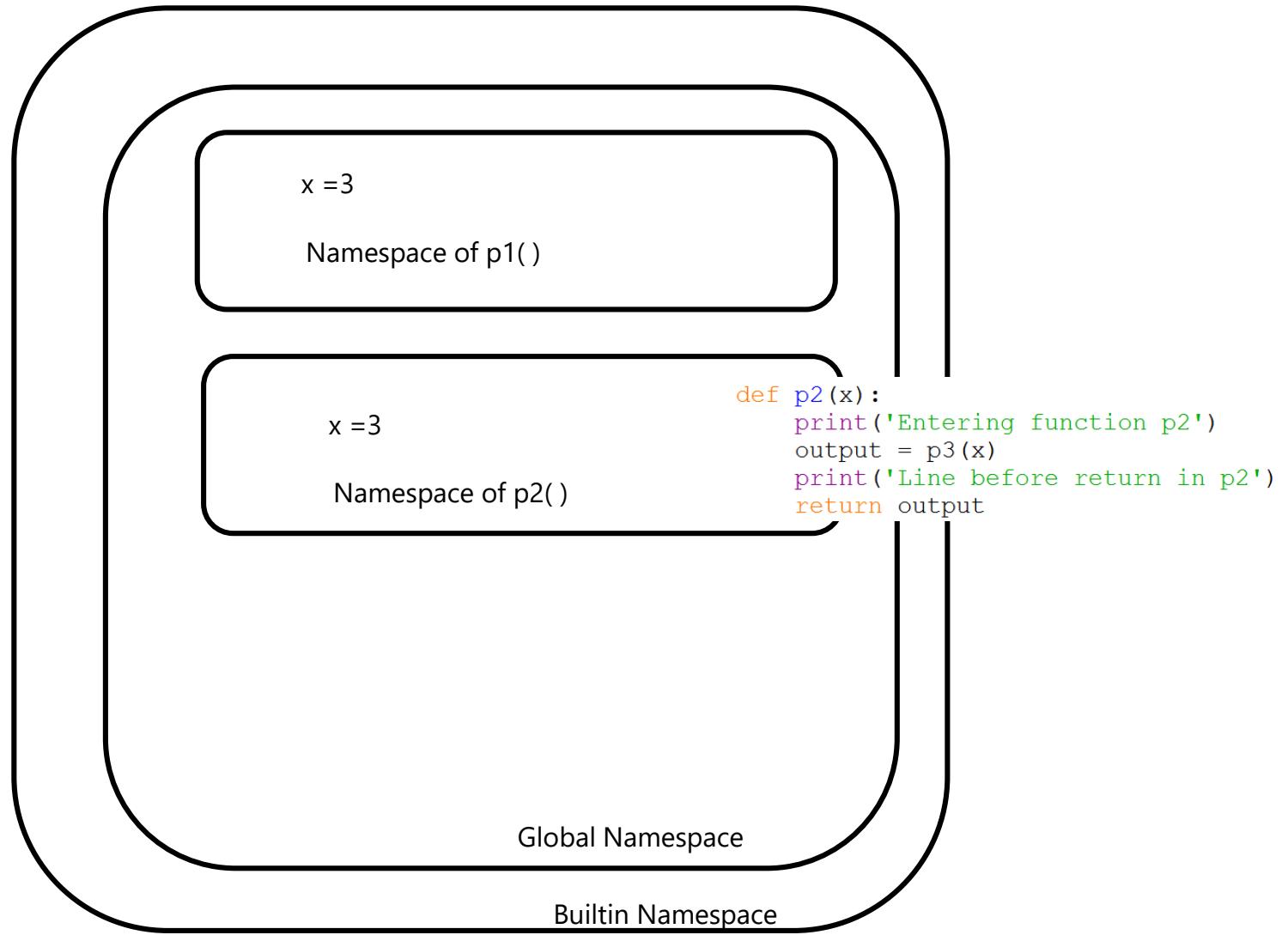


Namespaces: Calling Other Functions

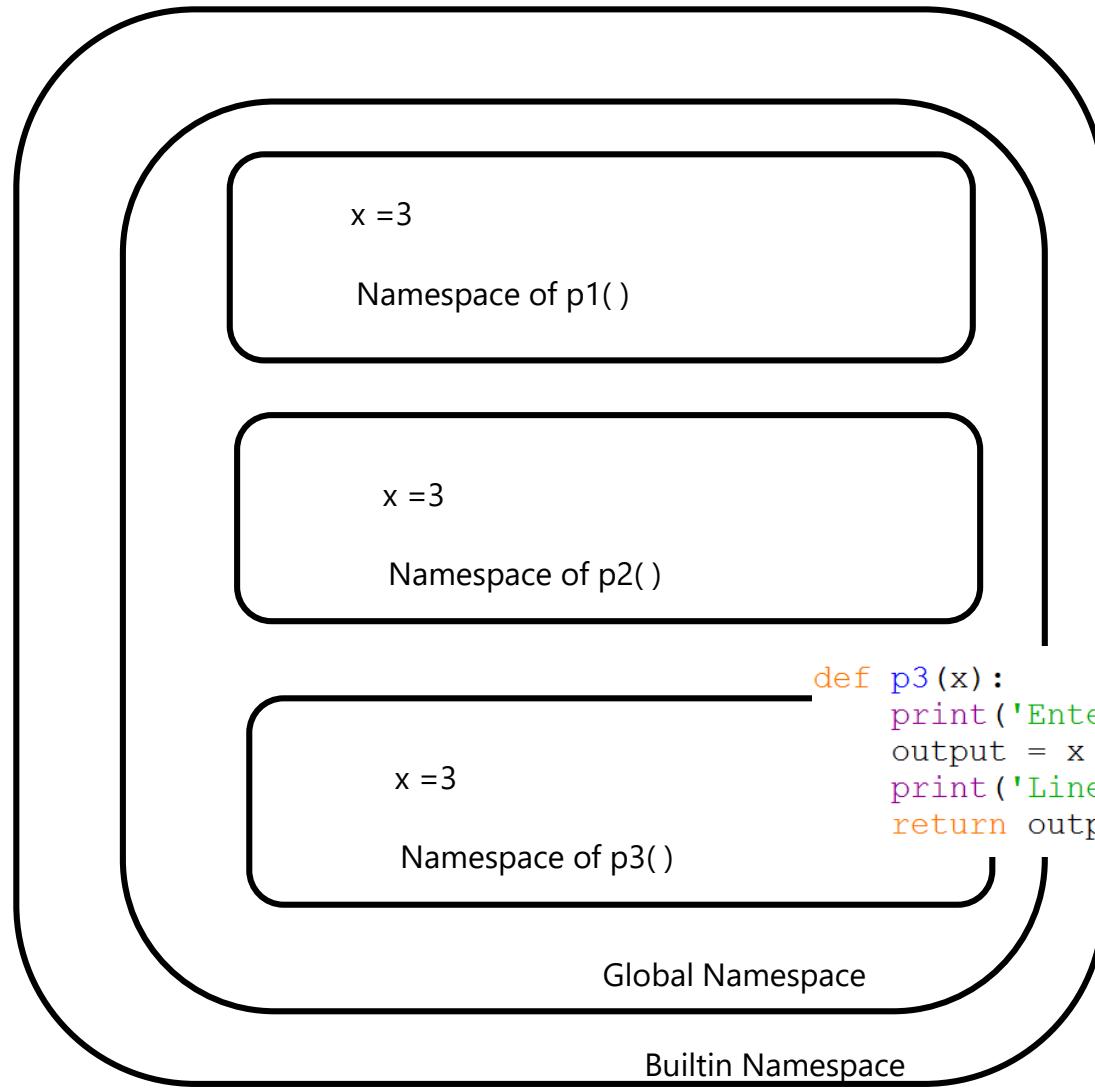


```
def p1(x):
    print('Entering function p1')
    output = p2(x)
    print('Line before return')
    return output
```

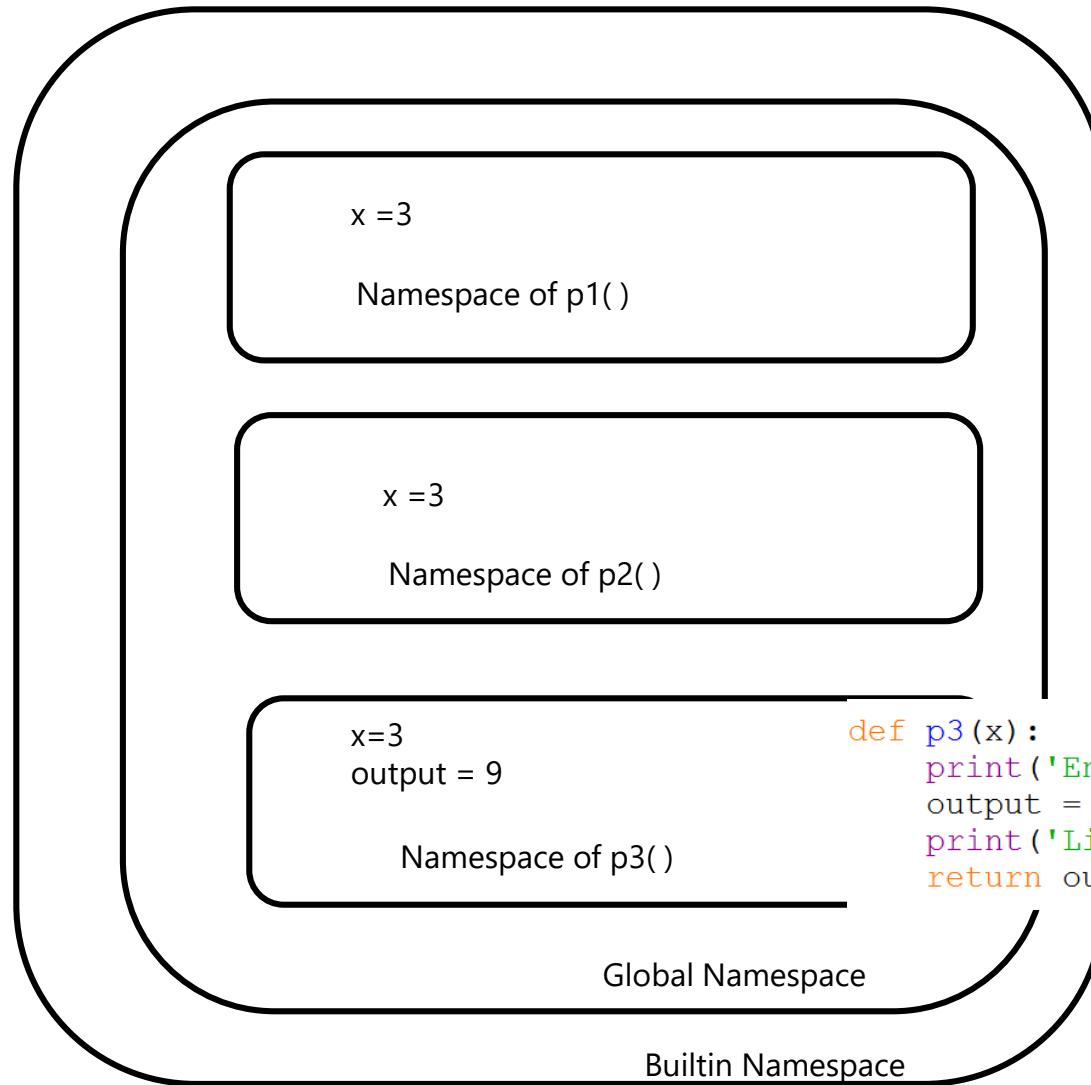
Namespaces: Calling Other Functions



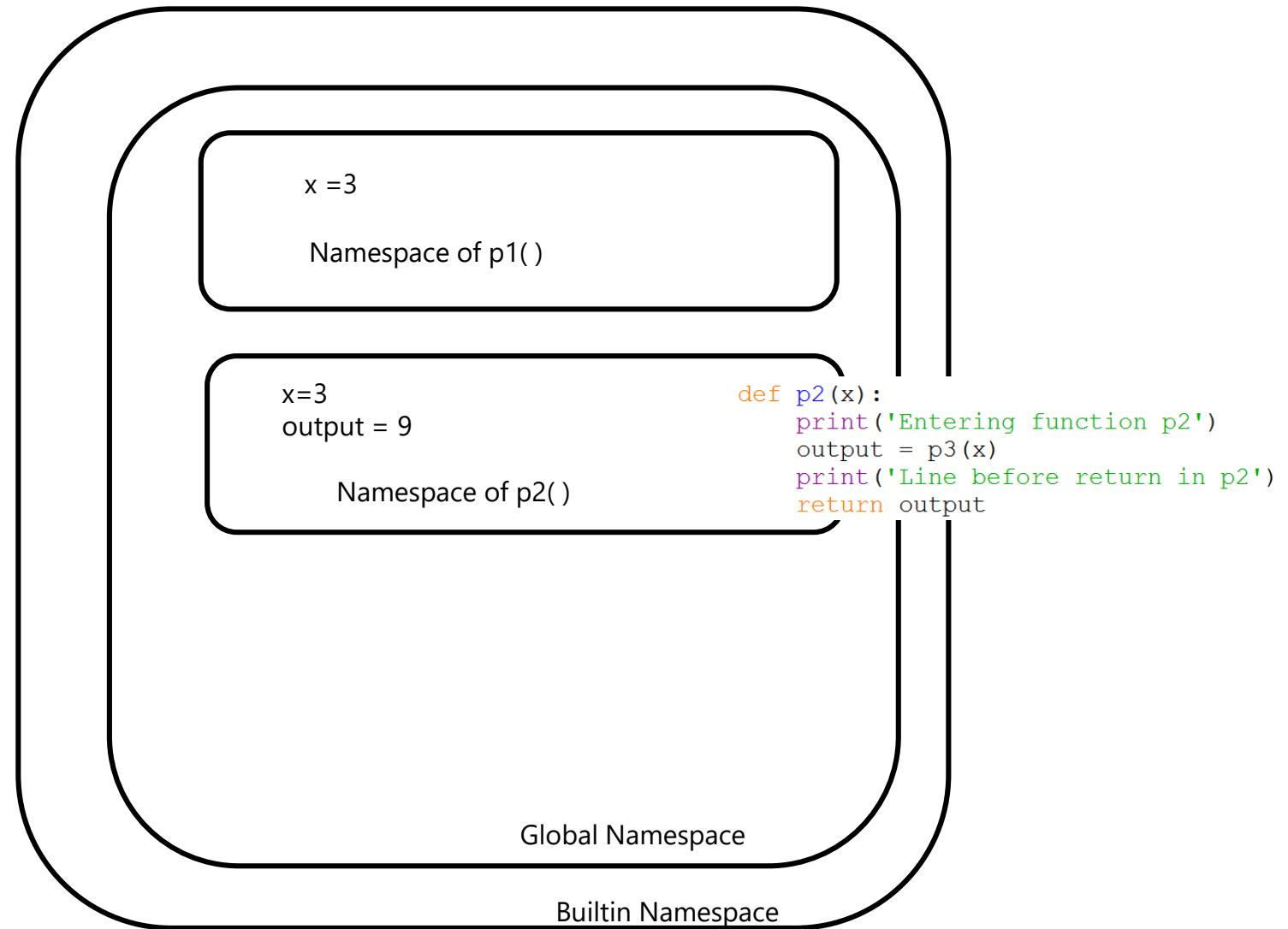
Namespaces: Calling Other Functions



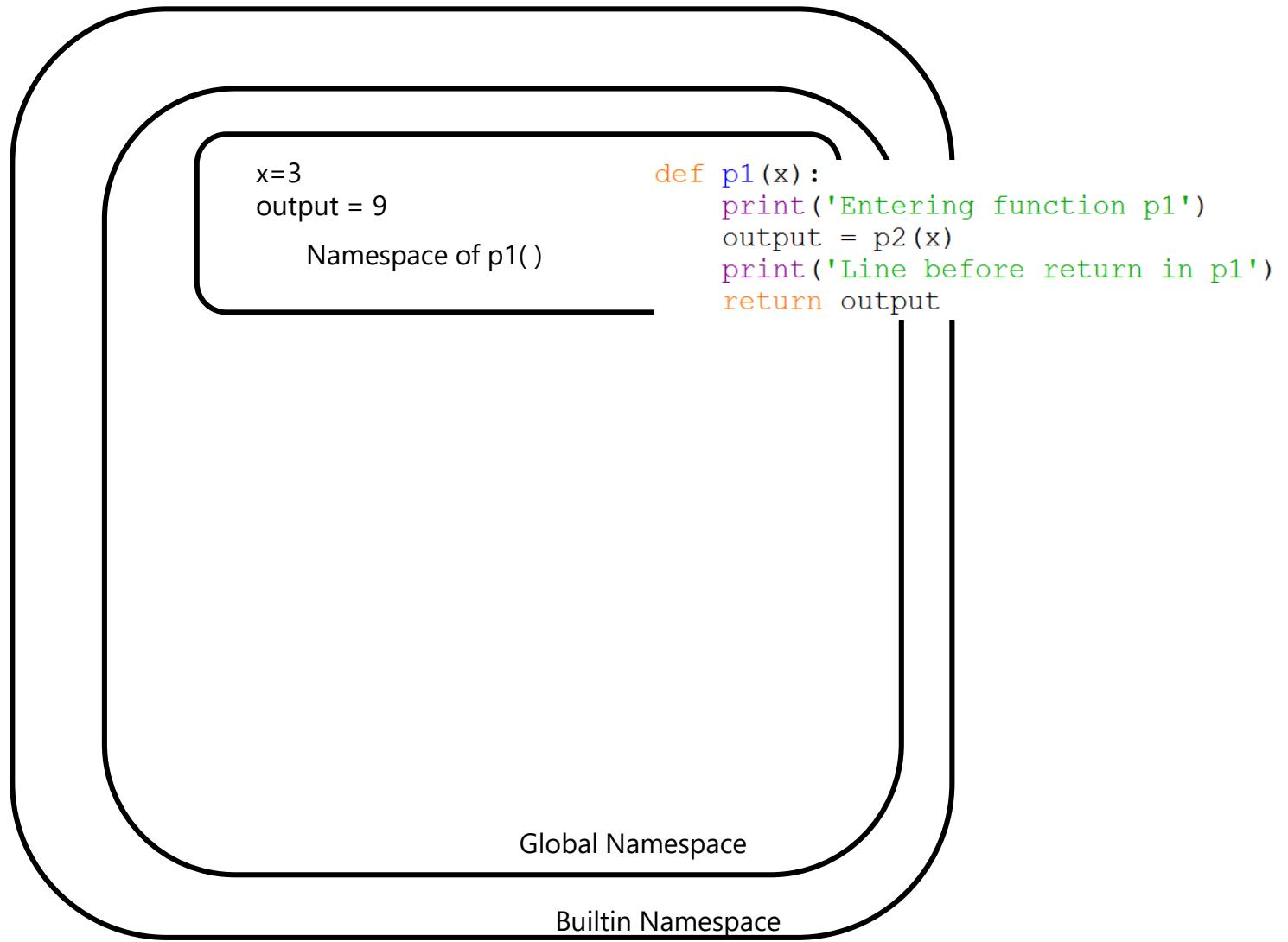
Namespaces: Calling Other Functions



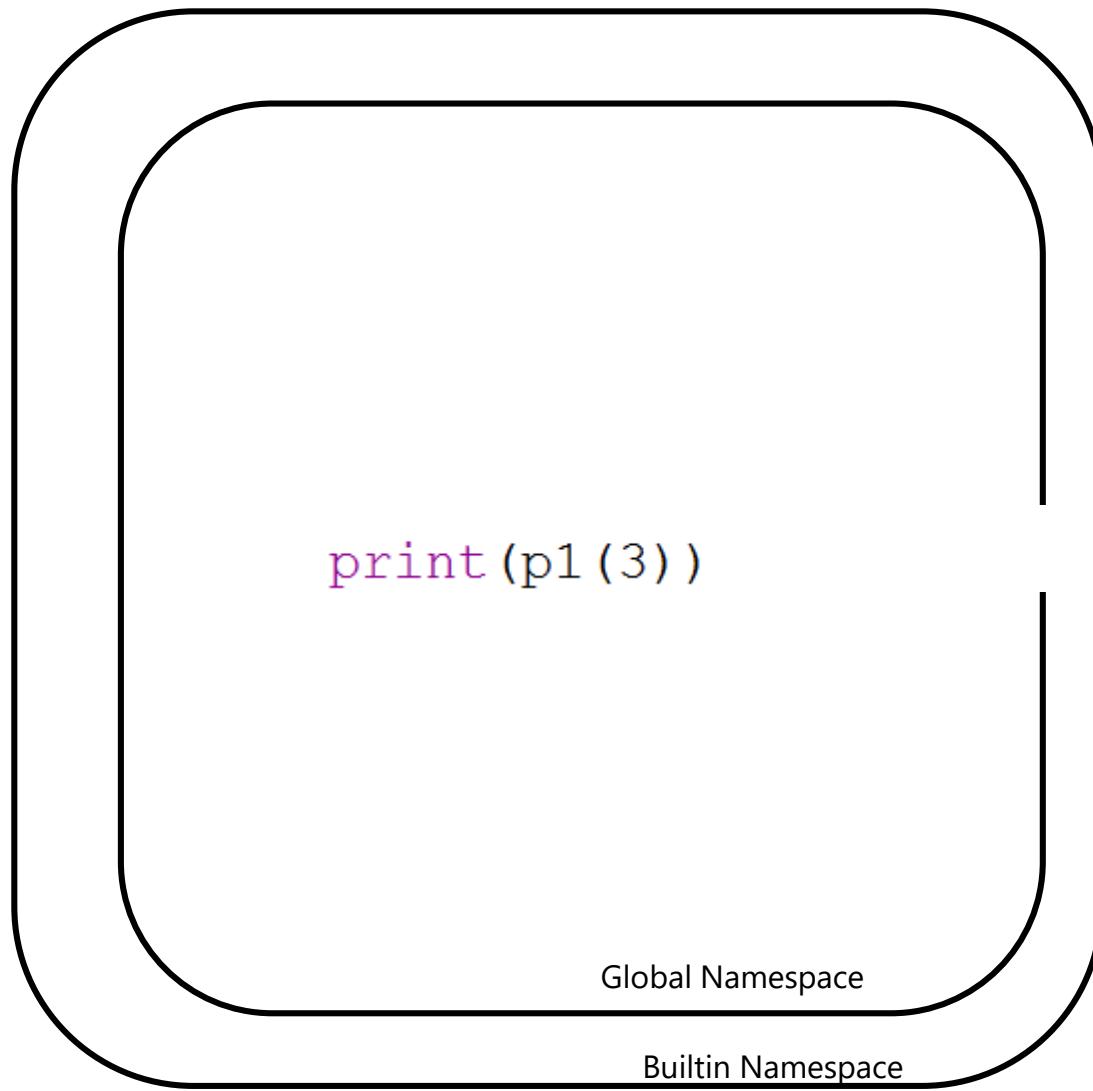
Namespaces: Calling Other Functions



Namespaces: Calling Other Functions



Namespaces: Calling Other Functions



Debug Control



Go Step Over Out Quit

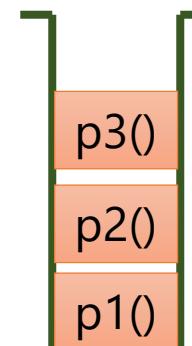
Stack Source
 Locals Globals

W03a Call Stack.py:16: p3()

```
'bdb'.run(), line 431: exec(cmd, globals, locals)
'_main_'.<module>(), line 1: p1(3)
'_main_'.p1(), line 3: output = p2(x)
'_main_'.p2(), line 10: output = p3(x)
> '_main_'.p3(), line 16: output = x * x
```

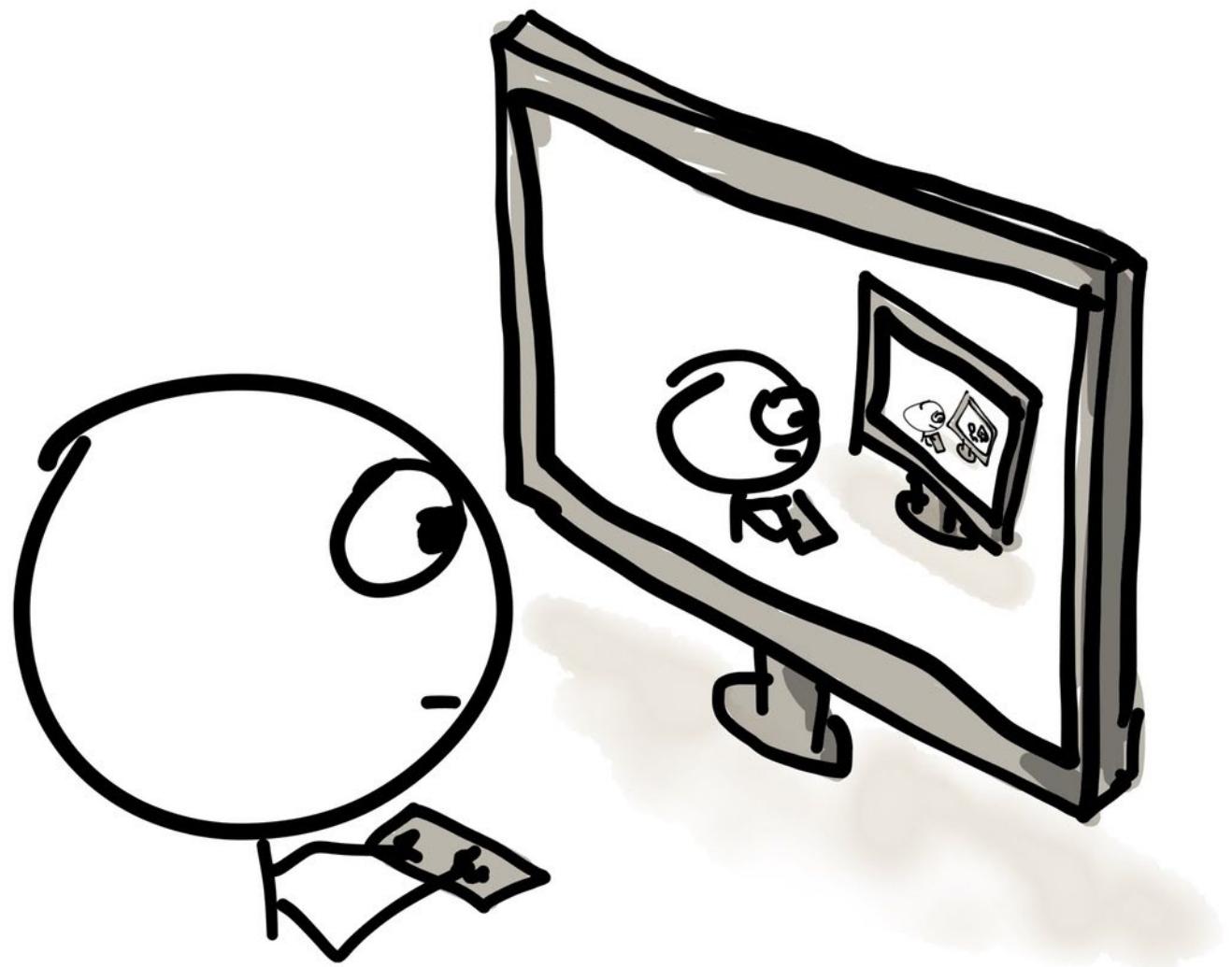
Locals

x 3



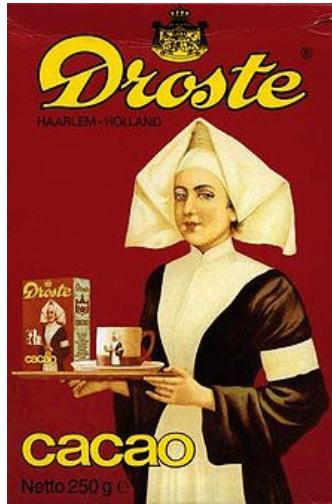
Recursion

Recursion

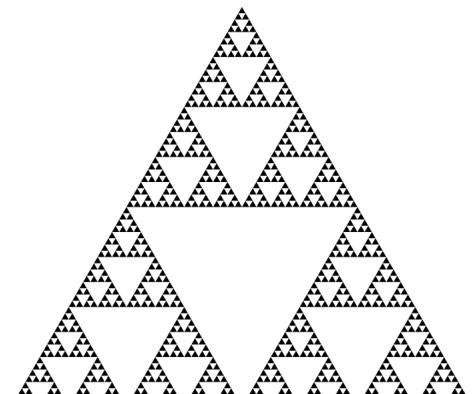


A Central Idea of CS

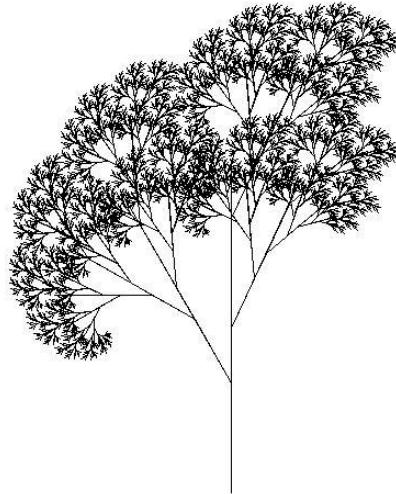
Some examples of recursion (inside and outside CS):



Droste effect

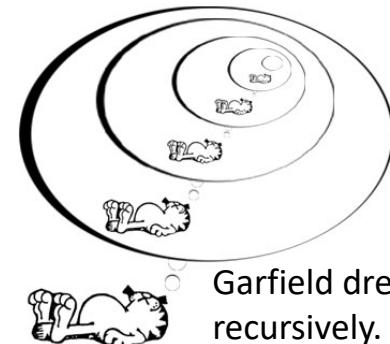


Sierpinski triangle



Recursive tree

Mandelbrot Fractal Endless Zoom



Garfield dreaming recursively.

Recursion

- A function that calls itself
- And extremely powerful technique
- Solve a big problem by solving a smaller version of itself
 - Mini-me



Recursive Functions

- Function calls itself

```
def factorial(n):  
    if n==1:  
        return 1  
    else:  
        return n*factorial(n-1)
```

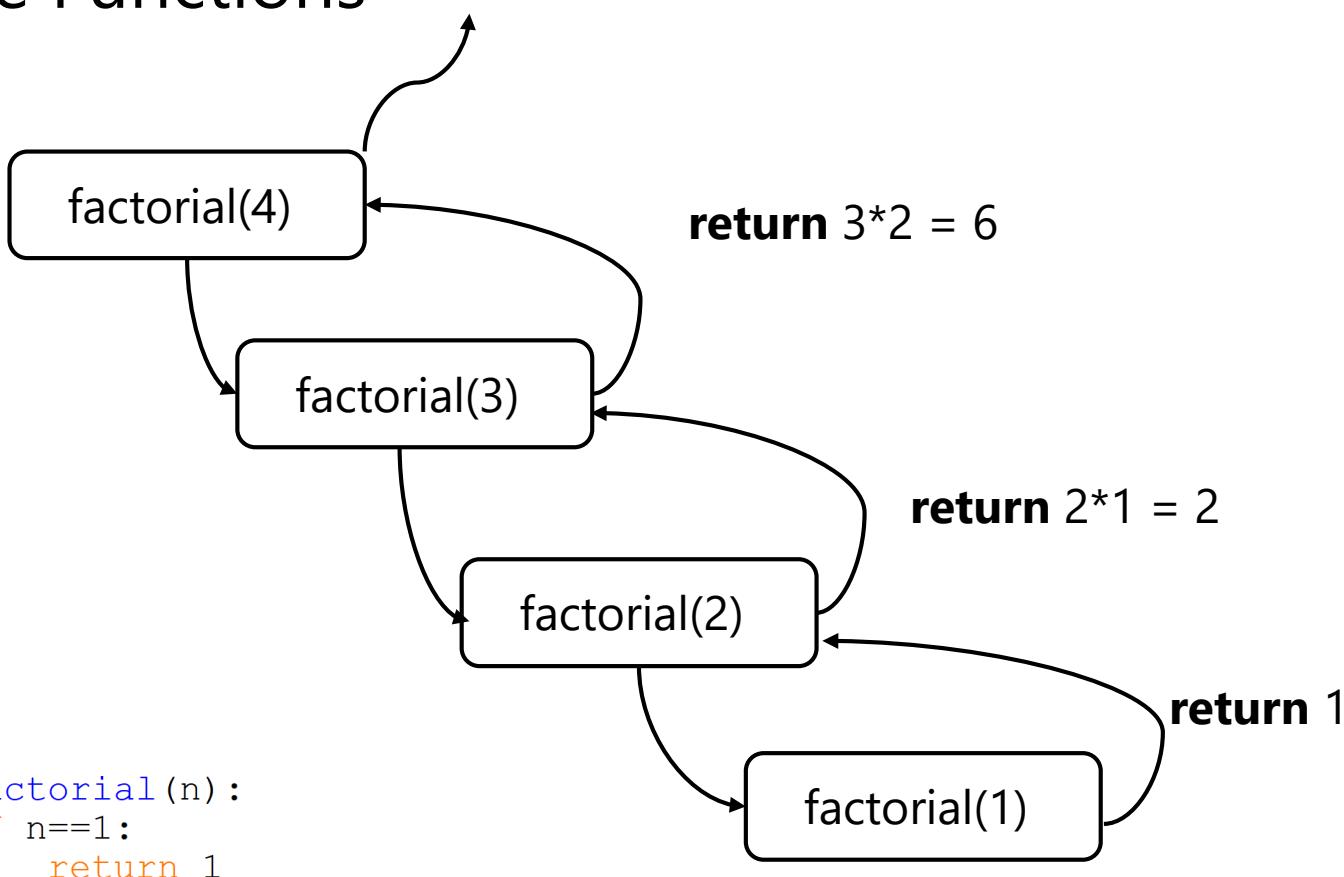
```
print(factorial(4))
```

Base Case

Calling itself
(smaller problem)

Recursive Functions

return $4 \times 6 = 24$



```
def factorial(n):  
    if n==1:  
        return 1  
    else:  
        return n*factorial(n-1)  
  
print(factorial(4))
```

```
def factorial(n):  
    if n==1:  
        return 1  
    else:  
        return n*factorial(n-1)
```

n = 4
Namespace of factorial(4)

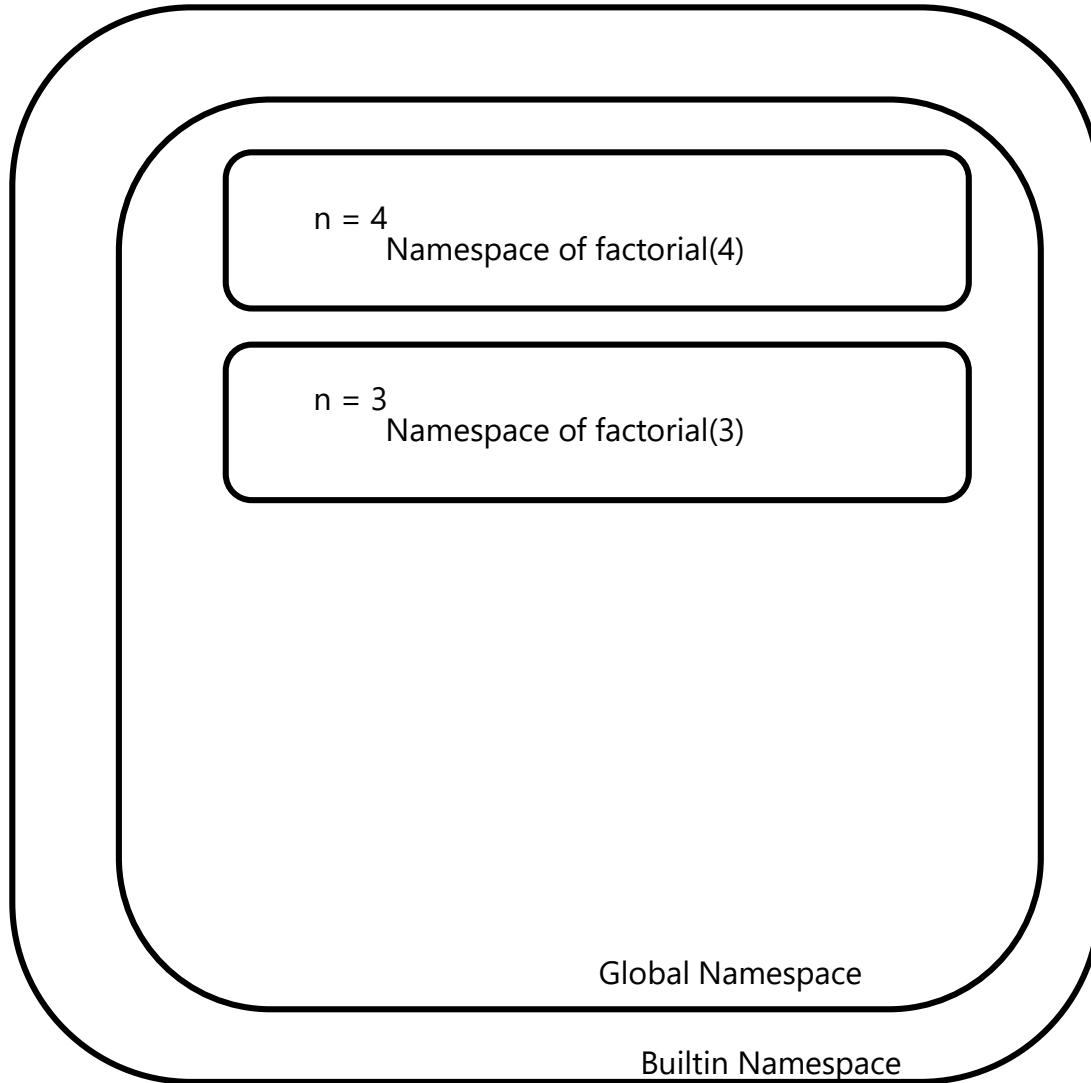
Global Namespace

Builtin Namespace

Namespaces: Recursive Function

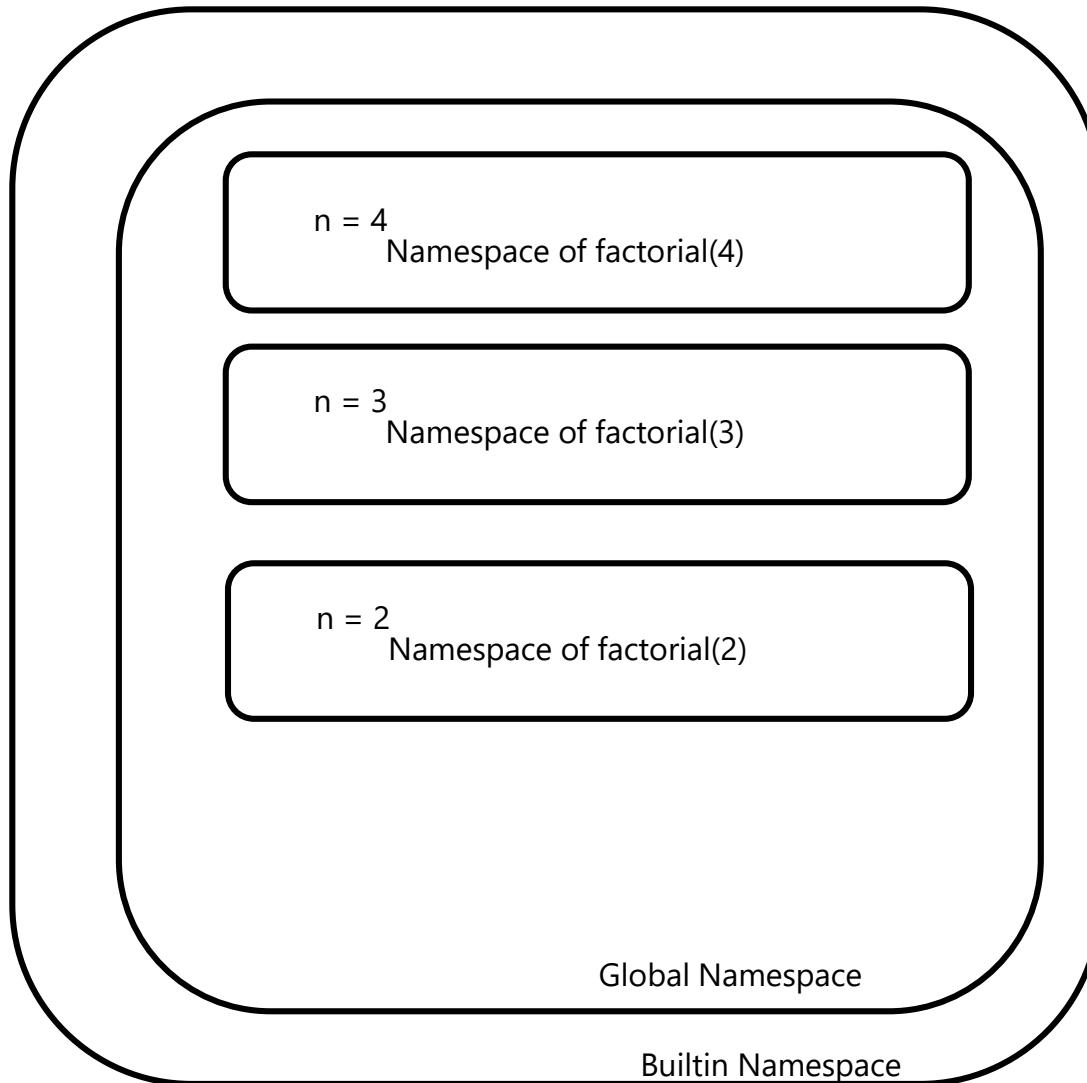
```
def factorial(n):  
    if n==1:  
        return 1  
    else:  
        return n*factorial(n-1)
```

Namespaces: Recursive Function



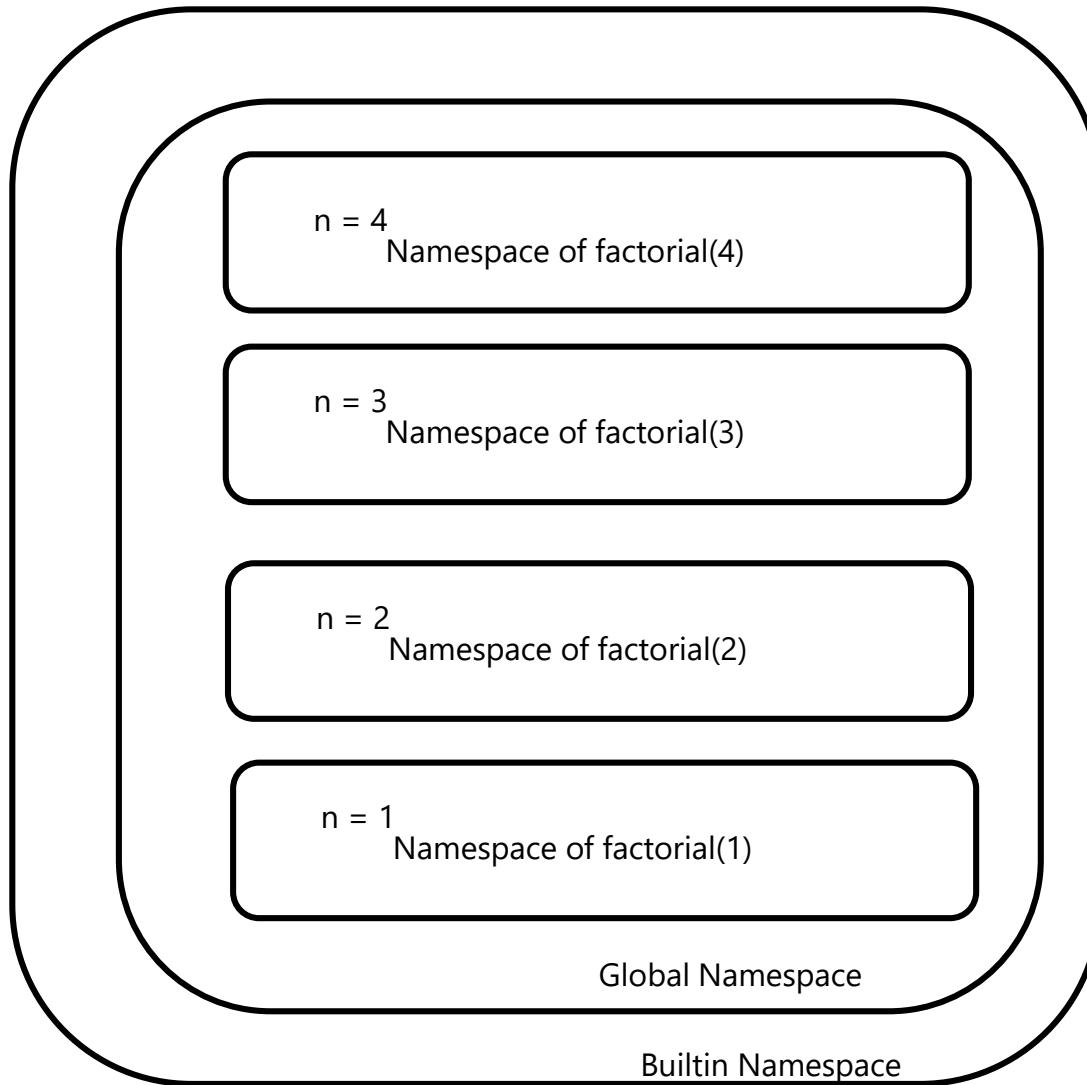
```
def factorial(n):  
    if n==1:  
        return 1  
    else:  
        return n*factorial(n-1)
```

Namespaces: Recursive Function



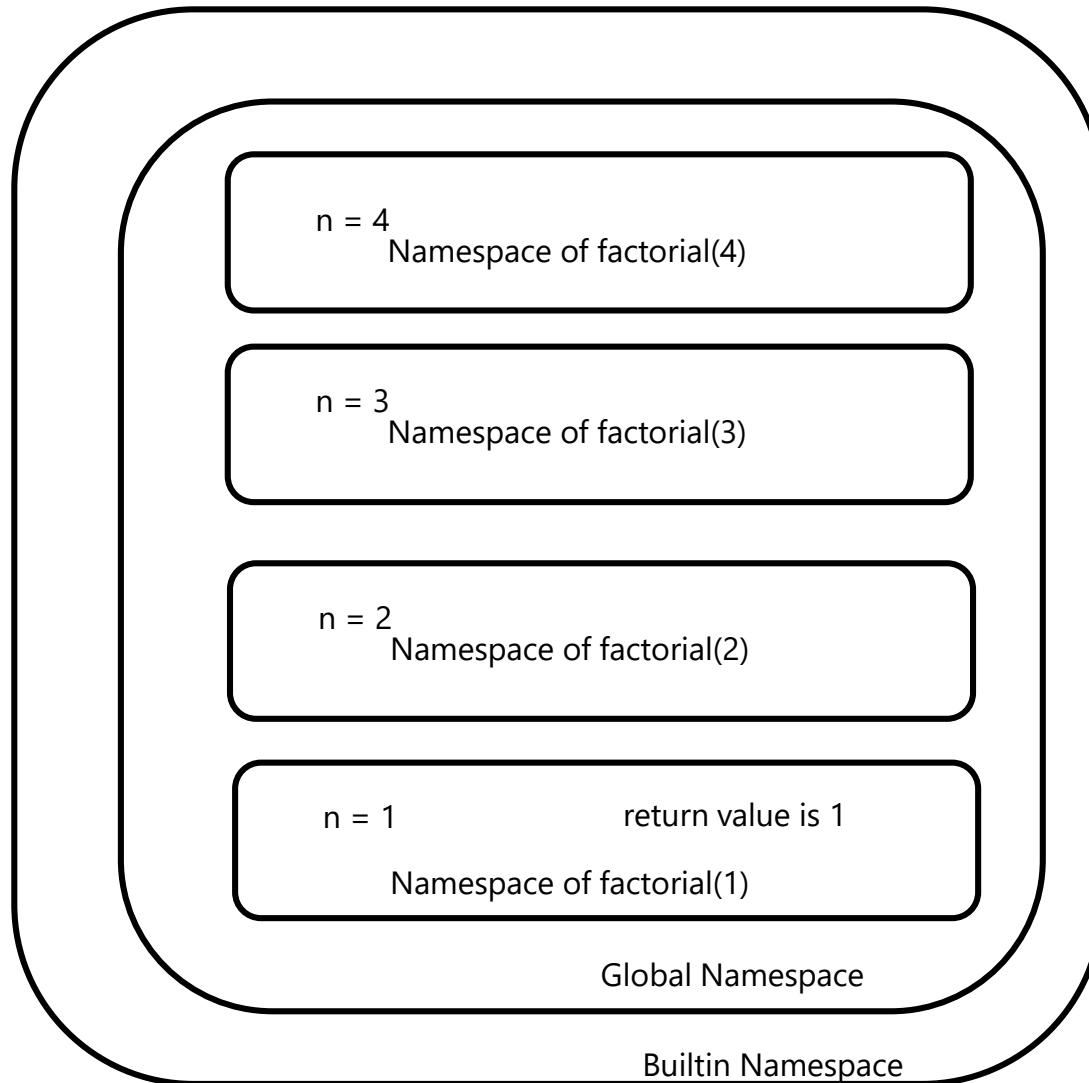
```
def factorial(n):  
    if n==1:  
        return 1  
    else:  
        return n*factorial(n-1)
```

Namespaces: Recursive Function



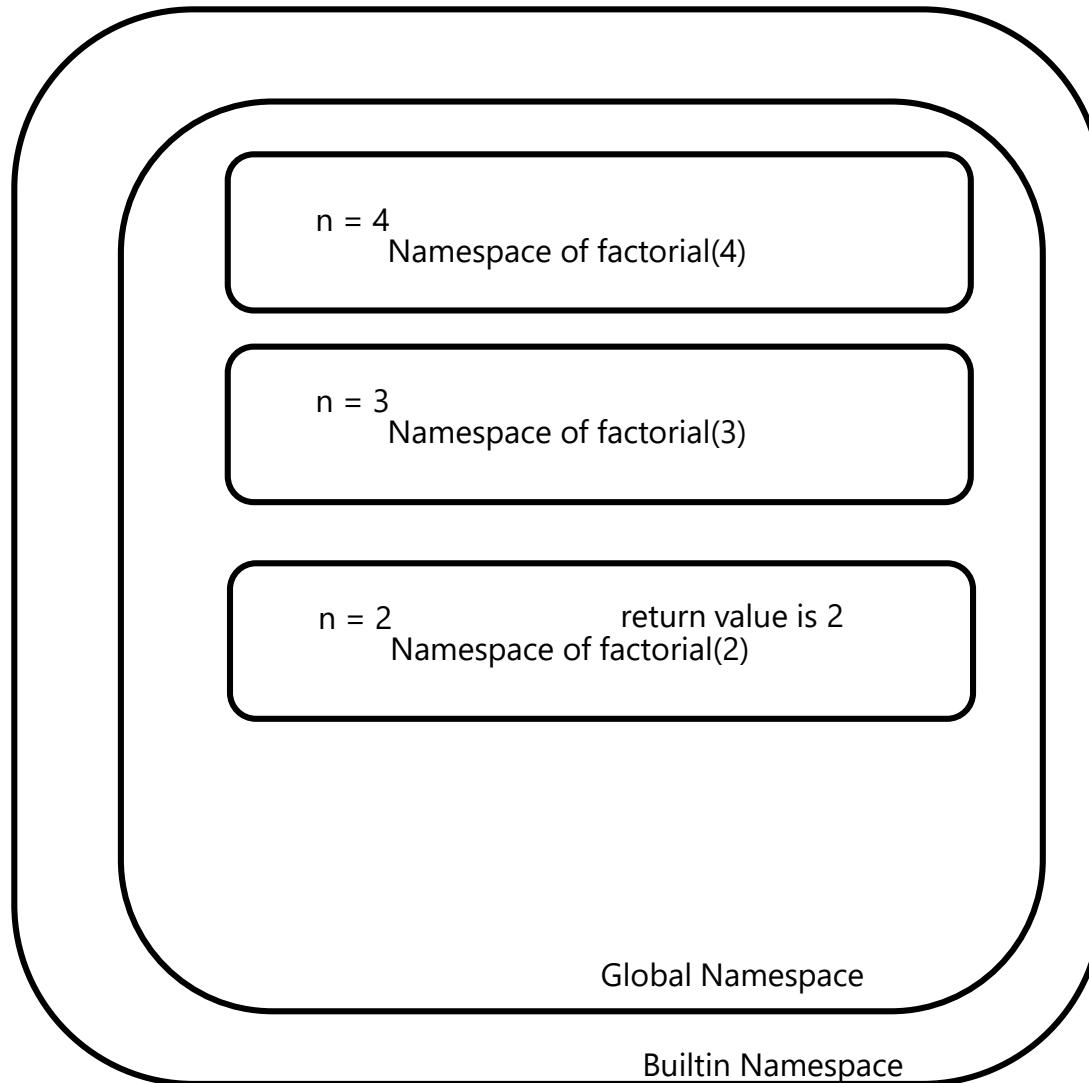
```
def factorial(n):  
    if n==1:  
        return 1  
    else:  
        return n*factorial(n-1)
```

Namespaces: Recursive Function



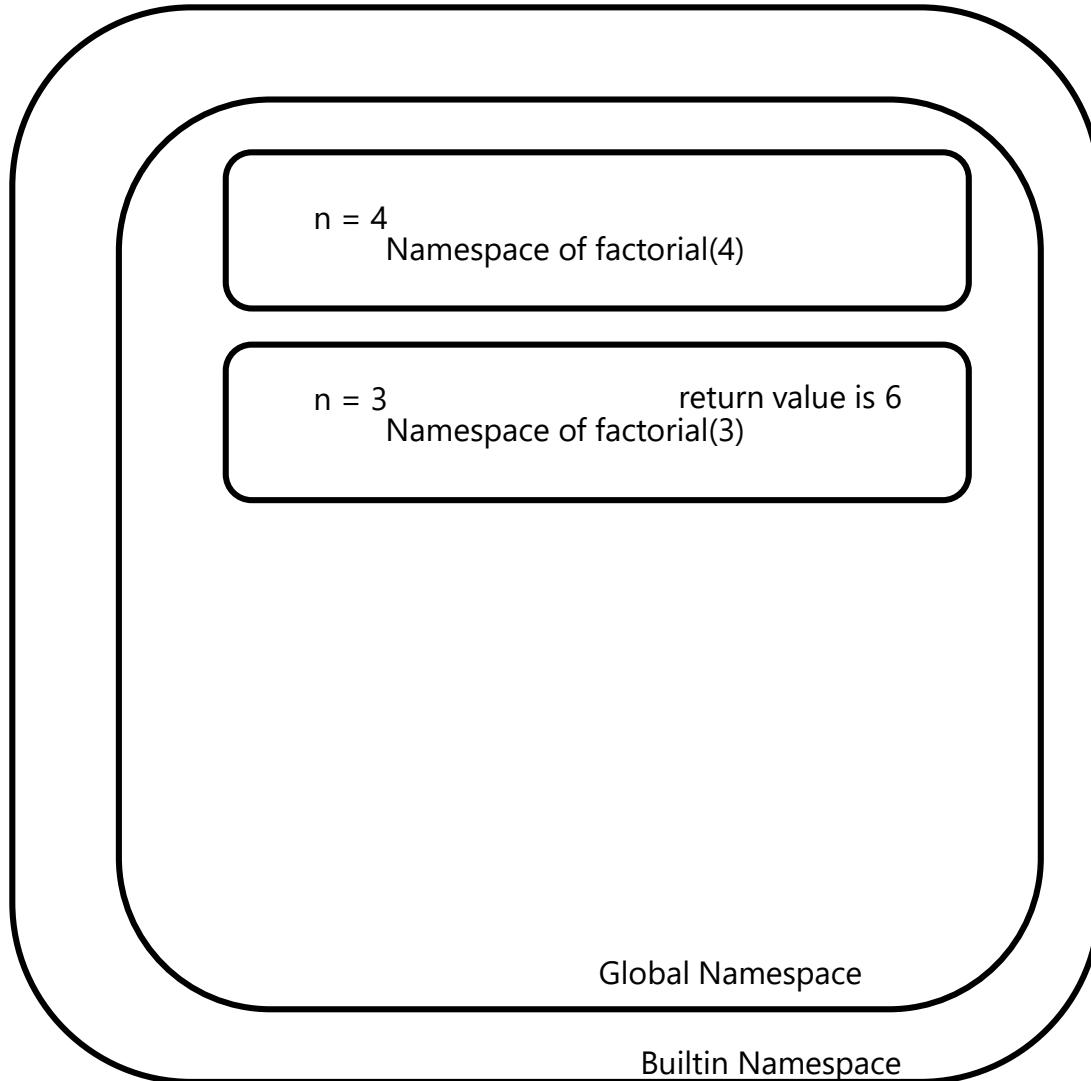
```
def factorial(n):  
    if n==1:  
        return 1  
    else:  
        return n*factorial(n-1)
```

Namespaces: Recursive Function



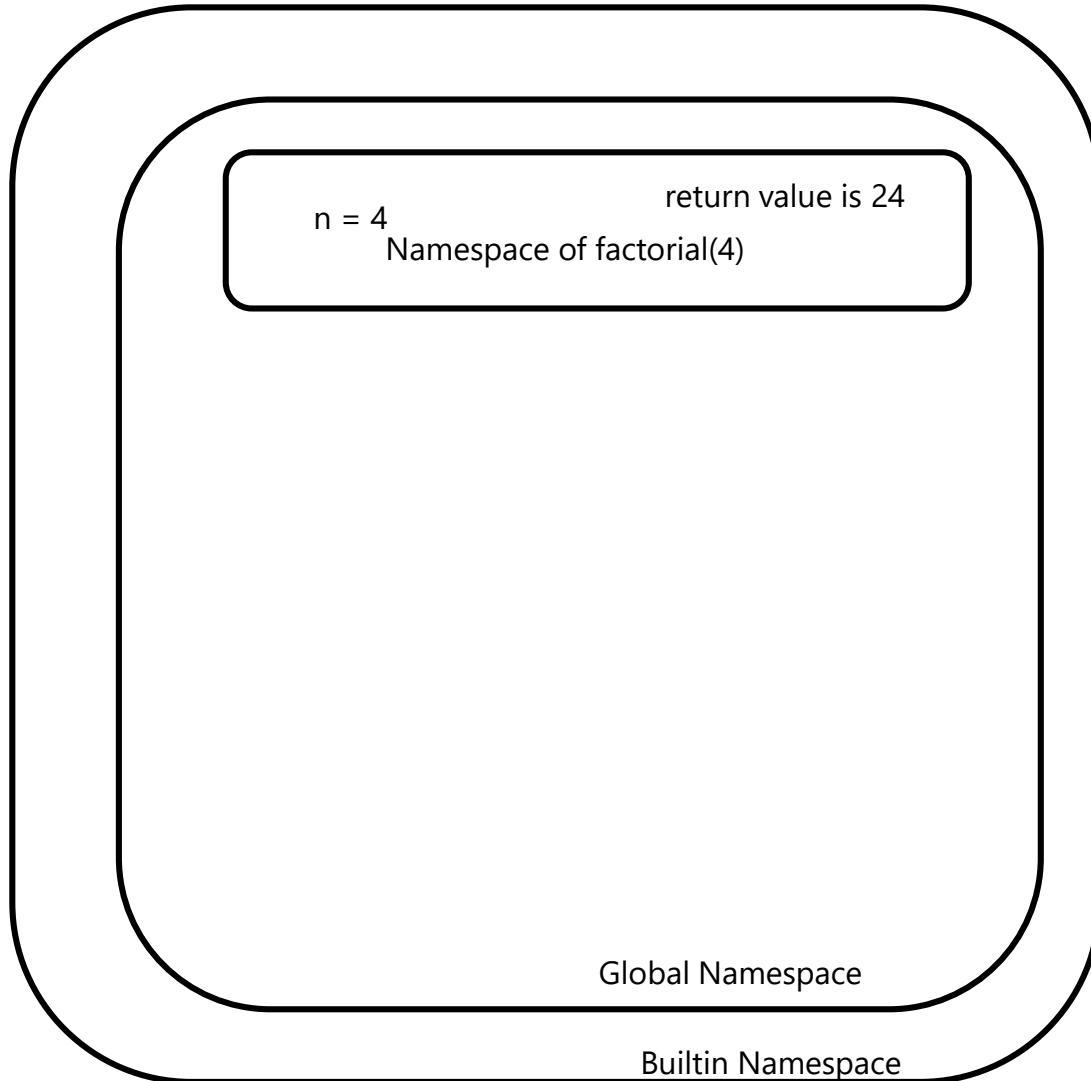
```
def factorial(n):  
    if n==1:  
        return 1  
    else:  
        return n*factorial(n-1)
```

Namespaces: Recursive Function

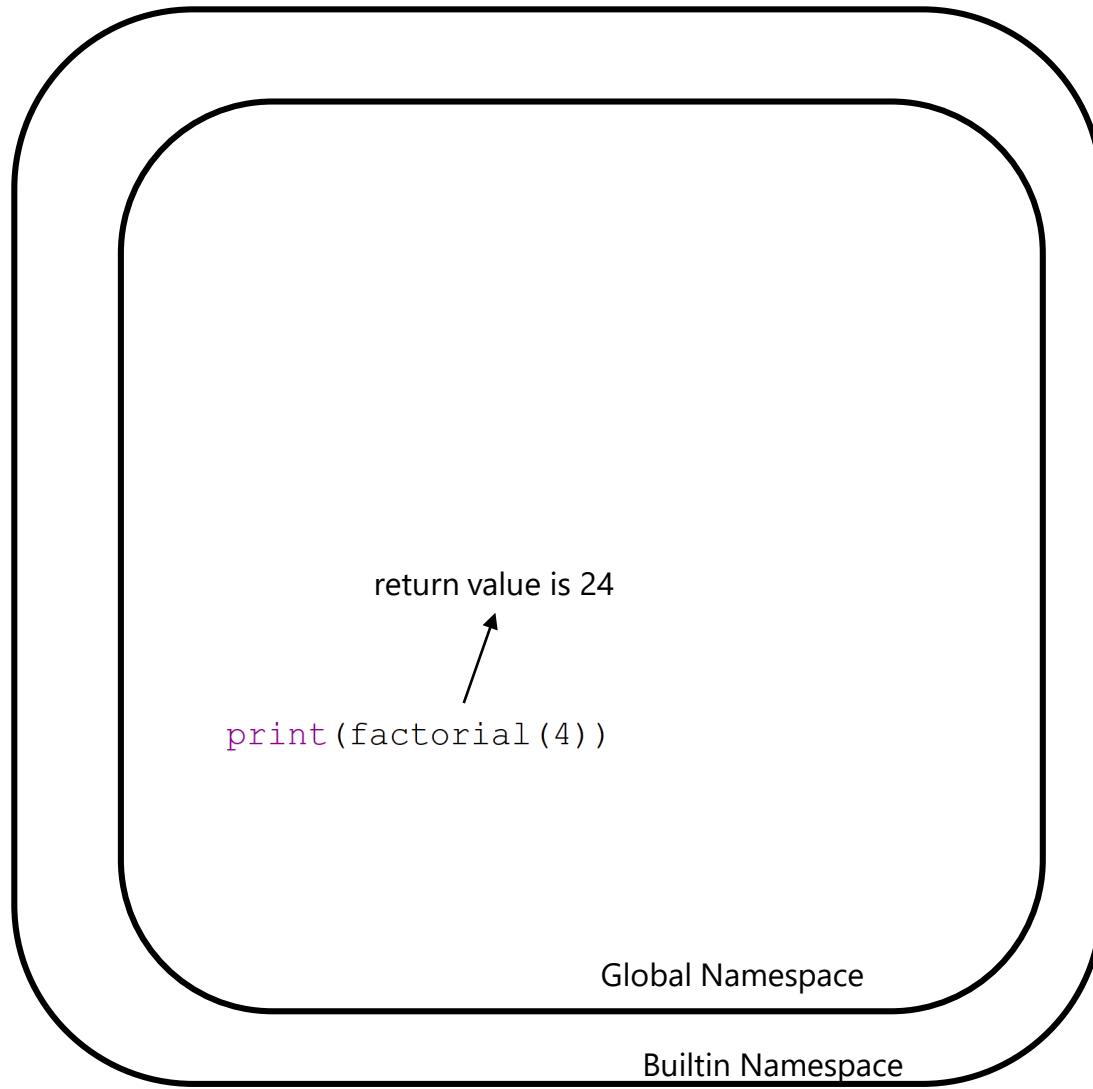


```
def factorial(n):  
    if n==1:  
        return 1  
    else:  
        return n*factorial(n-1)
```

Namespaces: Recursive Function



Namespaces: Recursive Function



Recursive Vs Iteration

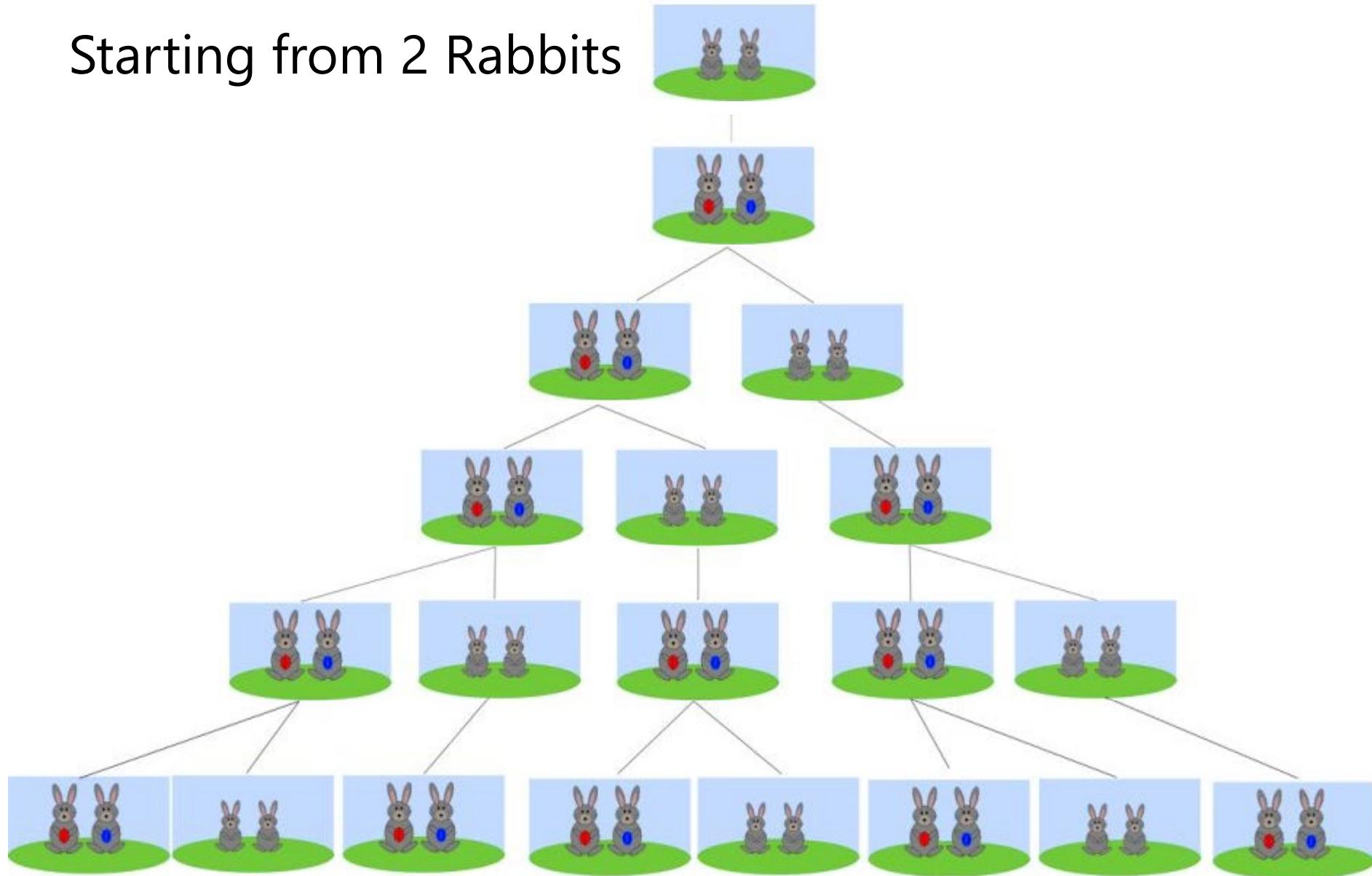
```
def factorial(n):
    if n==1:
        return 1
    else:
        return n*factorial(n-1)
```

```
def factorial_iter(n):
    fact = 1
    for elem in range(1,n+1):
        fact = fact*elem
    return fact
```

Fibonacci Number

(Recursion)

Starting from 2 Rabbits



How many ways to arrange cars?

- Let's say we have two types of vehicles, cars and buses

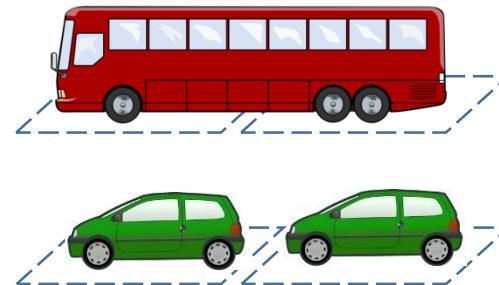


- And each car can park into one parking space, but a bus needs two consecutive ones
- If we have 1 parking space, I can only park a car



1 way

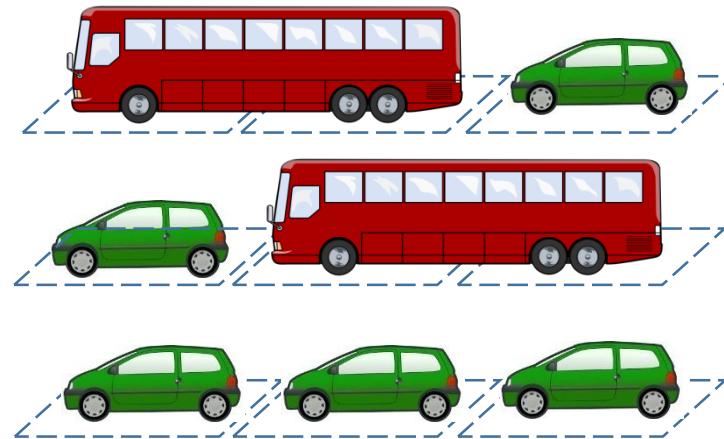
- But if there are 2 parking spaces, we can either park a bus or two cars



2 ways

How many ways to arrange cars?

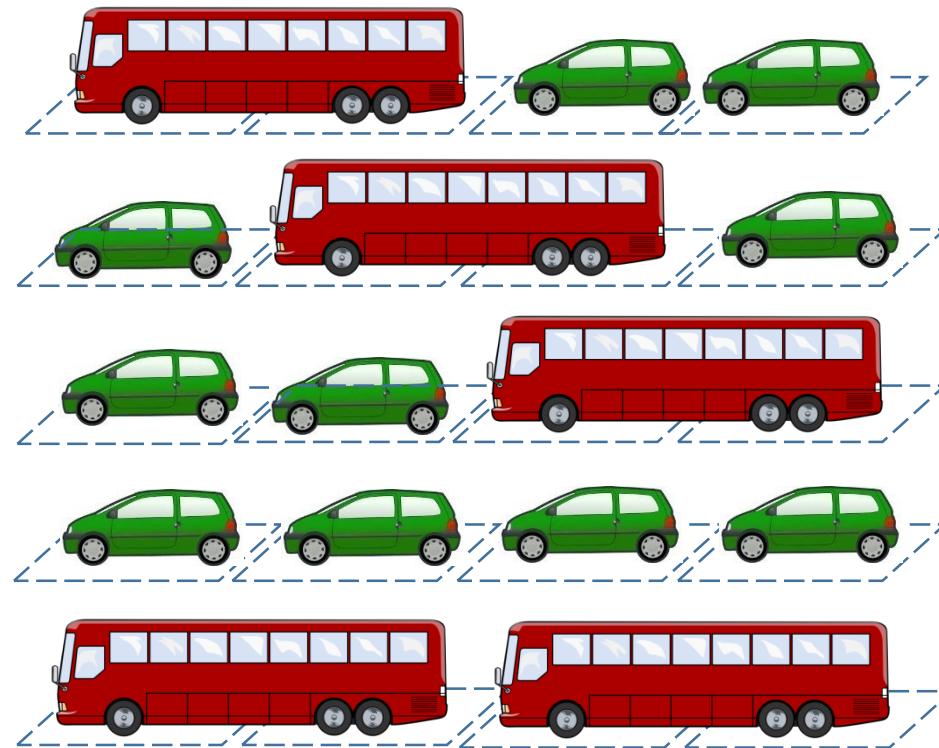
- So if we have 3 parking spaces, how many different ways can we park cars and buses?



3 ways

How many ways to arrange cars?

- So if we have 4 parking spaces, how many different ways can we park cars and buses?



5 ways

How many ways to arrange cars?

- 5 parking spaces?
- 6 parking spaces?

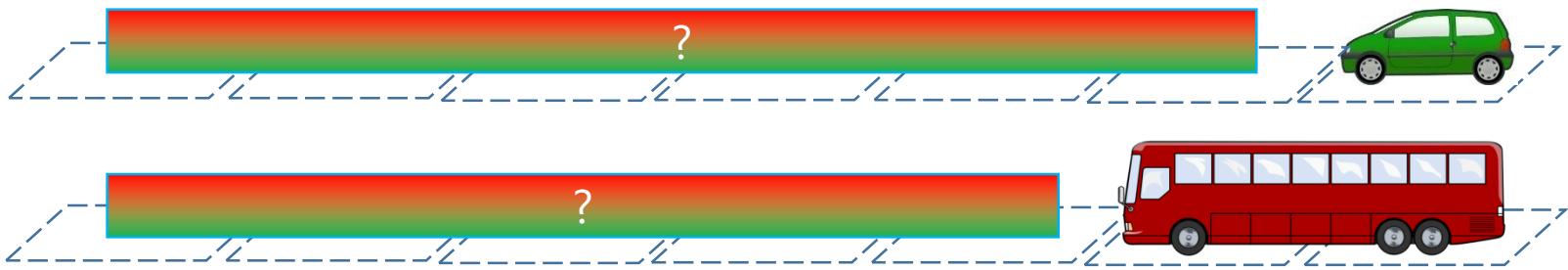


#parking spaces	#ways
0	1
1	1
2	2
3	3
4	5
5	8
6	13

- Can you figured out THE pattern?
 - 1, 1, 2, 3, 5, 8, 13, ...
 - What is the next number?

How many ways to arrange cars?

- In general, if we have n parking spaces, how many ways can we park the vehicles?
- You can think backward, the last parking space can be either a car or a bus



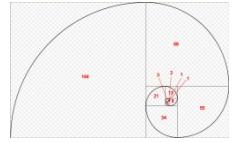
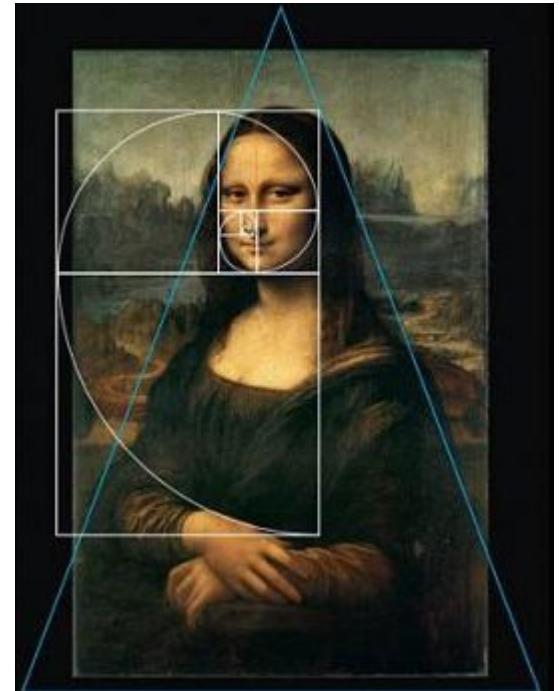
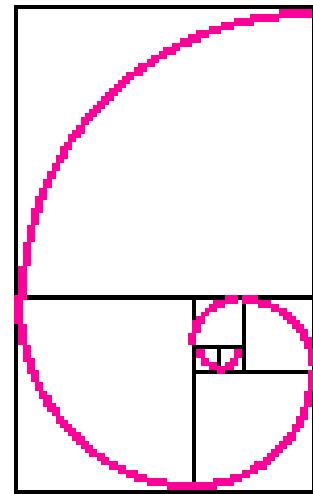
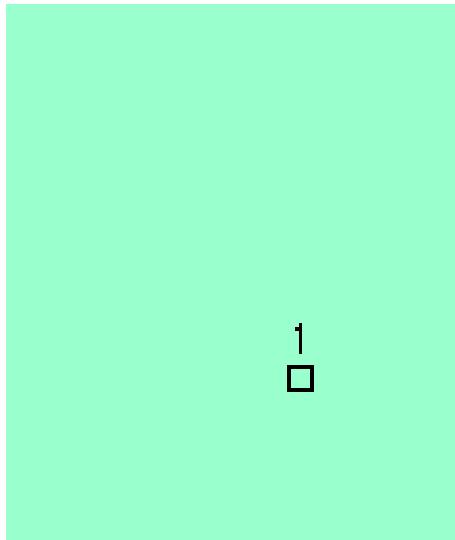
- If it's a car, there are $n - 1$ spaces left, you can have the number of way for $n - 1$ spaces
 - Otherwise, it's the number of way for $n - 2$ spaces
- So

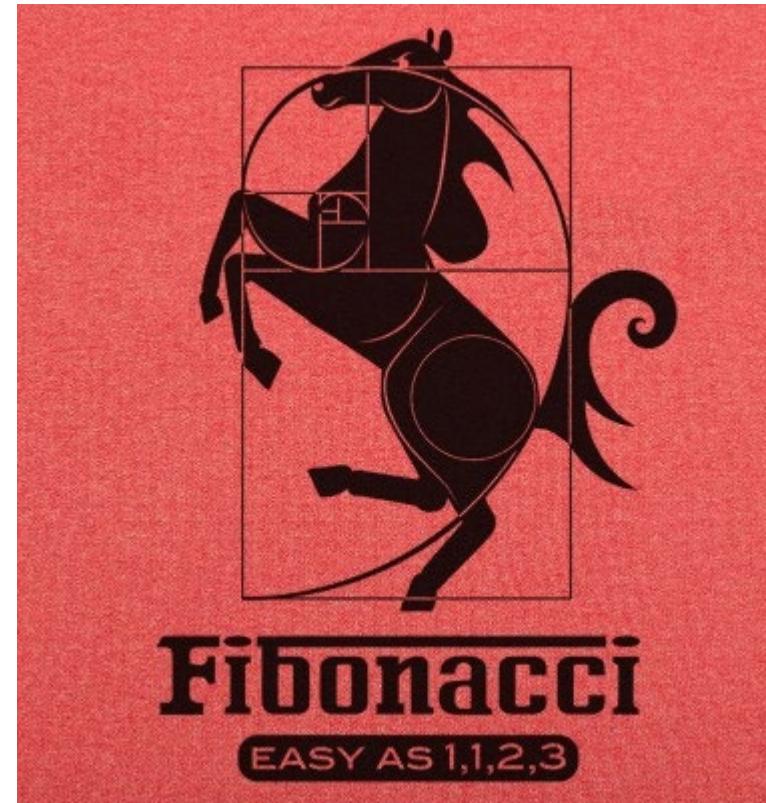
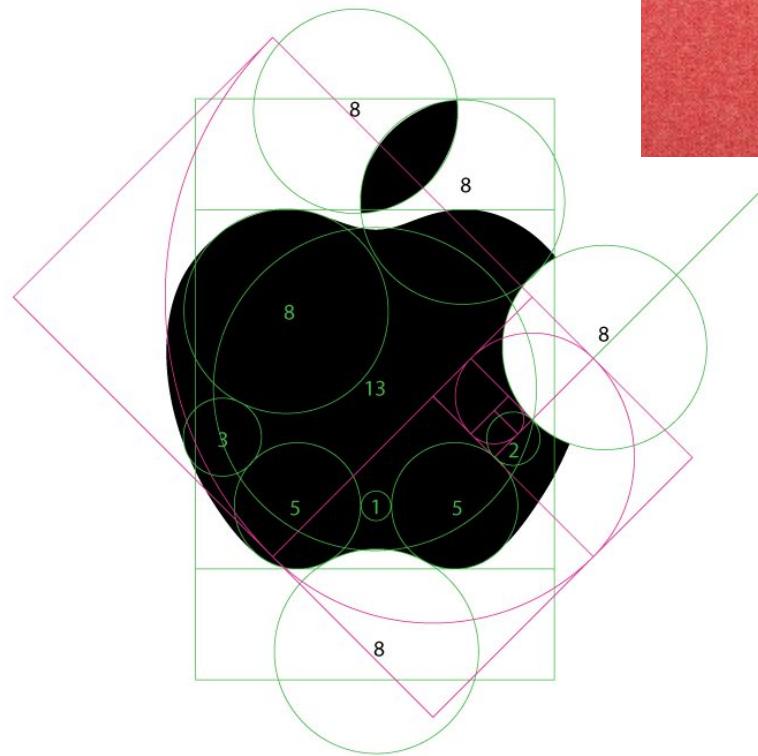
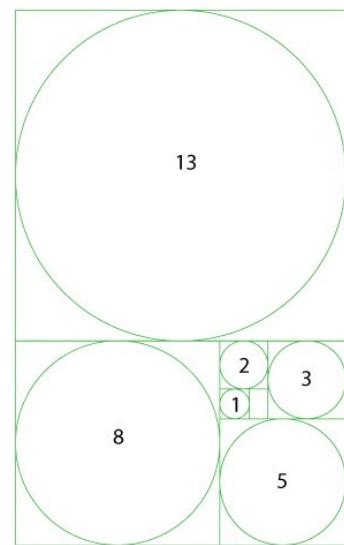
$$f(n) = f(n - 1) + f(n - 2) \quad \text{for } f(0) = f(1) = 1$$

Fibonacci Numbers



- Fibonacci numbers are found in nature (seashells, sunflowers, etc)
- <http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibnat.html>





```
def fibonacci(n):
    if n == 1 or n == 0:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

```
>>> fibonacci(10)
```

```
89
```

```
>>> fibonacci(20)
```

```
10946
```

```
>>> fibonacci(994)
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#16>", line 1, in <module>
```

```
    fibonacci(994)
```

```
  File "<pyshell#5>", line 5, in fibonacci
```

```
    return fibonacci(n-1) + fibonacci(n-2)
```

```
  File "<pyshell#5>", line 5, in fibonacci
```

```
    return fibonacci(n-1) + fibonacci(n-2)
```

```
  File "<pyshell#5>", line 5, in fibonacci
```

```
    return fibonacci(n-1) + fibonacci(n-2)
```

```
[Previous line repeated 989 more times]
```

```
  File "<pyshell#5>", line 2, in fibonacci
```

```
    if n == 1 or n == 0:
```

```
RecursionError: maximum recursion depth exceeded in comparison
```

```
>>> fibonacci(50)
```



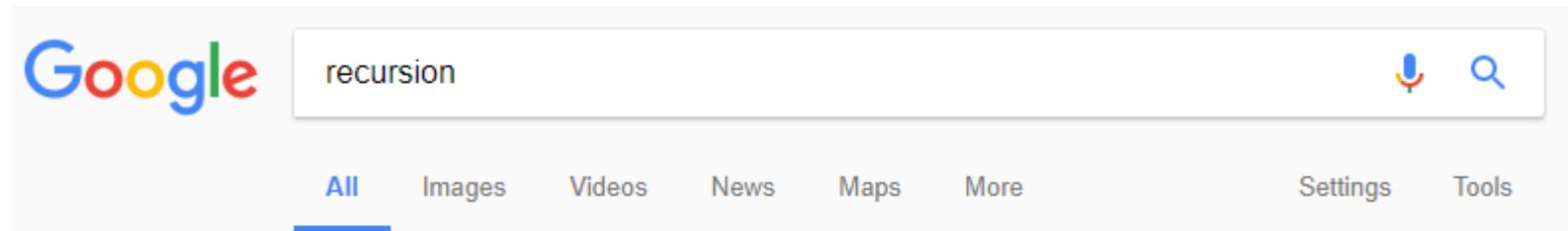
Challenge

- Write a fibonacci function that can compute $f(n)$ for $n > 1000$

```
>>> fibonacci(1000)
70330367711422815821835254877183549770181269836358732742604905087154537118196933
57974224949456261173348775044924176599108818636326545022364710601205337412127386
7339111198139373125598767690091902245245323403501
>>> fibonacci(2000)
68357022595758066470453965491705801070554080293655245654075533677980824544080540
14954534318953113802726603726769523447478238192192714526677939943338306101405105
41481970566409090181363729645376709552810486826470491443352935557914873104468563
41354877358979546298425169471014942535758696998934009765395457402148198191519520
85089538422954565146720383752121972115725761141759114990448978941370030912401573
418221496592822626
```



Google about Recursion



- Did you mean: *recursion*
 - Do a barrel roll
 - Askew
 - Anagram
 - Google in 1998
 - Zerg rush
- More in [Google Easter Eggs](#)



IT5001 Software Development Fundamentals

5. Recursion Vs Iterations and Nested Functions

Sirigina Rajendra Prasad

Recursion vs Iteration

Reversing a String

- How about reversing a string? Of course, we can just use string slicing

```
>>> s = 'abcde12345'  
>>> s[::-1]  
'54321edcba'  
>>>
```

- How about we write a function for it?

```
>>> reverseStringI(s)  
'54321edcba'  
>>>
```

Reverse String (Iterative Version 1)

```
def reverseStringI(s):
    output = ''
    l = len(s)
    for i in range(l):
        output += s[l-i-1]
    return output
```

```
>>> reverseStringI('abcde')
'edcba'
```

i
0
1
2
3
4

Reverse String (Iterative Version 1)

```
def reverseStringI(s):
    output = ''
    l = len(s)
    for i in range(l):
        output += s[l-i-1]
    return output
```

```
>>> reverseStringI('abcde')
'edcba'
```

i	l-i-1
0	4
1	3
2	2
3	1
4	0

Reverse String (Iterative Version 1)

```
def reverseStringI(s):
    output = ''
    l = len(s)
    for i in range(l):
        output += s[l-i-1]
    return output
```

```
>>> reverseStringI('abcde')
'edcba'
```

i	l-i-1	s[l-i-1]
0	4	e
1	3	d
2	2	c
3	1	b
4	0	a

Reverse String (Iterative Version 1)

```
def reverseStringI(s):
    output = ''
    l = len(s)
    for i in range(l):
        output += s[l-i-1]
    return output

>>> reverseStringI('abcde')
'edcba'
```

i	l-i-1	s[l-i-1]	output
0	4	e	e
1	3	d	ed
2	2	c	edc
3	1	b	edcb
4	0	a	edcba

Reverse String (Iterative Version 2)

```
def reverseStringI(s):
    output = ''
    for c in s:
        output = c + output
    return output

>>> reverseStringI('abcde')
'edcba'
```

c	output
a	a
b	ba
c	cba
d	dcba
e	edcba

Reversing String (Recursive Version)

```
def reverseStringR(s):
    if not s:
        return ''
    return reverseStringR(s[1:]) + s[0]
```

- `reverseStringR('abcde')`
- `reverseStringR('bcde')+'a'`
- `reverseStringR('cde')+'b'+'a'`
- `reverseStringR('de')+'c'+'b'+'a'`
- `reverseStringR('e')+'d'+'c'+'b'+'a'`
- `reverseStringR('')+'e'+'d'+'c'+'b'+'a'`
- `'+'+'e'+'d'+'c'+'b'+'a'`
- `'edcba'`

Taylor Series

$$\begin{aligned}\sin x &= \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots &\text{for all } x \\ \cos x &= \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots &\text{for all } x \\ \tan x &= \sum_{n=1}^{\infty} \frac{B_{2n}(-4)^n (1-4^n)}{(2n)!} x^{2n-1} &= x + \frac{x^3}{3} + \frac{2x^5}{15} + \dots &\text{for } |x| < \frac{\pi}{2} \\ \sec x &= \sum_{n=0}^{\infty} \frac{(-1)^n E_{2n}}{(2n)!} x^{2n} &= 1 + \frac{x^2}{2} + \frac{5x^4}{24} + \dots &\text{for } |x| < \frac{\pi}{2} \\ \arcsin x &= \sum_{n=0}^{\infty} \frac{(2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1} &= x + \frac{x^3}{6} + \frac{3x^5}{40} + \dots &\text{for } |x| \leq 1 \\ \arccos x &= \frac{\pi}{2} - \arcsin x & & \\ &= \frac{\pi}{2} - \sum_{n=0}^{\infty} \frac{(2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1} &= \frac{\pi}{2} - x - \frac{x^3}{6} - \frac{3x^5}{40} - \dots &\text{for } |x| \leq 1 \\ \arctan x &= \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} x^{2n+1} &= x - \frac{x^3}{3} + \frac{x^5}{5} - \dots &\text{for } |x| \leq 1, x \neq \pm i\end{aligned}$$

Taylor Series

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

$$= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \quad \text{for all } x$$

$n = 0 \quad n = 1 \quad n = 2$

- We do not need the infinite precision
- We may just sum up to k terms

$$\sin x = \sum_{n=0}^{\cancel{k}} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

$$= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \quad \text{for all } x$$

Computing sine by Iteration

$$\sin x = \sum_{n=0}^{\cancel{k}} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \quad \text{for all } x$$

- Using iteration

```
def sinI(x, k):  
    result = 0  
    for n in range(0, k):  
        result += ((-1)**n / fact(2*n+1)) * x**(2*n+1)  
    return result
```

```
>>> print(sinI(PI/6, 10))  
0.5000000000592083  
>>> from math import sin  
>>> sin(PI/6) ←  
0.5000000000592083
```

Python Library version of “sin()”

Computing sine by Recursion

Sum up to $n = 2$

$$\sin x = \sum_{n=0}^{\cancel{k}} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

$$= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$$

for all x

Sum up to $n = 1$

- In general, if we want to sum up to the k terms

Sum up to $n = k$

$$\sin x = \sum_{n=0}^{\cancel{k}} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

$$= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$$

k^{th}

for all x

Sum up to $n = k - 1$

$n = k$

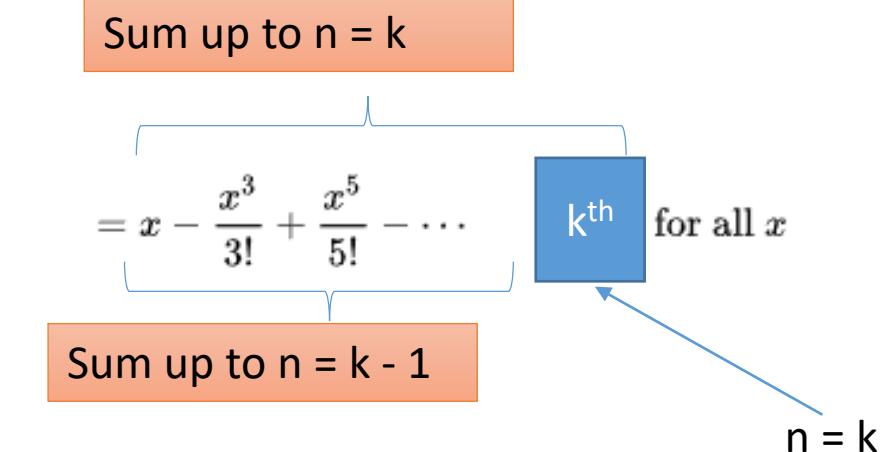
Computing sine by Recursion

- Assuming that if the function $\sinR(x, k)$ sums until $n = k$, then

$$\sinR(x, k) = \sinR(x, k-1) + \text{the } k^{\text{th}} \text{ term}$$

- In general, if we want to sum up to the k terms

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$



Computing sine by Recursion

- Assuming that if the function $\sinR(x, k)$ sums until $n = k$, then

$$\sinR(x, k) = \sinR(x, k-1) + \text{the } k\text{th term}$$

```
def sinR(x, k):  
    if k < 0:  
        return 0  
    return sinR(x, k-1) + ((-1)**k / fact(2*k+1)) * x**(2*k+1)
```

```
>>> sinR(PI/6, 6)  
0.500000000592083  
>>> from math import sin  
>>> sin(PI/6)  
0.500000000592083
```

More Taylor Series

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \quad \text{for all } x$$

$$\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots \quad \text{for all } x$$

$$\tan x = \sum_{n=1}^{\infty} \frac{B_{2n} (-4)^n (1 - 4^n)}{(2n)!} x^{2n-1} = x + \frac{x^3}{3} + \frac{2x^5}{15} + \dots \quad \text{for } |x| < \frac{\pi}{2}$$

$$\sec x = \sum_{n=0}^{\infty} \frac{(-1)^n E_{2n}}{(2n)!} x^{2n} = 1 + \frac{x^2}{2} + \frac{5x^4}{24} + \dots \quad \text{for } |x| < \frac{\pi}{2}$$

$$\arcsin x = \sum_{n=0}^{\infty} \frac{(2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1} = x + \frac{x^3}{6} + \frac{3x^5}{40} + \dots \quad \text{for } |x| \leq 1$$

$$\begin{aligned} \arccos x &= \frac{\pi}{2} - \arcsin x \\ &= \frac{\pi}{2} - \sum_{n=0}^{\infty} \frac{(2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1} = \frac{\pi}{2} - x - \frac{x^3}{6} - \frac{3x^5}{40} - \dots \quad \text{for } |x| \leq 1 \end{aligned}$$

$$\arctan x = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} x^{2n+1} = x - \frac{x^3}{3} + \frac{x^5}{5} - \dots \quad \text{for } |x| \leq 1, x \neq \pm i$$

Recursion Common Patterns

```
def reverseStringR(s):
    if not s:
        return ''
    return reverseStringR(s[1:]) + s[0]

def sinR(x, k):
    if k < 0:
        return 0
    return sinR(x, k-1) + ((-1)**k / fact(2*k+1)) * x**(2*k+1)
```

→ Base cases

Recursion step to reduce the problem one-by-one ←

Iteration Common Patterns

```
def reverseStringI(s):
    output = ''
    l = len(s)
    for i in range(l):
        output += s[l-i-1]
    return output

def sinI(x, k):
    result = 0
    for n in range(0, k):
        result += ((-1)**n / fact(2*n+1)) * x**(2*n+1)
    return result
```

Accumulate element one-by-one

- Initial the final answer to “nothing” at the beginning.
- Accumulate and return the final answer

Iteration/Recursion Conversion

```
def sinR(x, k):
    if k < 0:
        return 0
    return sinR(x, k-1) + ((-1)**k / fact(2*k+1)) * x**(2*k+1)
```

Base case

The answer for previous $k - 1$ terms

The k th term

```
def sinI(x, k):
    result = 0
    for n in range(0, k):
        result += ((-1)**n / fact(2*n+1)) * x**(2*n+1)
    return result
```

Iteration/Recursion Conversion

```
def reverseStringR(s):  
    if not s:  
        return ''  
    return reverseStringR(s[1:]) + s[0]
```

Base case

The answer for previous $k - 1$ terms

The k th term

```
def reverseStringI(s):  
    output = ''  
    l = len(s)  
    for i in range(l):  
        output += s[l-i-1]  
    return output
```

“Homework”

$$\arcsin x = \sum_{n=0}^{\infty} \frac{(2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1} = x + \frac{x^3}{6} + \frac{3x^5}{40} + \dots \quad \text{for } |x| \leq 1$$

- The answer for all k-1 terms?
- Base case?
- Kth term?

Another Example

Recursion vs Iteration

- SumDigits
 - Given a positive number n , the sum of all digits is obtained by adding the digit one-by-one
 - For example, the sum of 52634 = $5 + 2 + 6 + 3 + 4 = 20$
 - Write a function `sum(n)` to compute the sum of all the digits in n
- Factorial
 - Factorial is defined (recursively) as $n! = n * (n - 1)!$ such that $0! = 1$
 - Write a function `fact(n)` to compute the value of $n!$
- Can you do it in both recursion and iteration?

SumDigits

Iteration

```
def sum(n):  
    res = 0  
    while n > 0:  
        res = res + n%10  
        n = n//10  
  
    return res
```

base/initial value

computation

continuation/next value

Recursion

```
def sum(n):  
    if n == 0:  
        return 0  
    else:  
        return n%10 + sum(n//10)
```

stop/base case (*they are related, how?*)

temporary result variables
not needed in recursion (*why?*)

Factorial

Iteration

```
def fact(n):  
    res = 1  
    while n > 0:  
        res = res * n  
        n = n-1  
  
    return res
```

base/initial value

computation

continuation/next value

Recursion

```
def fact(n):  
    if n == 0:  
        return 1  
    else:  
        return n * fact(n-1)
```

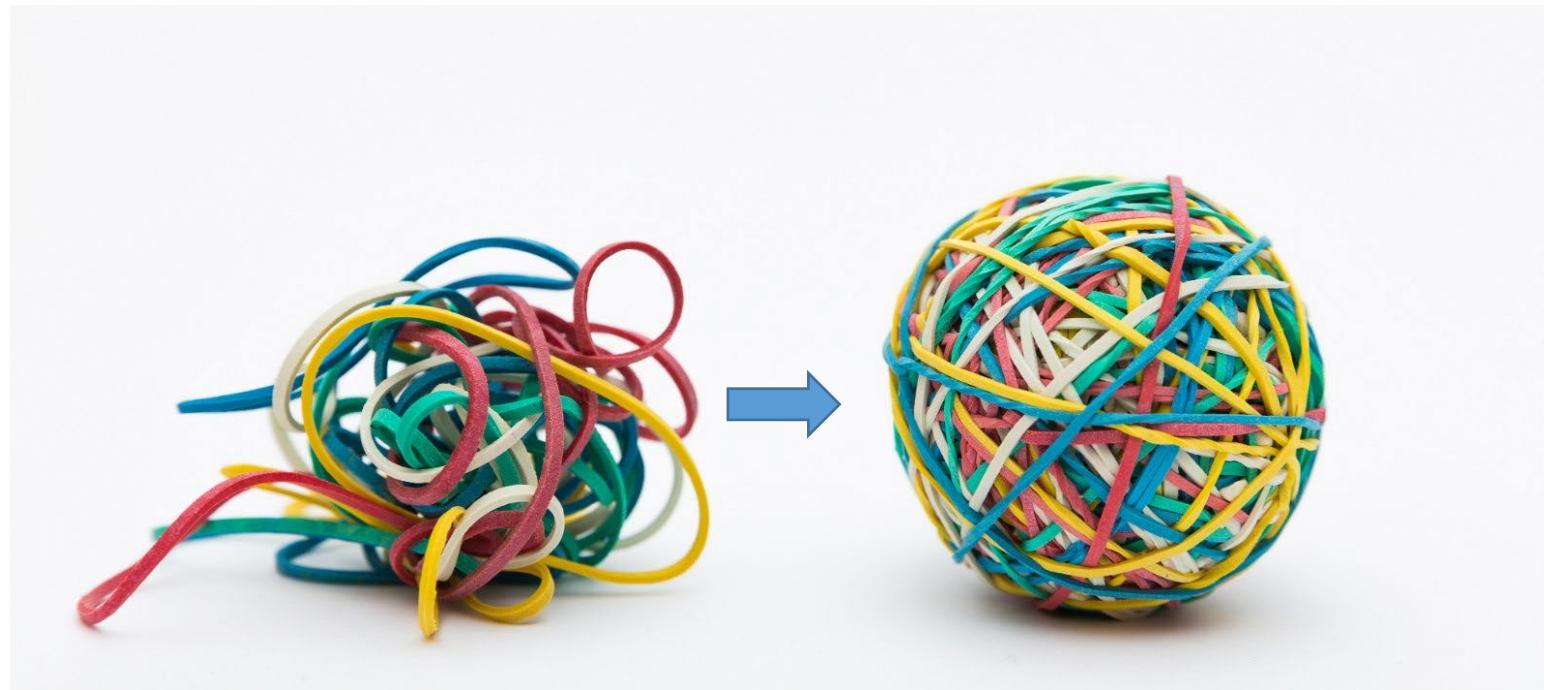
stop/base case (*they are related, how?*)

temporary result variables
not needed in recursion (*why?*)

“Homework”

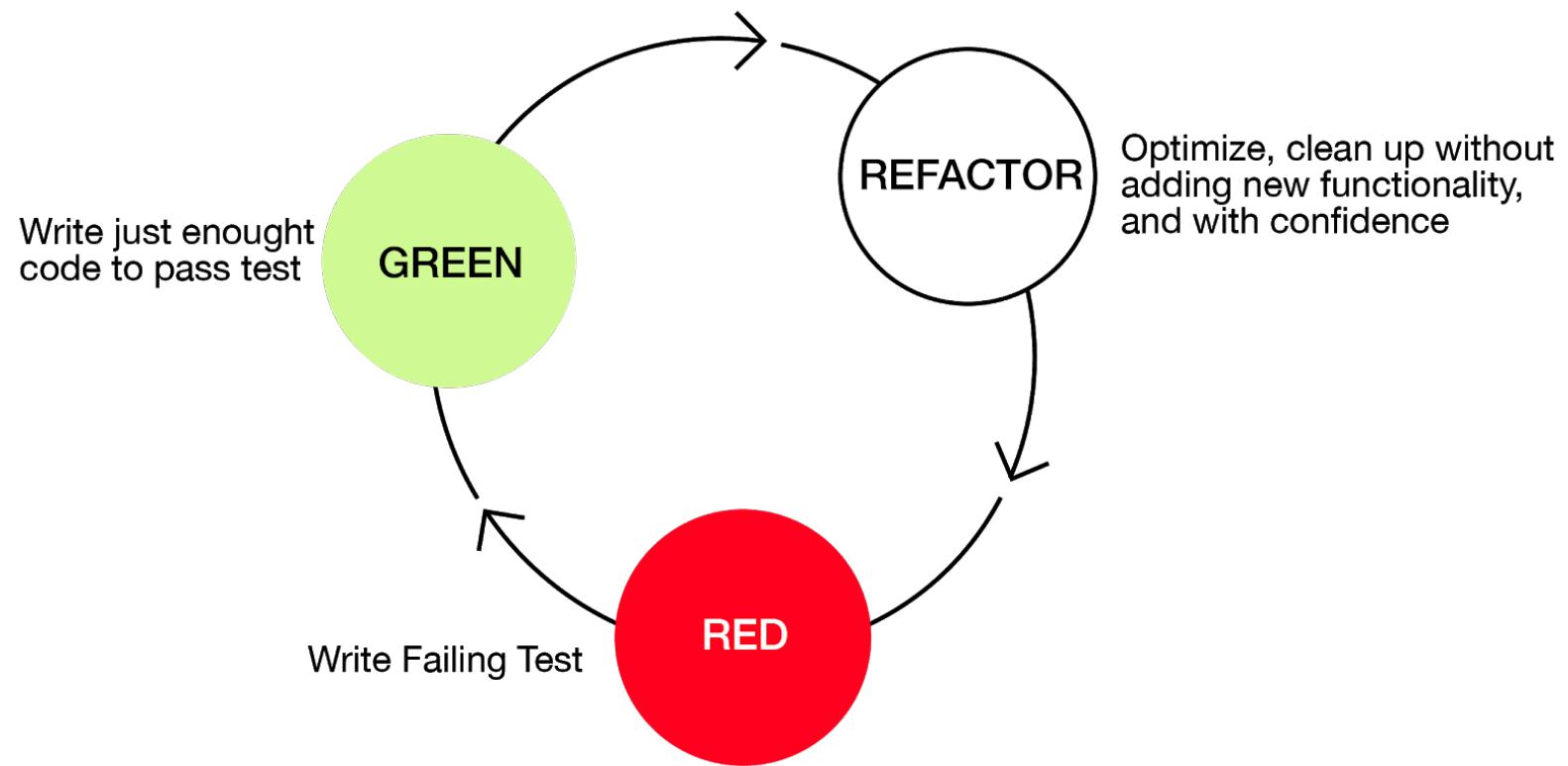
- How to re-write your code with both iterative/recursion version mentioned in this course before?
 - burgerPrice ()
 - checkAllAlpha ()
 - Etc.
- The answer for all k-1 terms?
- Base case?
- Kth term?

Code Refactoring



Code Refactoring

- **Refactoring** is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.



Nested Functions

Nested Functions

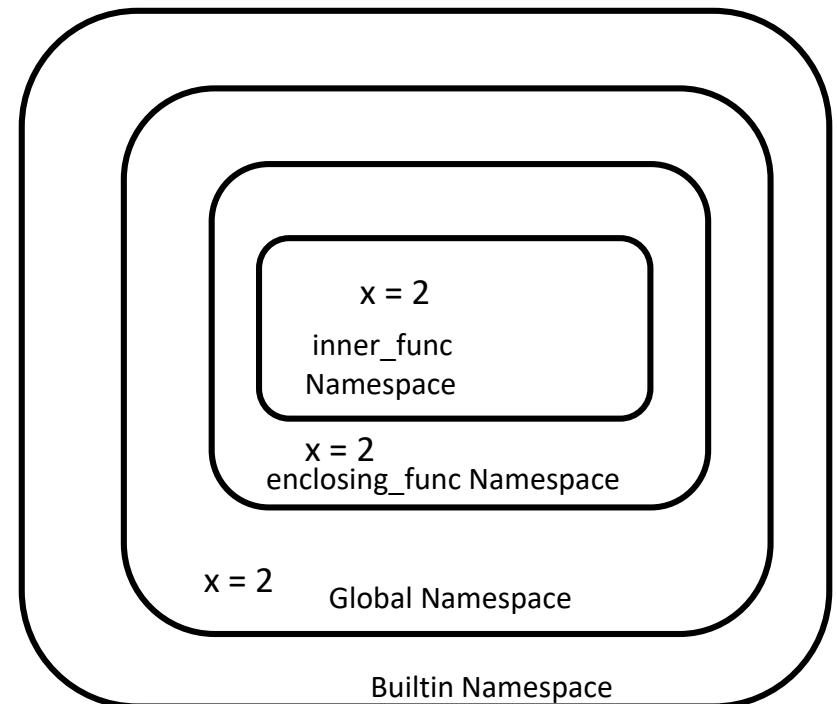
- Functions defined inside other functions

```
x = 2
def enclosing_func(x):
    def inner_func(x):
        return x**2
    output = inner_func(x)
    return output

print(enclosing_func(x))
```

Output:

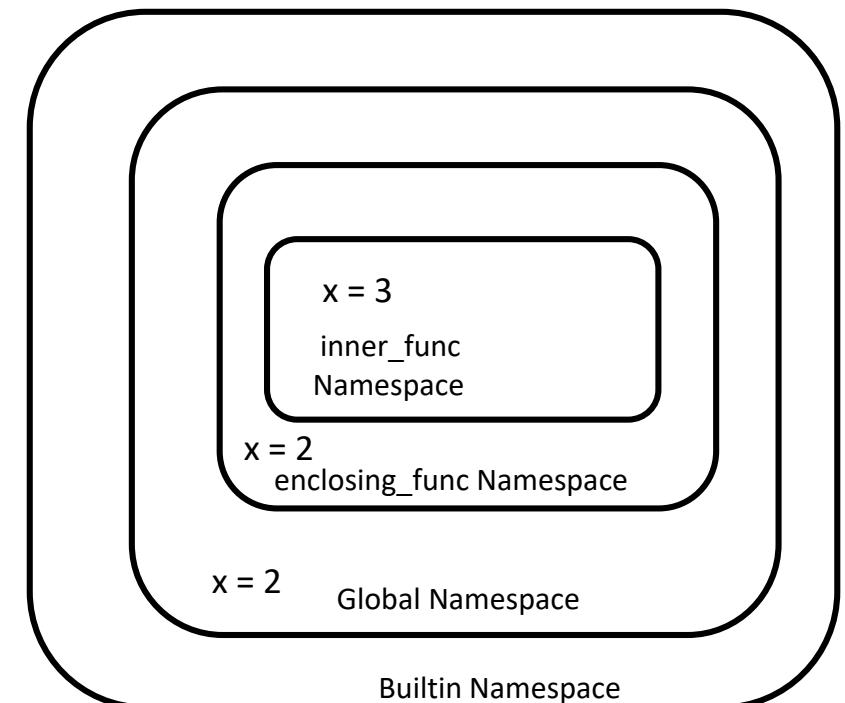
4



Nested Functions

Namespace of inner function is different from enclosing function and global namespace

```
def enclosing_func(x):
    def inner_func(x):
        x += 1
        return x**2
    print(x)
    output = inner_func(x)
    print(x)
    return output
x = 2
print(x)
print(enclosing_func(x))
print(x)
```



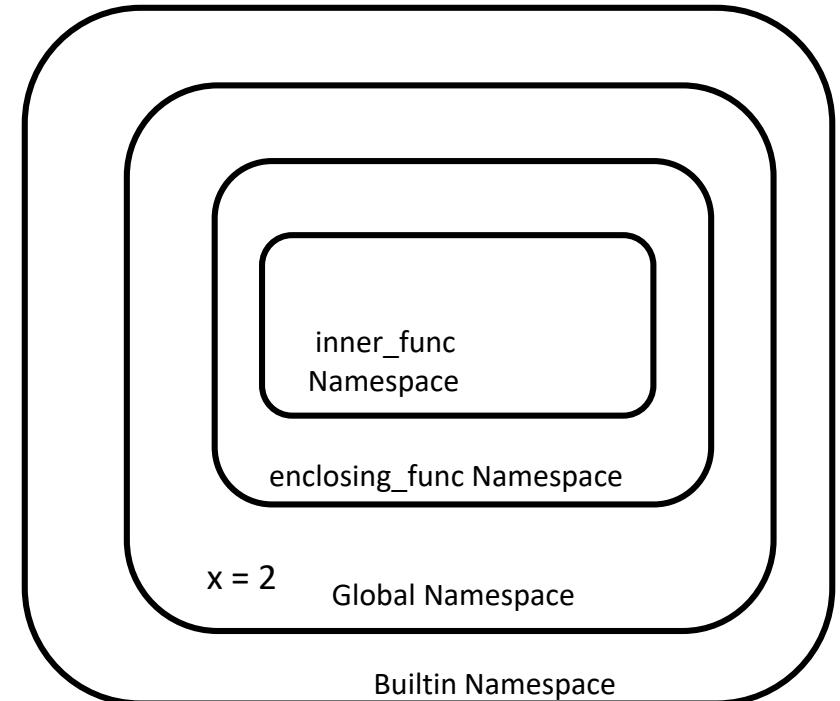
Output:

```
2
2
2
9
2
```

Nested Functions

Inner functions can access global variables

```
def enclosing_func():
    def inner_func():
        return x**2
    output = inner_func()
    return output
x = 2
print(enclosing_func())
```

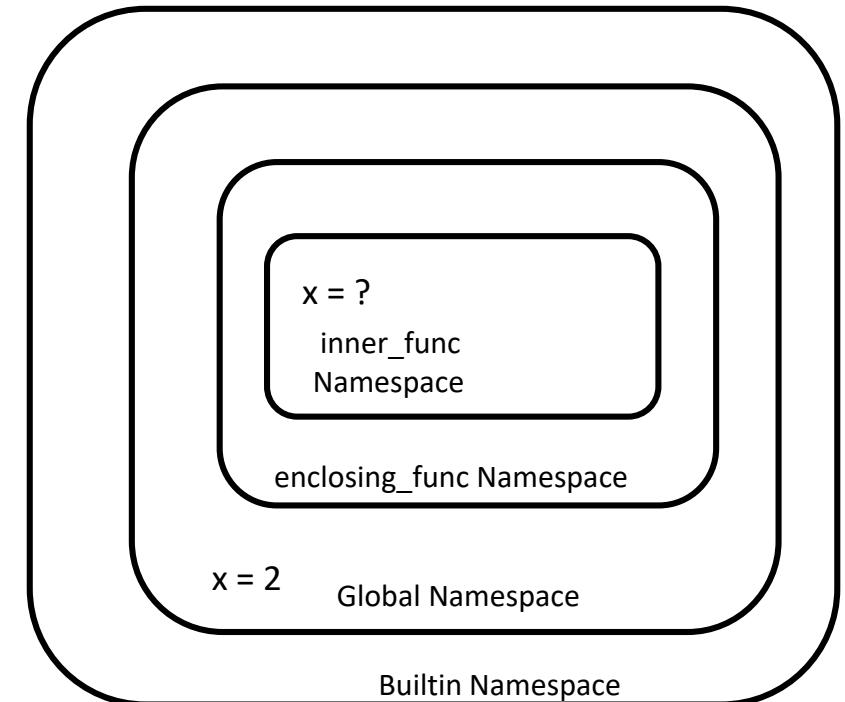


Output:

Nested Functions

Inner functions cannot modify global variables

```
def enclosing_func():
    def inner_func():
        x = x+2
        return x**2
    output = inner_func()
    return output
x = 2
print(enclosing_func())
```



UnboundLocalError: local variable 'x' referenced before assignment

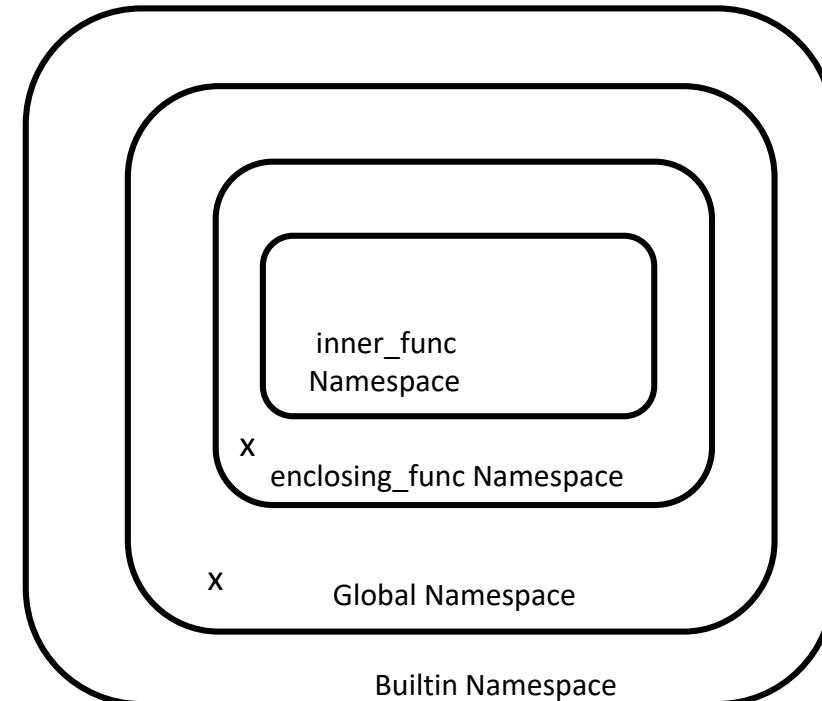
Nested Functions: *global* keyword

Modifying global variable from inner function

```
def enclosing_func():
    x = 3
    def inner_func():
        global x
        print(x)
        x = 1
        print(x)
    print(x)
    inner_func()
    print(x)
x = 5
print(x)
print(enclosing_func())
print(x)
```

nonlocal/enclosing
namespace
for inner function

global keyword
binds this variable to
global variable x



Output:

```
5
3
5
1
3
None
1
```

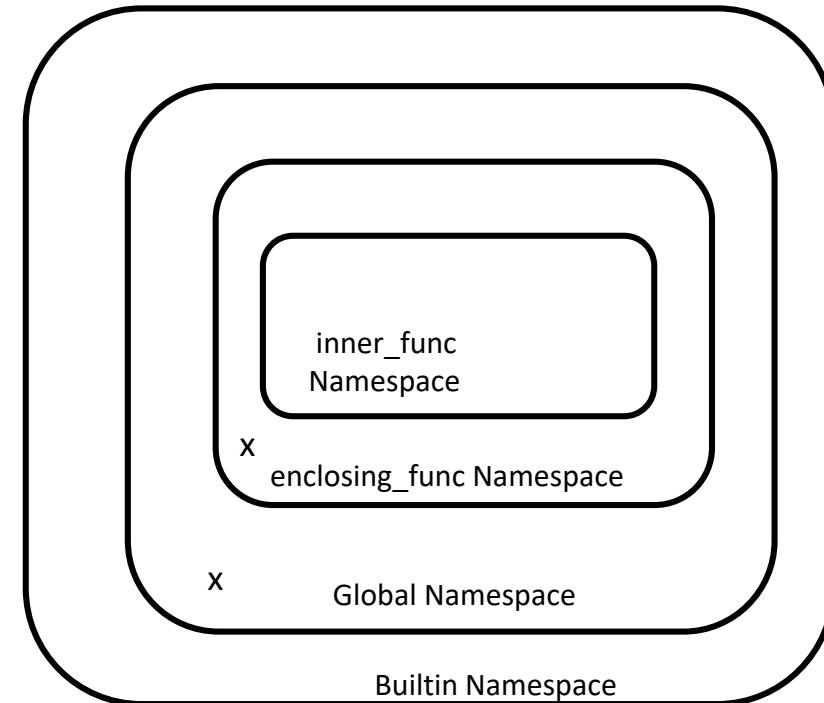
Nested Functions

Inner functions can access nonlocal variables

```
def enclosing_func():
    x = 3
    def inner_func():
        print(x)

    print(x)
    inner_func()
    print(x)

x = 2
print(x)
print(enclosing_func())
print(x)
```



Output:

2

3

3

3

None

2

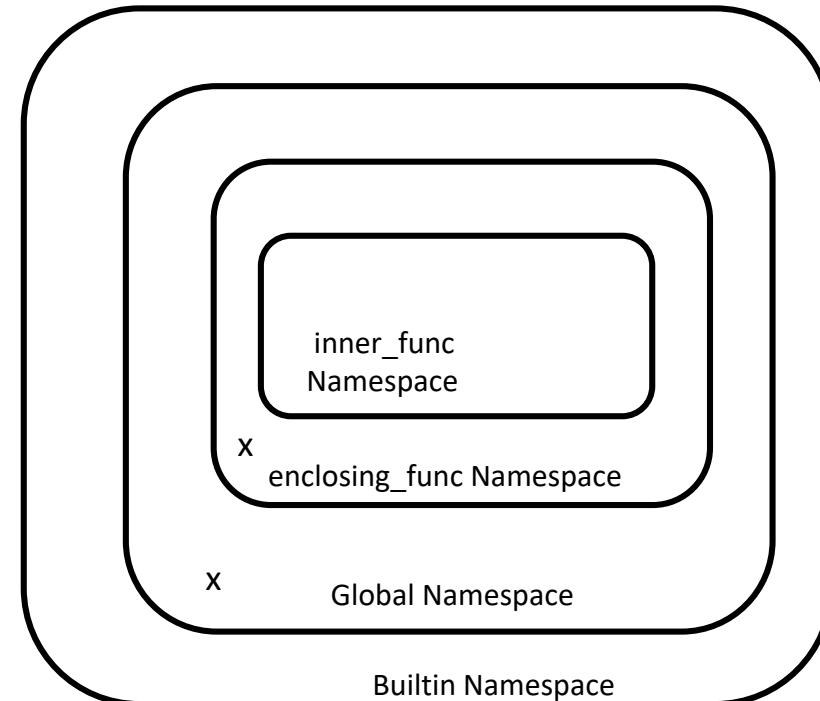
Nested Functions

Inner functions cannot modify nonlocal variables

```
def enclosing_func():
    x = 3
    def inner_func():
        x = x+1
        print(x)

    print(x)
    inner_func()
    print(x)

x = 2
print(x)
print(enclosing_func())
print(x)
```



Output:

UnboundLocalError: local variable 'x' referenced before assignment

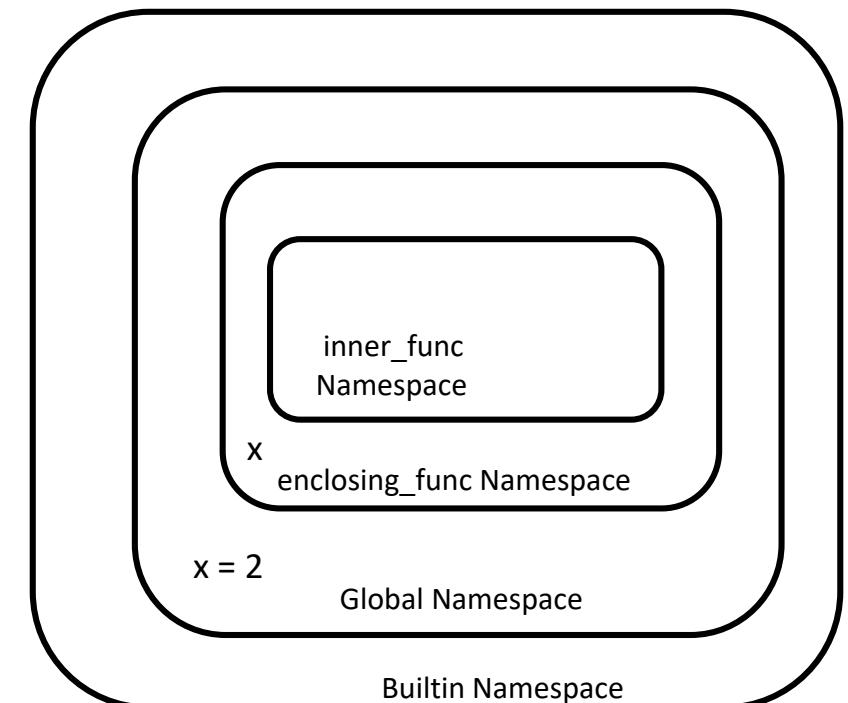
Nested Functions: *nonlocal* keyword

```
Names in enclosing_func namespace  
are nonlocal variables for inner_func  
  
def enclosing_func():  
    x = 3  
    def inner_func():  
        nonlocal x  
        print(x)  
        x = x+1  
        print(x)  
    print(x)  
    inner_func()  
    print(x)  
  
x = 2  
print(x)  
print(enclosing_func())  
print(x)
```

Binds this name to variable
in nearest enclosing namespace

Output:

2
3
3
4
4
None
2



Nested Functions

```
def p1(x):
    y = 2
    print('Entering p1')
    def p2(x):
        print('Entering p2')
        def p3(x):
            print('Entering p3')
            output = x**2
            print('Leaving p3')
            return output
        output = p3(x)
        print('Leaving p2')
        return output
    output = p2(x)
    print('Leaving p1')
    return output
```

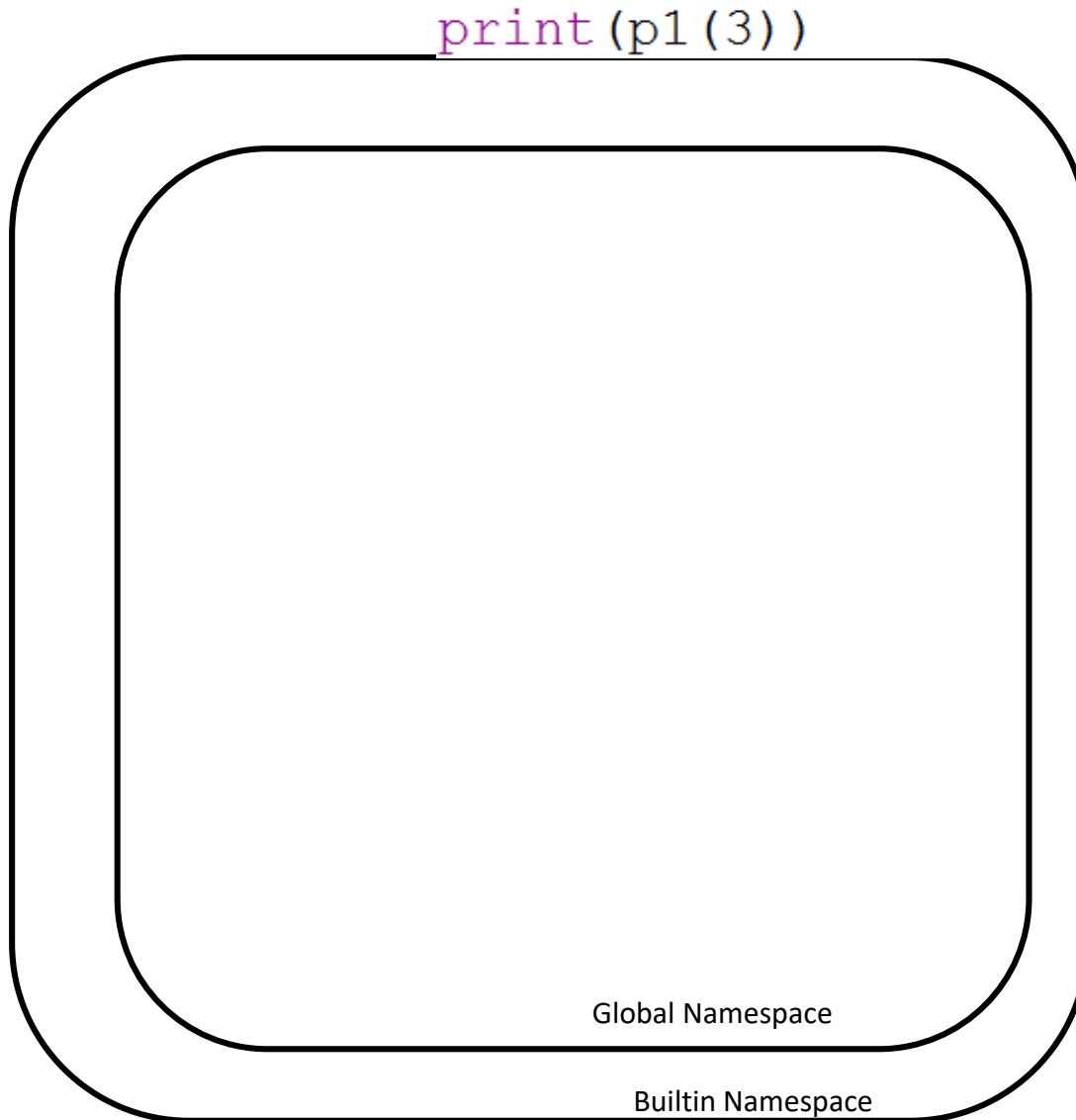
enclosing function
for p1() and p3()

Inner function for
p1() and
enclosing function for p3()

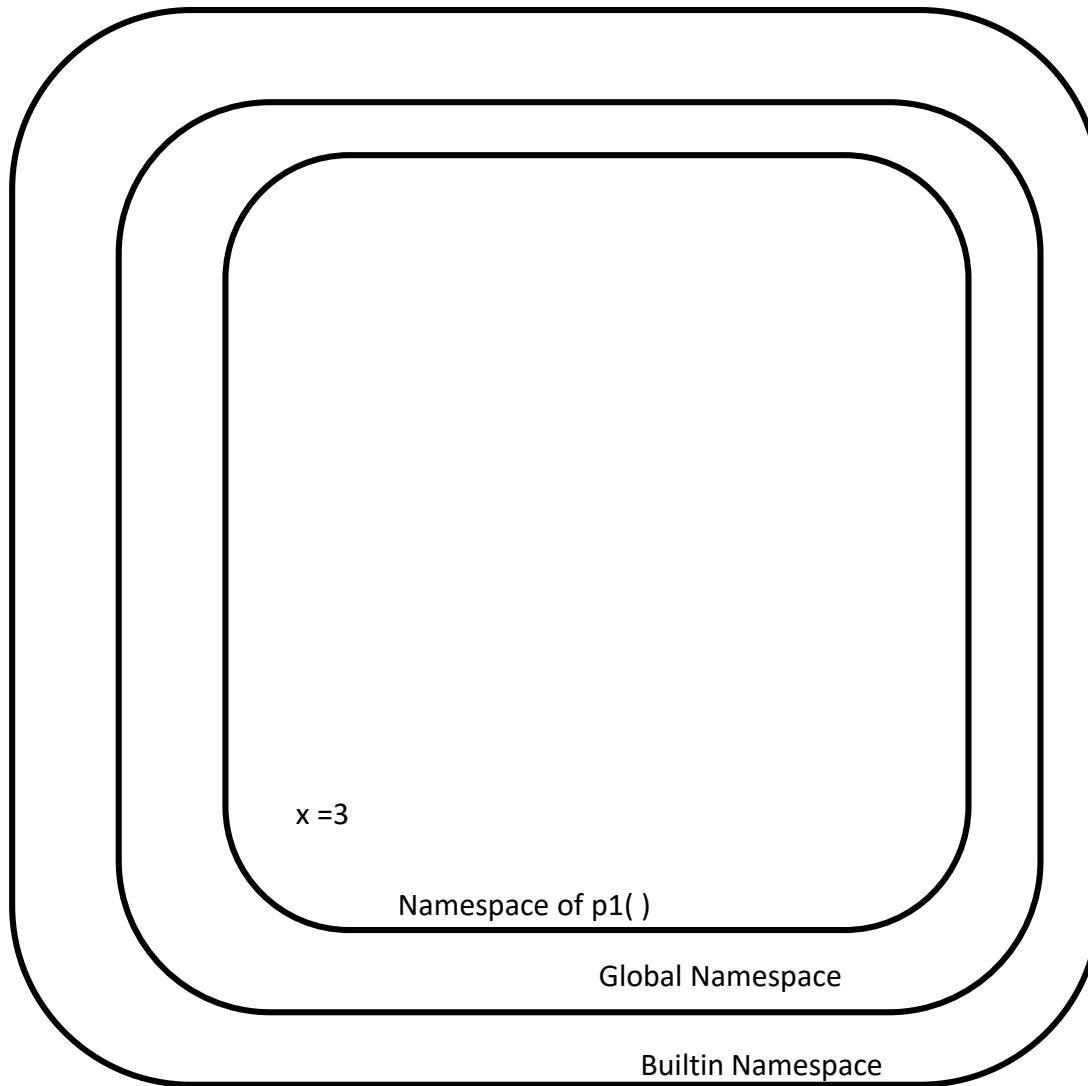
Inner function for
p1() and p2()

```
print(p1(3))
```

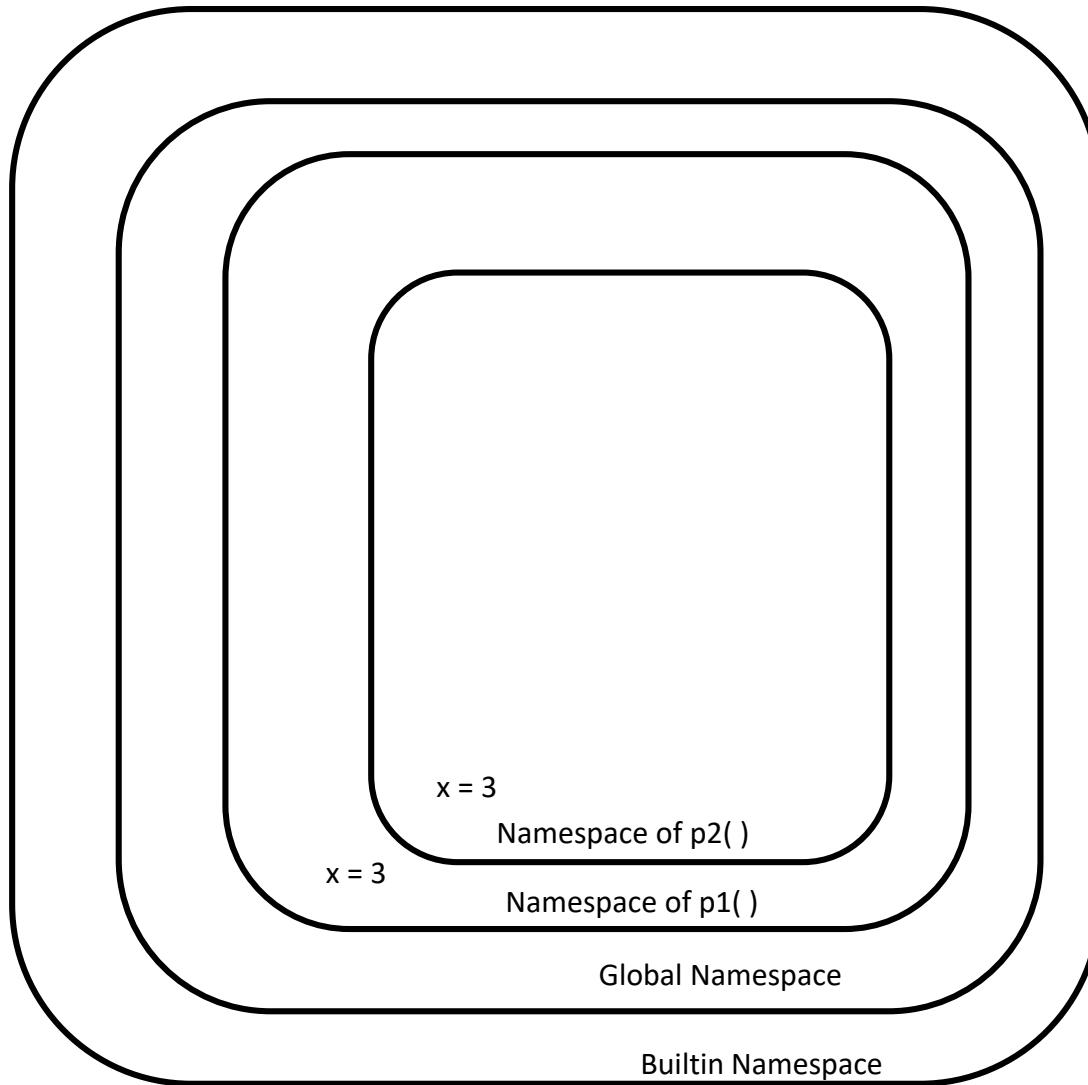
Namespaces: Nested Functions



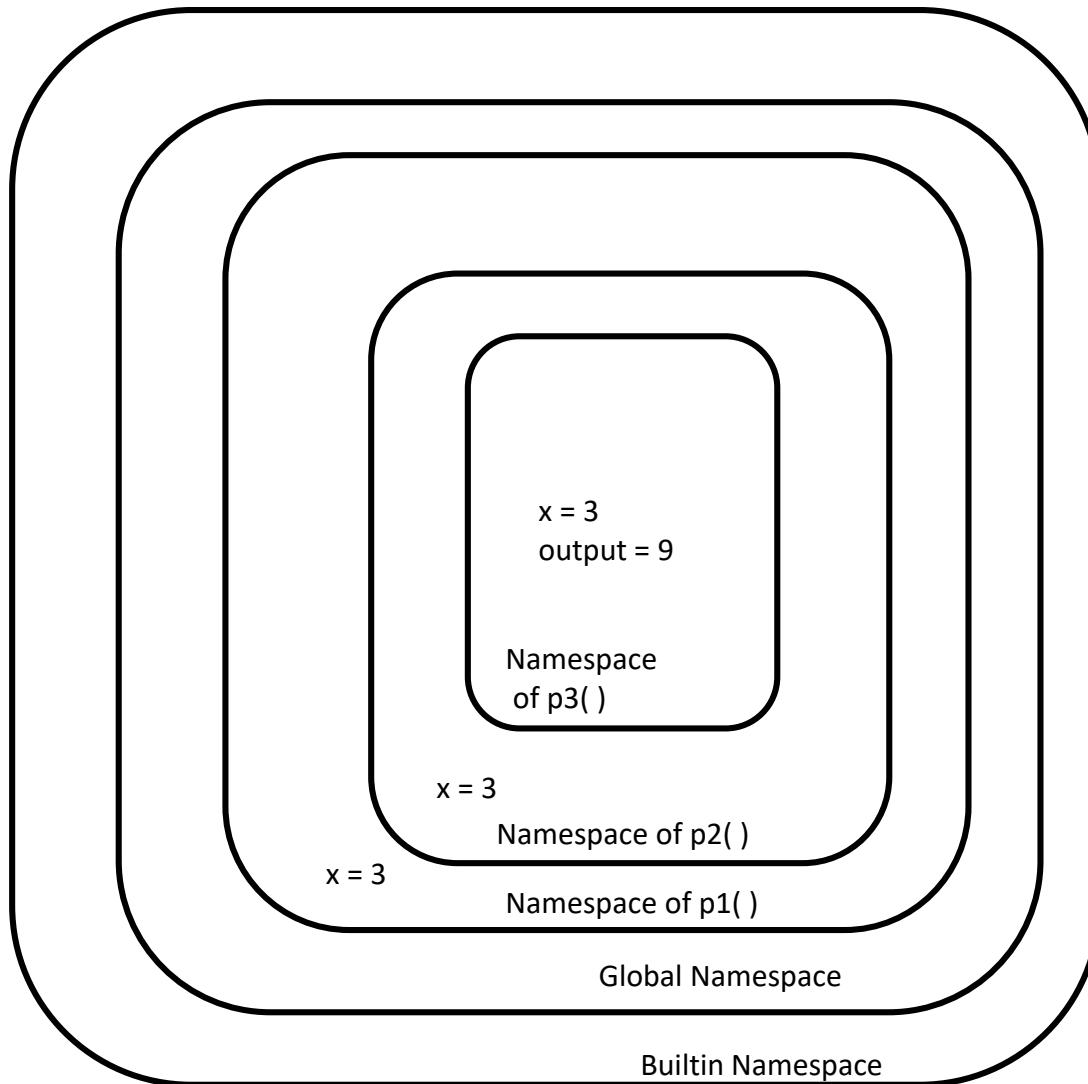
Namespaces: Nested Functions



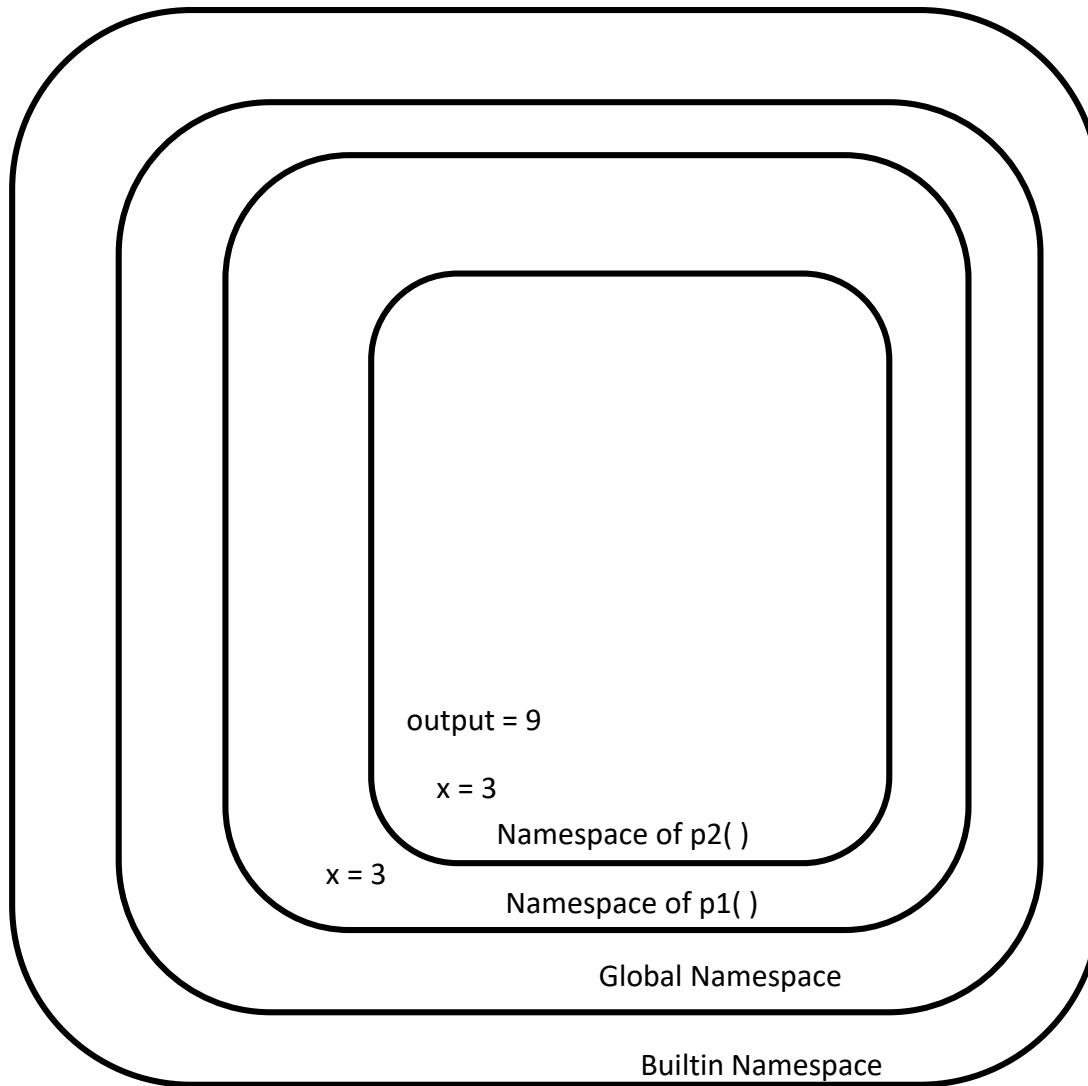
Namespaces: Nested Functions



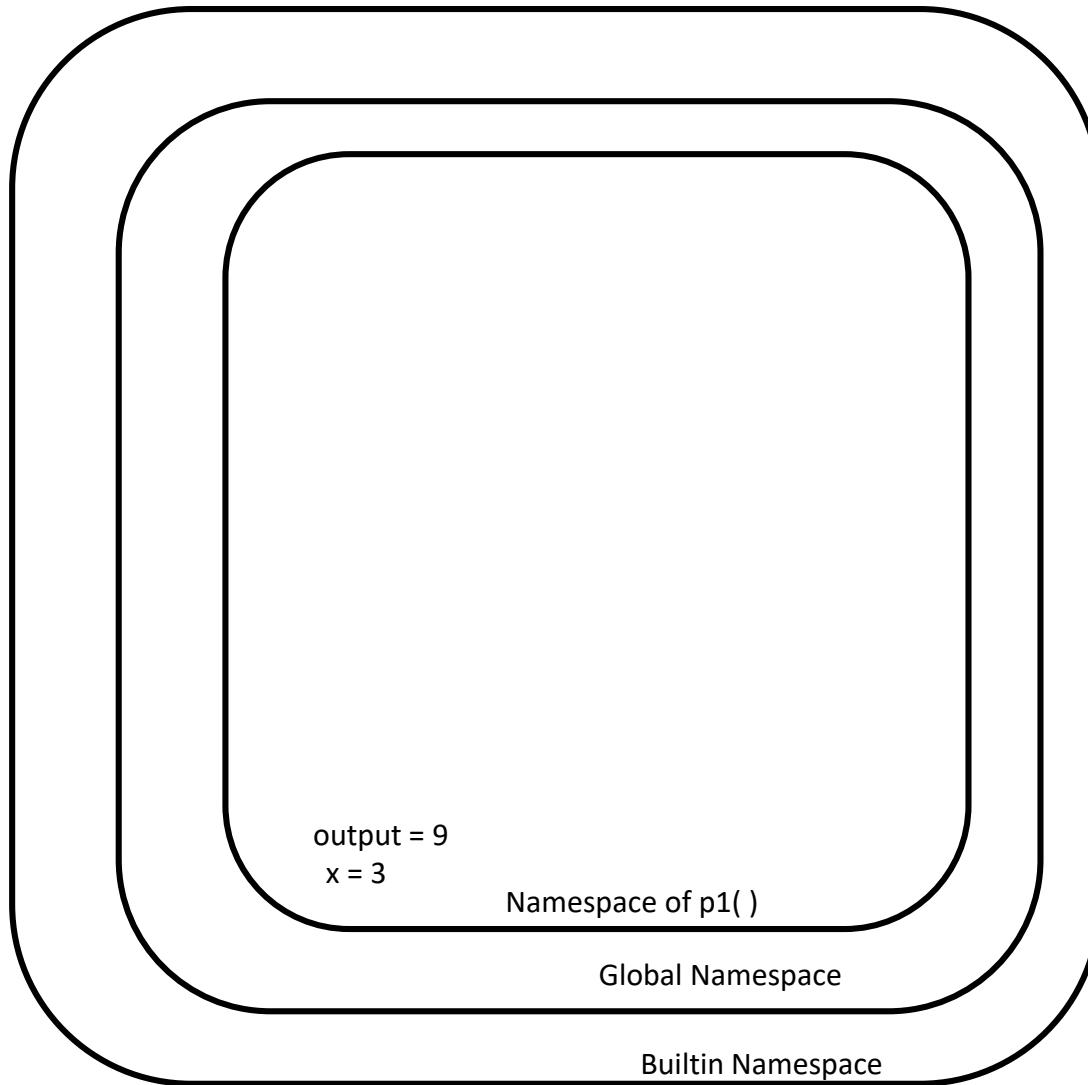
Namespaces: Nested Functions



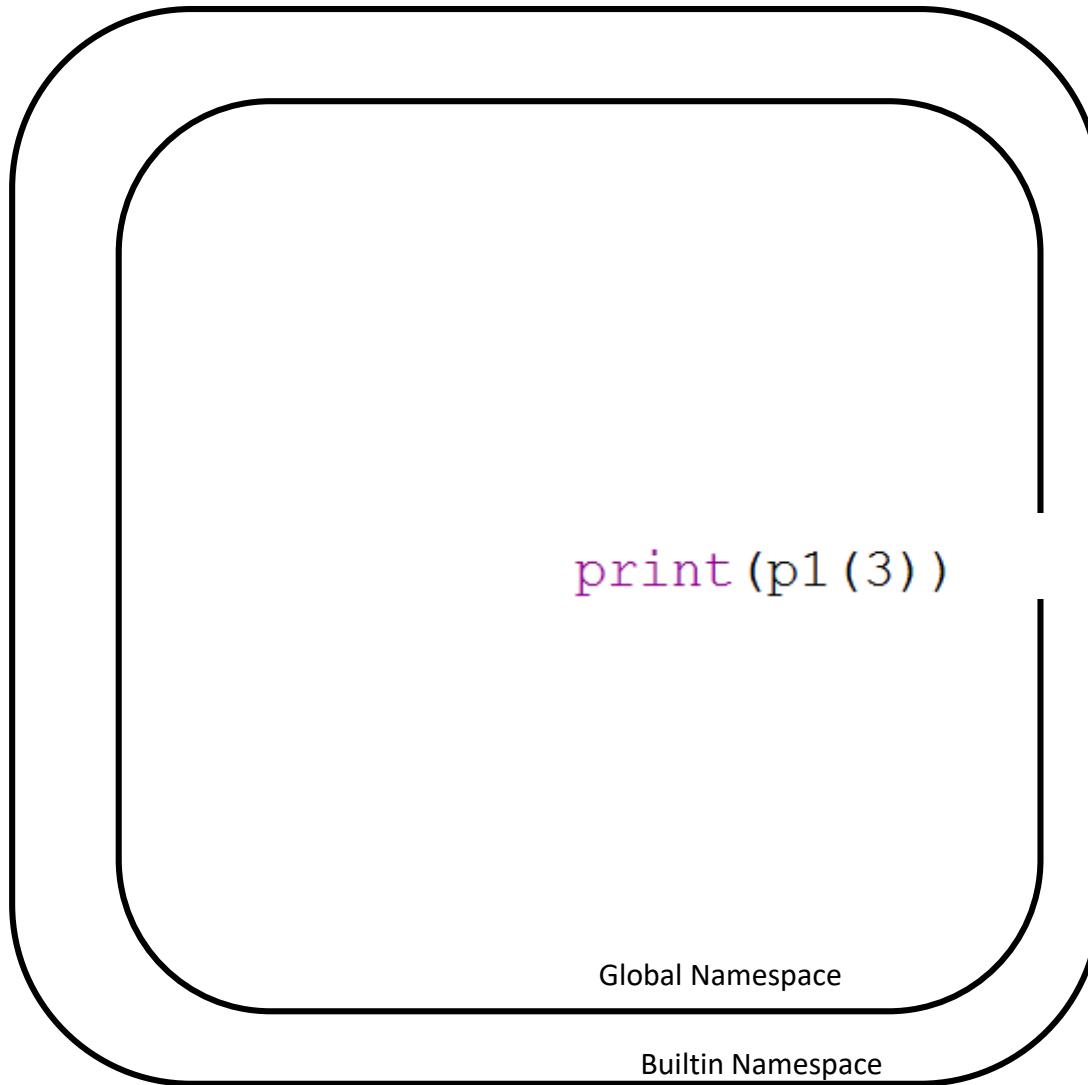
Namespaces: Nested Functions



Namespaces: Nested Functions



Namespaces: Nested Functions



What is the output?

```
def p1(x):
    y = 2
    print('Entering p1')
    def p2(x):
        print('Entering p2')
        z = 4
        def p3(x):
            print('Entering p2')
            output = x**2 +y**2+z**2
            print('Leaving p3')
            return output
        output = p3(x)
        print('Leaving p2')
        return output
    output = p2(x)
    print('Leaving p1')
    return output

print(p1(3))
```

What is the output?

```
def p1(x):
    y = 2
    print('Entering p1')
    def p2(x):
        print('Entering p2')
        z = 4
        def p3(x):
            print('Entering p2')
            output = x**2
            print('Leaving p3')
            return output
        output = p3(x)
        print('Leaving p2')
        return output
    output = p2(x)+z**2
    print('Leaving p1')
    return output

print(p1(3))
```

Where do we use inner functions?

- Higher-Order Functions (Week 5)



NUS | Computing

National University
of Singapore

IT5001 Software Development Fundamentals

5. Recursion Vs Iterations and Nested Functions

Sirigina Rajendra Prasad

Recursion vs Iteration

Reversing a String

- How about reversing a string? Of course, we can just use string slicing

```
>>> s = 'abcde12345'  
>>> s[::-1]  
'54321edcba'  
>>>
```

- How about we write a function for it?

```
>>> reverseStringI(s)  
'54321edcba'  
>>>
```

Reverse String (Iterative Version 1)

```
def reverseStringI(s):
    output = ''
    l = len(s)
    for i in range(l):
        output += s[l-i-1]
    return output

>>> reverseStringI('abcde')
'edcba'
```

i
0
1
2
3
4

Reverse String (Iterative Version 1)

```
def reverseStringI(s):
    output = ''
    l = len(s)
    for i in range(l):
        output += s[l-i-1]
    return output
```

```
>>> reverseStringI('abcde')
'edcba'
```

i	l-i-1
0	4
1	3
2	2
3	1
4	0

Reverse String (Iterative Version 1)

```
def reverseStringI(s):
    output = ''
    l = len(s)
    for i in range(l):
        output += s[l-i-1]
    return output

>>> reverseStringI('abcde')
'edcba'
```

i	l-i-1	s[l-i-1]
0	4	e
1	3	d
2	2	c
3	1	b
4	0	a

Reverse String (Iterative Version 1)

```
def reverseStringI(s):
    output = ''
    l = len(s)
    for i in range(l):
        output += s[l-i-1]
    return output

>>> reverseStringI('abcde')
'edcba'
```

i	l-i-1	s[l-i-1]	output
0	4	e	e
1	3	d	ed
2	2	c	edc
3	1	b	edcb
4	0	a	edcba

Reverse String (Iterative Version 2)

```
def reverseStringI(s):
    output = ''
    for c in s:
        output = c + output
    return output

>>> reverseStringI('abcde')
'edcba'
```

c	output
a	a
b	ba
c	cba
d	dcba
e	edcba

Reversing String (Recursive Version)

```
def reverseStringR(s):
    if not s:
        return ''
    return reverseStringR(s[1:]) + s[0]
```

- `reverseStringR('abcde')`
- `reverseStringR('bcde')+'a'`
- `reverseStringR('cde')+'b'+'a'`
- `reverseStringR('de')+'c'+'b'+'a'`
- `reverseStringR('e')+'d'+'c'+'b'+'a'`
- `reverseStringR('')+'e'+'d'+'c'+'b'+'a'`
- `'+'+'e'+'d'+'c'+'b'+'a'`
- `'edcba'`

Taylor Series

$$\begin{aligned}\sin x &= \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots &\text{for all } x \\ \cos x &= \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots &\text{for all } x \\ \tan x &= \sum_{n=1}^{\infty} \frac{B_{2n}(-4)^n (1-4^n)}{(2n)!} x^{2n-1} &= x + \frac{x^3}{3} + \frac{2x^5}{15} + \dots &\text{for } |x| < \frac{\pi}{2} \\ \sec x &= \sum_{n=0}^{\infty} \frac{(-1)^n E_{2n}}{(2n)!} x^{2n} &= 1 + \frac{x^2}{2} + \frac{5x^4}{24} + \dots &\text{for } |x| < \frac{\pi}{2} \\ \arcsin x &= \sum_{n=0}^{\infty} \frac{(2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1} &= x + \frac{x^3}{6} + \frac{3x^5}{40} + \dots &\text{for } |x| \leq 1 \\ \arccos x &= \frac{\pi}{2} - \arcsin x & & \\ &= \frac{\pi}{2} - \sum_{n=0}^{\infty} \frac{(2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1} &= \frac{\pi}{2} - x - \frac{x^3}{6} - \frac{3x^5}{40} - \dots &\text{for } |x| \leq 1 \\ \arctan x &= \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} x^{2n+1} &= x - \frac{x^3}{3} + \frac{x^5}{5} - \dots &\text{for } |x| \leq 1, x \neq \pm i\end{aligned}$$

Taylor Series

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

$$= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \quad \text{for all } x$$

$n = 0 \quad n = 1 \quad n = 2$

- We do not need the infinite precision
- We may just sum up to k terms

$$\sin x = \sum_{n=0}^k \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

$$= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \quad \text{for all } x$$

Computing sine by Iteration

$$\sin x = \sum_{n=0}^{\cancel{k}} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \quad \text{for all } x$$

- Using iteration

```
def sinI(x, k):  
    result = 0  
    for n in range(0, k):  
        result += ((-1)**n / fact(2*n+1)) * x**(2*n+1)  
    return result
```

```
>>> print(sinI(PI/6, 10))  
0.5000000000592083  
>>> from math import sin  
>>> sin(PI/6) ←  
0.5000000000592083
```

Python Library version of “sin()”

Computing sine by Recursion

Sum up to $n = 2$

$$\sin x = \sum_{n=0}^{\cancel{k}} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

$$= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$$

for all x

Sum up to $n = 1$

- In general, if we want to sum up to the k terms

Sum up to $n = k$

$$\sin x = \sum_{n=0}^{\cancel{k}} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

$$= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$$

k^{th}

for all x

Sum up to $n = k - 1$

$n = k$

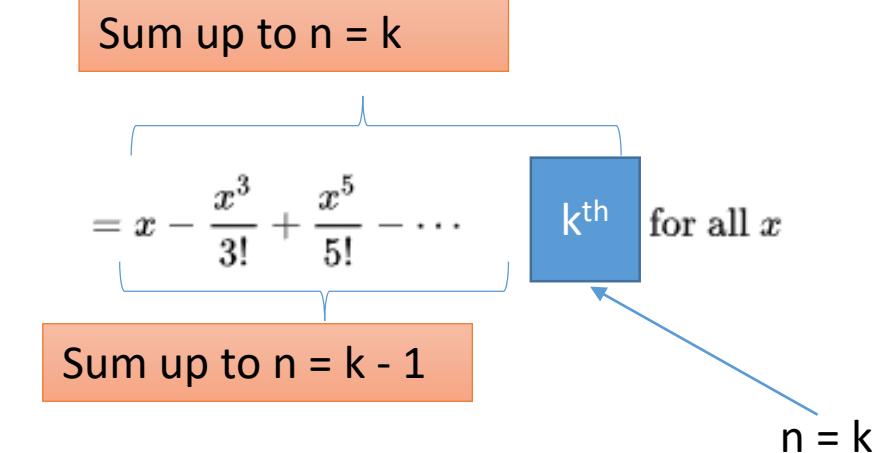
Computing sine by Recursion

- Assuming that if the function $\sinR(x, k)$ sums until $n = k$, then

$$\sinR(x, k) = \sinR(x, k-1) + \text{the } k^{\text{th}} \text{ term}$$

- In general, if we want to sum up to the k terms

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$



Computing sine by Recursion

- Assuming that if the function $\sinR(x, k)$ sums until $n = k$, then

$$\sinR(x, k) = \sinR(x, k-1) + \text{the } k\text{th term}$$

```
def sinR(x, k):  
    if k < 0:  
        return 0  
    return sinR(x, k-1) + ((-1)**k / fact(2*k+1)) * x**(2*k+1)
```

```
>>> sinR(PI/6, 6)  
0.500000000592083  
>>> from math import sin  
>>> sin(PI/6)  
0.500000000592083
```

More Taylor Series

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \quad \text{for all } x$$

$$\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots \quad \text{for all } x$$

$$\tan x = \sum_{n=1}^{\infty} \frac{B_{2n} (-4)^n (1 - 4^n)}{(2n)!} x^{2n-1} = x + \frac{x^3}{3} + \frac{2x^5}{15} + \dots \quad \text{for } |x| < \frac{\pi}{2}$$

$$\sec x = \sum_{n=0}^{\infty} \frac{(-1)^n E_{2n}}{(2n)!} x^{2n} = 1 + \frac{x^2}{2} + \frac{5x^4}{24} + \dots \quad \text{for } |x| < \frac{\pi}{2}$$

$$\arcsin x = \sum_{n=0}^{\infty} \frac{(2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1} = x + \frac{x^3}{6} + \frac{3x^5}{40} + \dots \quad \text{for } |x| \leq 1$$

$$\begin{aligned} \arccos x &= \frac{\pi}{2} - \arcsin x \\ &= \frac{\pi}{2} - \sum_{n=0}^{\infty} \frac{(2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1} = \frac{\pi}{2} - x - \frac{x^3}{6} - \frac{3x^5}{40} - \dots \quad \text{for } |x| \leq 1 \end{aligned}$$

$$\arctan x = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} x^{2n+1} = x - \frac{x^3}{3} + \frac{x^5}{5} - \dots \quad \text{for } |x| \leq 1, x \neq \pm i$$

Recursion Common Patterns

```
def reverseStringR(s):
    if not s:
        return ''
    return reverseStringR(s[1:]) + s[0]
```

```
def sinR(x, k):
    if k < 0:
        return 0
    return sinR(x, k-1) + ((-1)**k / fact(2*k+1)) * x**(2*k+1)
```

→ Base cases

Recursion step to reduce the problem one-by-one

Iteration Common Patterns

```
def reverseStringI(s):
    output = ''
    l = len(s)
    for i in range(l):
        output += s[l-i-1]
    return output

def sinI(x, k):
    result = 0
    for n in range(0, k):
        result += ((-1)**n / fact(2*n+1)) * x**(2*n+1)
    return result
```

Accumulate element one-by-one

- Initial the final answer to “nothing” at the beginning.
- Accumulate and return the final answer

Iteration/Recursion Conversion

```
def sinR(x, k):
    if k < 0:
        return 0
    return sinR(x, k-1) + ((-1)**k / fact(2*k+1)) * x**(2*k+1)
```

Base case

The answer for previous $k - 1$ terms

The k th term

```
def sinI(x, k):
    result = 0
    for n in range(0, k):
        result += ((-1)**n / fact(2*n+1)) * x**(2*n+1)
    return result
```

Iteration/Recursion Conversion

```
def reverseStringR(s):  
    if not s:  
        return ''  
    return reverseStringR(s[1:]) + s[0]
```

Base case

The answer for previous $k - 1$ terms

The k th term

```
def reverseStringI(s):  
    output = ''  
    l = len(s)  
    for i in range(l):  
        output += s[l-i-1]  
    return output
```

“Homework”

$$\arcsin x = \sum_{n=0}^{\infty} \frac{(2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1} = x + \frac{x^3}{6} + \frac{3x^5}{40} + \dots \quad \text{for } |x| \leq 1$$

- The answer for all k-1 terms?
- Base case?
- Kth term?

Another Example

Recursion vs Iteration

- SumDigits
 - Given a positive number n , the sum of all digits is obtained by adding the digit one-by-one
 - For example, the sum of 52634 = $5 + 2 + 6 + 3 + 4 = 20$
 - Write a function `sum(n)` to compute the sum of all the digits in n
- Factorial
 - Factorial is defined (recursively) as $n! = n * (n - 1)!$ such that $0! = 1$
 - Write a function `fact(n)` to compute the value of $n!$
- Can you do it in both recursion and iteration?

SumDigits

Iteration

```
def sum(n):  
    res = 0  
    while n > 0:  
        res = res + n%10  
        n = n//10  
  
    return res
```

base/initial value

computation

continuation/next value

Recursion

```
def sum(n):  
    if n == 0:  
        return 0  
    else:  
        return n%10 + sum(n//10)
```

stop/base case (*they are related, how?*)

temporary result variables
not needed in recursion (*why?*)

Factorial

Iteration

```
def fact(n):  
    res = 1  
    while n > 0:  
        res = res * n  
        n = n-1  
  
    return res
```

base/initial value

computation

continuation/next value

Recursion

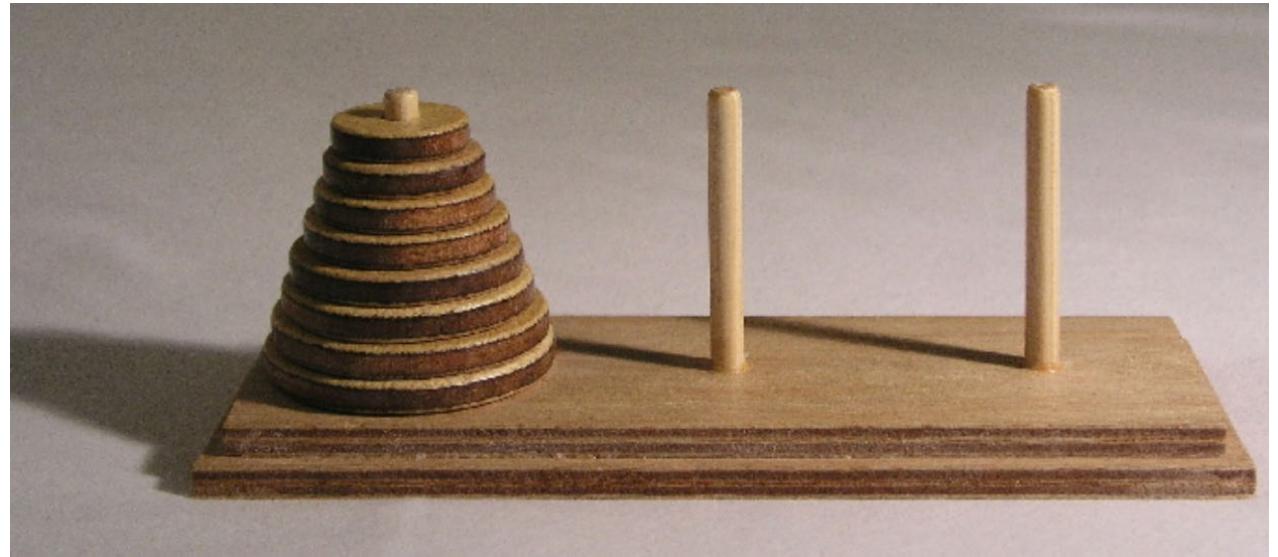
```
def fact(n):  
    if n == 0:  
        return 1  
    else:  
        return n * fact(n-1)
```

stop/base case (*they are related, how?*)

temporary result variables
not needed in recursion (*why?*)

Towers of Hanoi Problem

- Move entire stack to the last tower while obeying the following rules
 - Only one disc may be moved at a time
 - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
 - No disk may be placed on top of a disk that is smaller than it.



https://en.wikipedia.org/wiki/Tower_of_Hanoi

Question: How many moves for n discs?

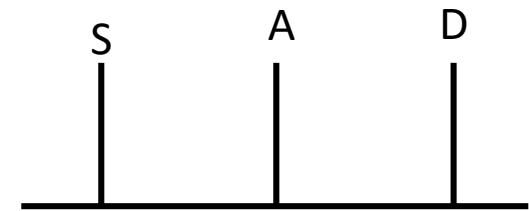
$$2^n - 1$$

Towers of Hanoi

```
def towers_Hanoi(n, S, D, A):
    #Base case
    if n == 1:
        return print(f'Move Disc {n} from Tower {S} to Tower {D}')

    #Move top n-1 discs to auxiliary tower
    towers_Hanoi(n - 1, S, A, D)
    print(f'Move Disc {n} from Tower {S} to Tower {D}')
    #Move top n-1 discs to destination tower
    towers_Hanoi(n - 1, A, D, S)

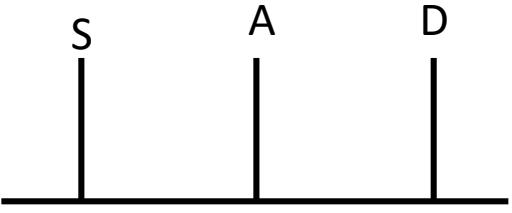
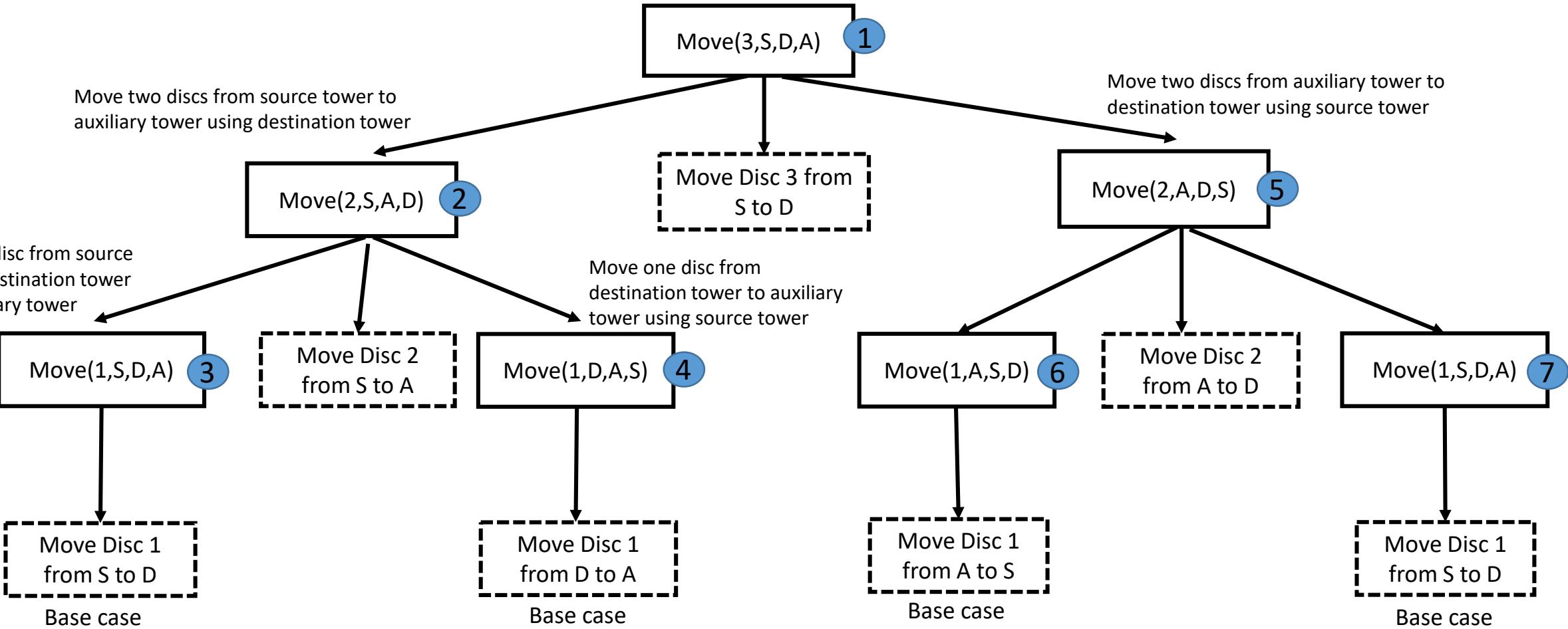
towers_Hanoi(3, 'S', 'D', 'A')
```



Solution:

Move Disc 1 from Tower S to Tower D
Move Disc 2 from Tower S to Tower A
Move Disc 1 from Tower D to Tower A
Move Disc 3 from Tower S to Tower D
Move Disc 1 from Tower A to Tower S
Move Disc 2 from Tower A to Tower D
Move Disc 1 from Tower S to Tower D

Towers of Hanoi

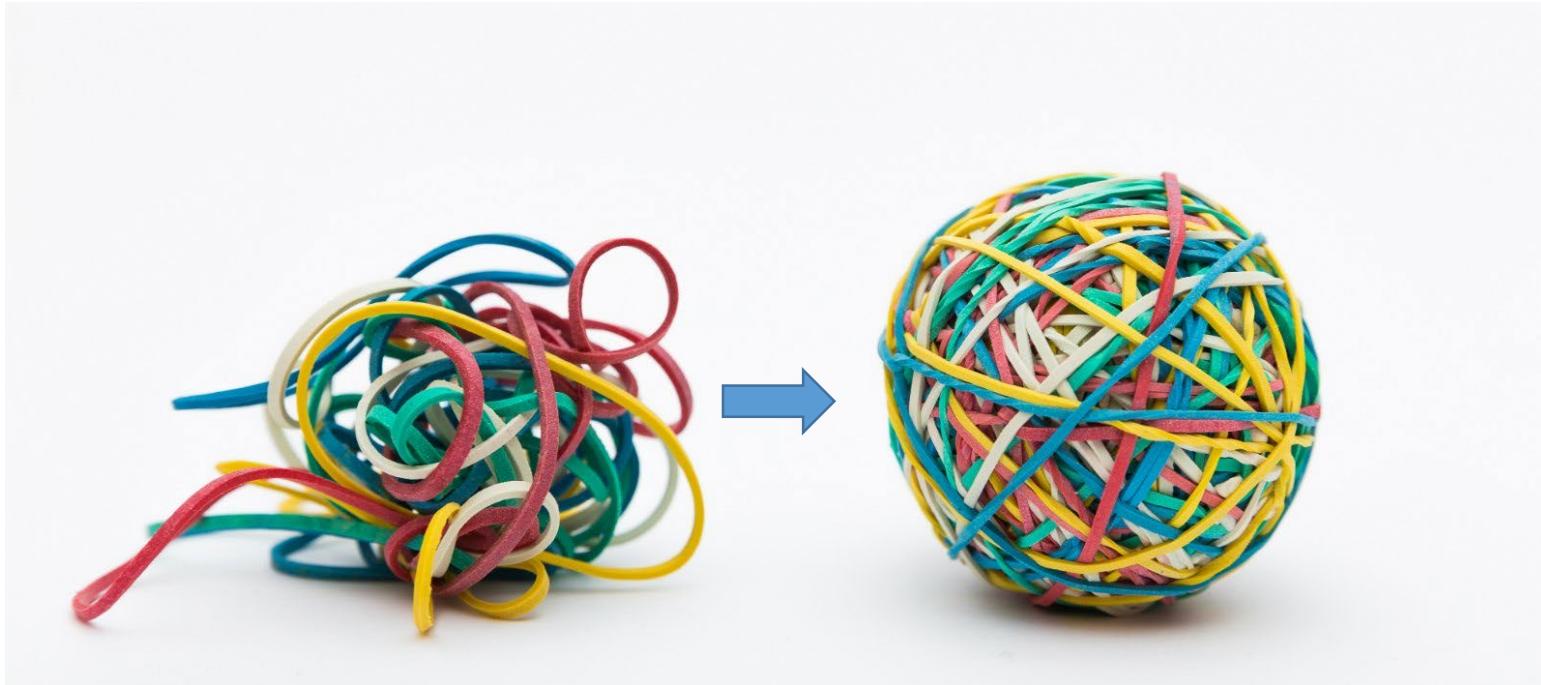


Depth-first
In-order traversal

“Homework”

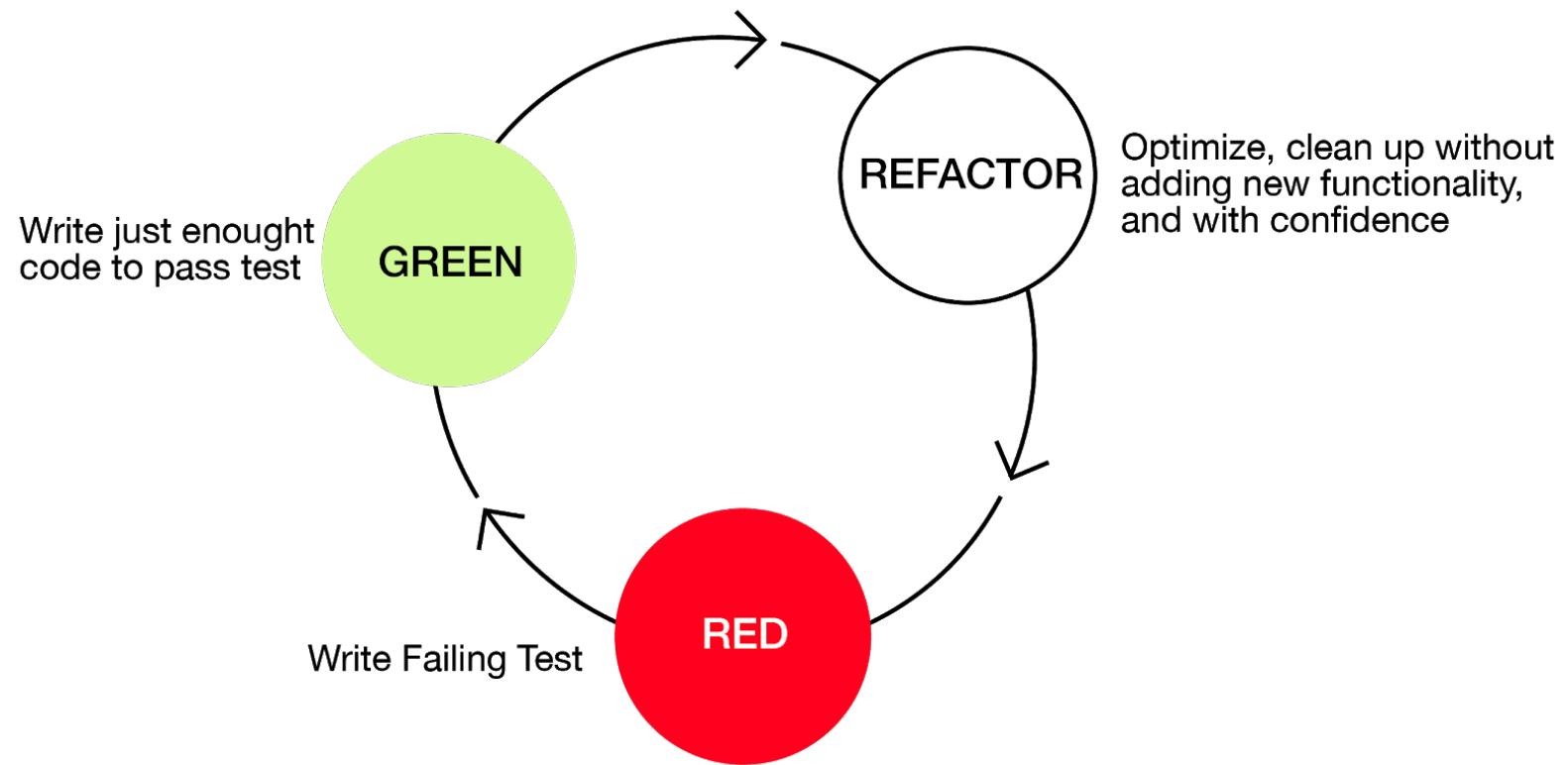
- How to re-write your code with both iterative/recursion version mentioned in this course before?
 - burgerPrice ()
 - checkAllAlpha ()
 - Etc.
- The answer for all k-1 terms?
- Base case?
- Kth term?

Code Refactoring



Code Refactoring

- **Refactoring** is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.



Nested Functions

Nested Functions

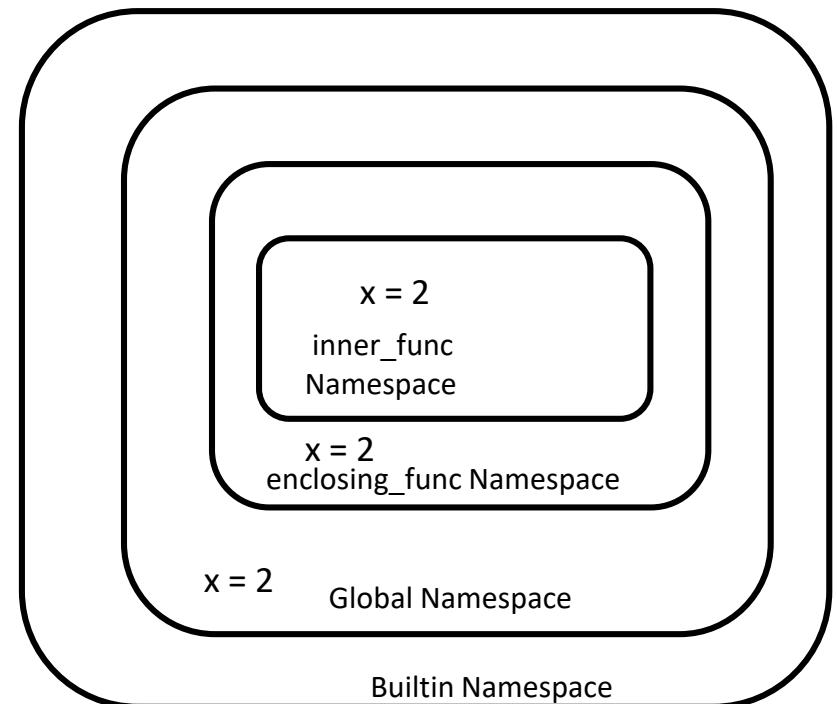
- Functions defined inside other functions

```
x = 2
def enclosing_func(x):
    def inner_func(x):
        return x**2
    output = inner_func(x)
    return output

print(enclosing_func(x))
```

Output:

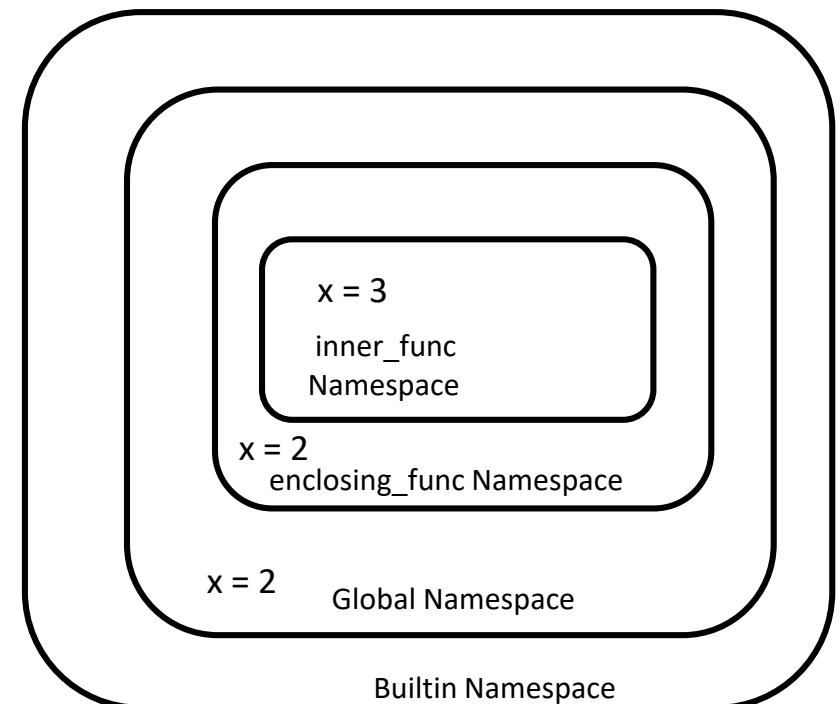
4



Nested Functions

Namespace of inner function is different from enclosing function and global namespace

```
def enclosing_func(x):
    def inner_func(x):
        x += 1
        return x**2
    print(x)
    output = inner_func(x)
    print(x)
    return output
x = 2
print(x)
print(enclosing_func(x))
print(x)
```



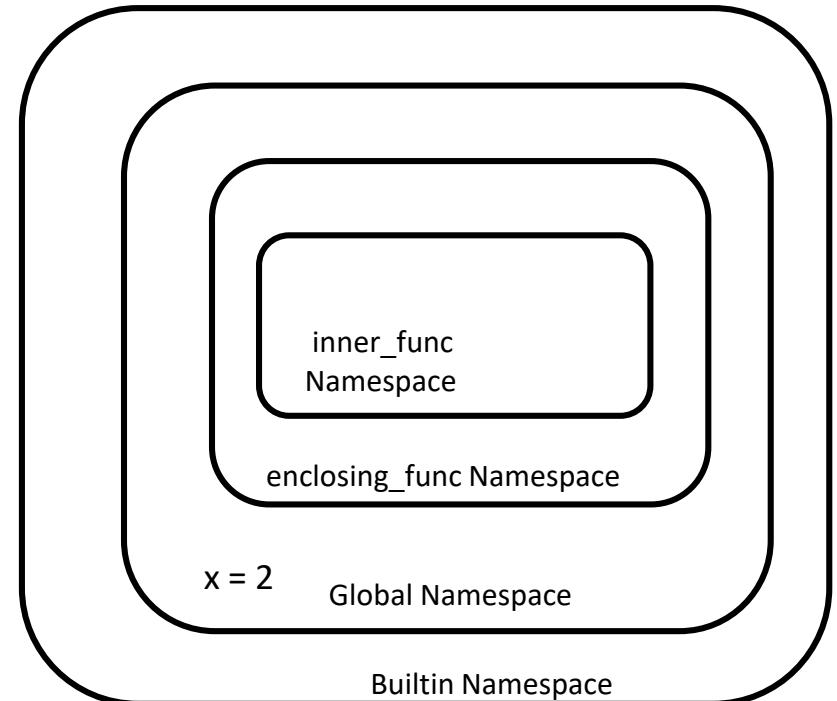
Output:

```
2
2
2
9
2
```

Nested Functions

Inner functions can access global variables

```
def enclosing_func():
    def inner_func():
        return x**2
    output = inner_func()
    return output
x = 2
print(enclosing_func())
```

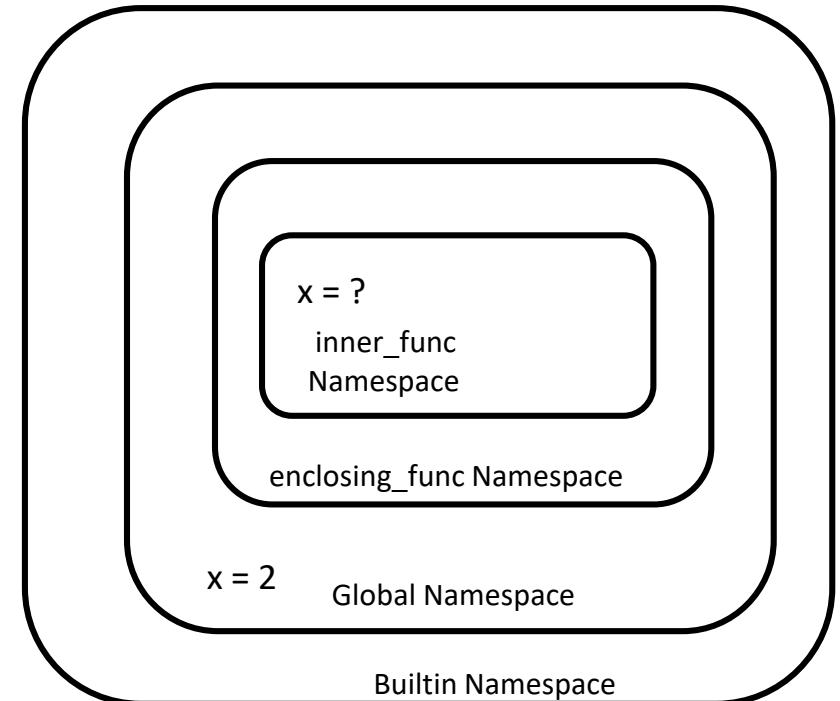


Output:

Nested Functions

Inner functions cannot modify global variables

```
def enclosing_func():
    def inner_func():
        x = x+2
        return x**2
    output = inner_func()
    return output
x = 2
print(enclosing_func())
```



UnboundLocalError: local variable 'x' referenced before assignment

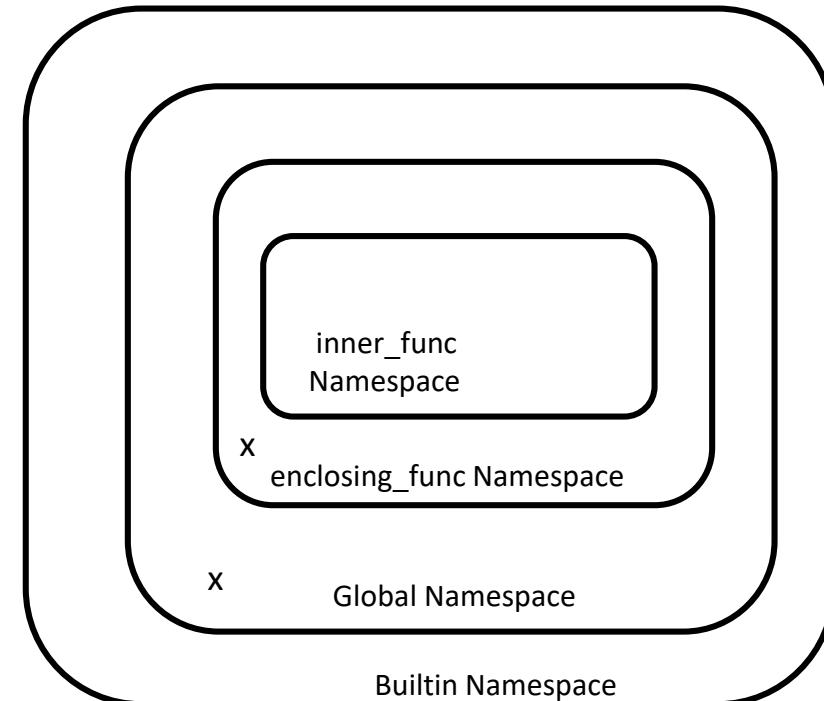
Nested Functions: *global* keyword

Modifying global variable from inner function

```
def enclosing_func():
    x = 3
    def inner_func():
        global x
        print(x)
        x = 1
        print(x)
    print(x)
    inner_func()
    print(x)
x = 5
print(x)
print(enclosing_func())
print(x)
```

nonlocal/enclosing
namespace
for inner function

global keyword
binds this variable to
global variable x



Output:

```
5
3
5
1
3
None
1
```

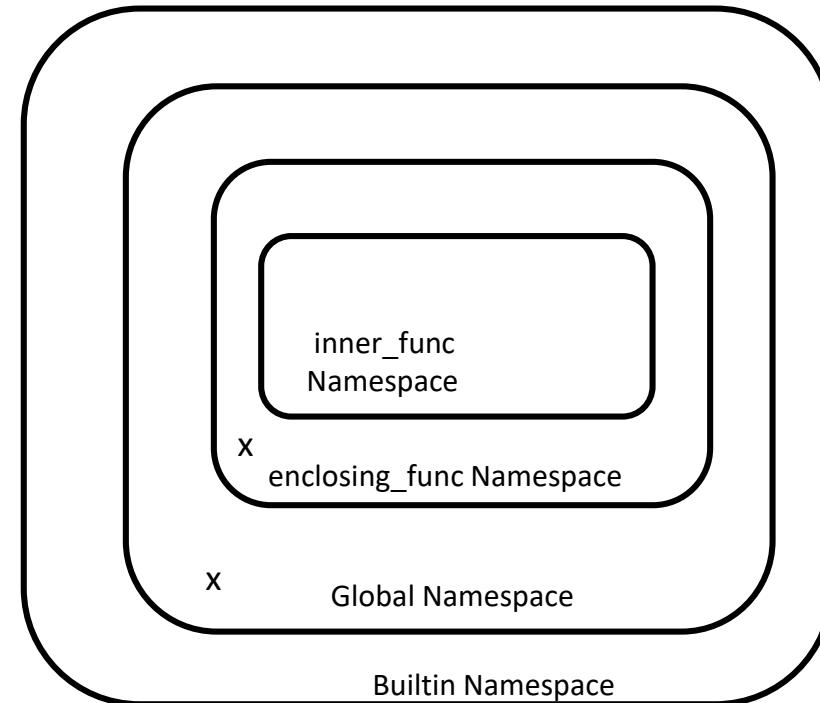
Nested Functions

Inner functions can access nonlocal variables

```
def enclosing_func():
    x = 3
    def inner_func():
        print(x)

    print(x)
    inner_func()
    print(x)

x = 2
print(x)
print(enclosing_func())
print(x)
```



Output:

2

3

3

3

None

2

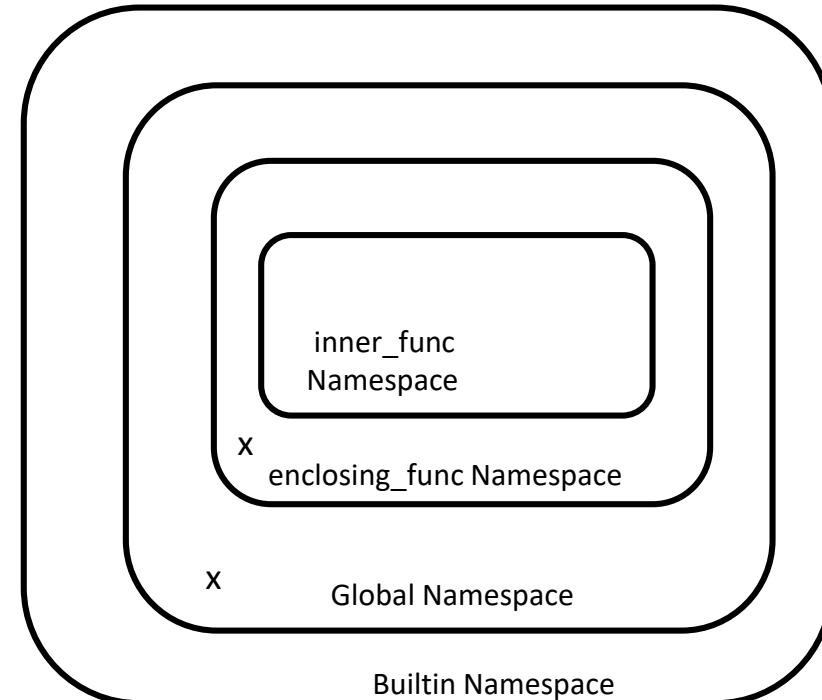
Nested Functions

Inner functions cannot modify nonlocal variables

```
def enclosing_func():
    x = 3
    def inner_func():
        x = x+1
        print(x)

    print(x)
    inner_func()
    print(x)

x = 2
print(x)
print(enclosing_func())
print(x)
```



Output:

UnboundLocalError: local variable 'x' referenced before assignment

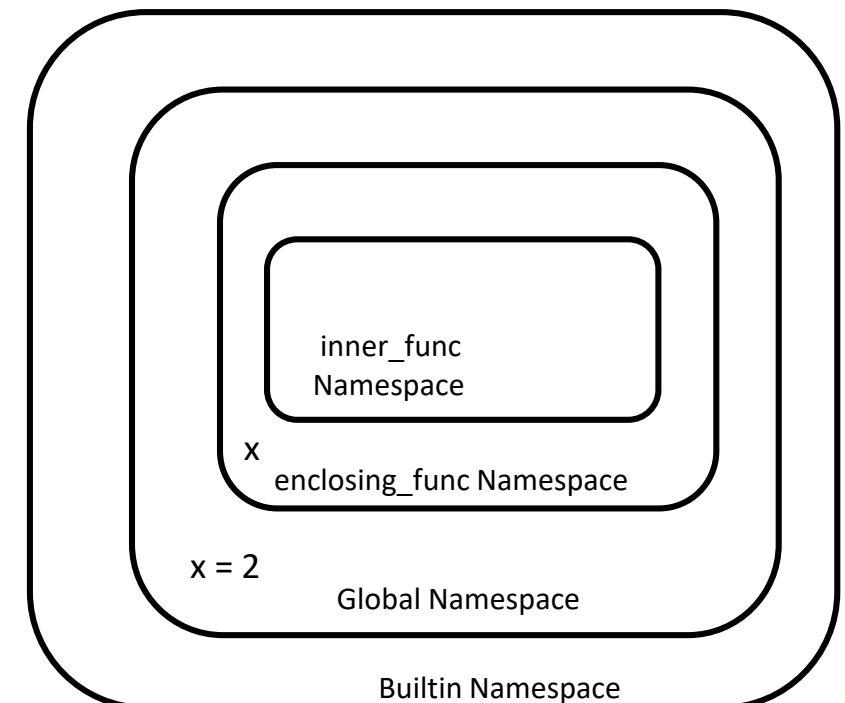
Nested Functions: *nonlocal* keyword

```
Names in enclosing_func namespace  
are nonlocal variables for inner_func  
  
def enclosing_func():  
    x = 3  
    def inner_func():  
        nonlocal x  
        print(x)  
        x = x+1  
        print(x)  
    print(x)  
    inner_func()  
    print(x)  
  
x = 2  
print(x)  
print(enclosing_func())  
print(x)
```

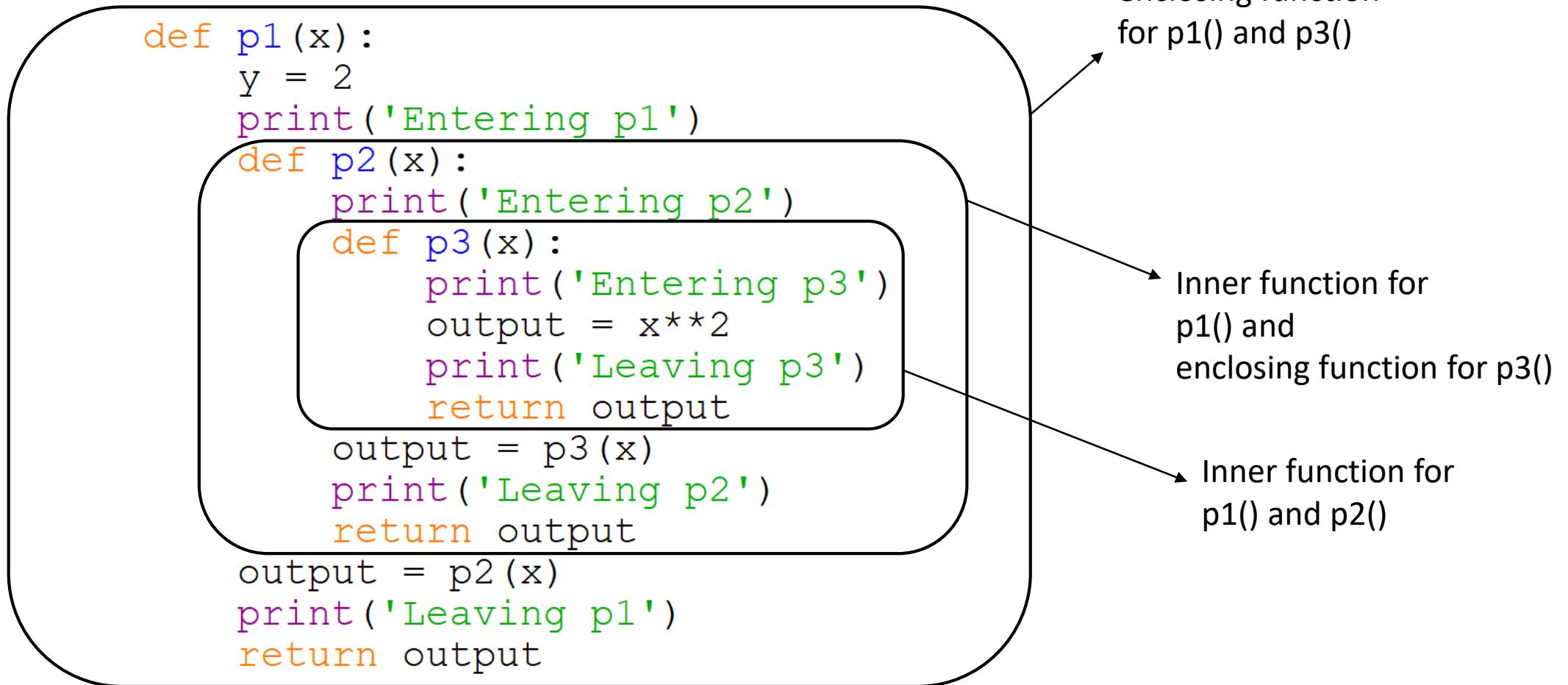
Binds this name to variable
in nearest enclosing namespace

Output:

2
3
3
4
4
None
2

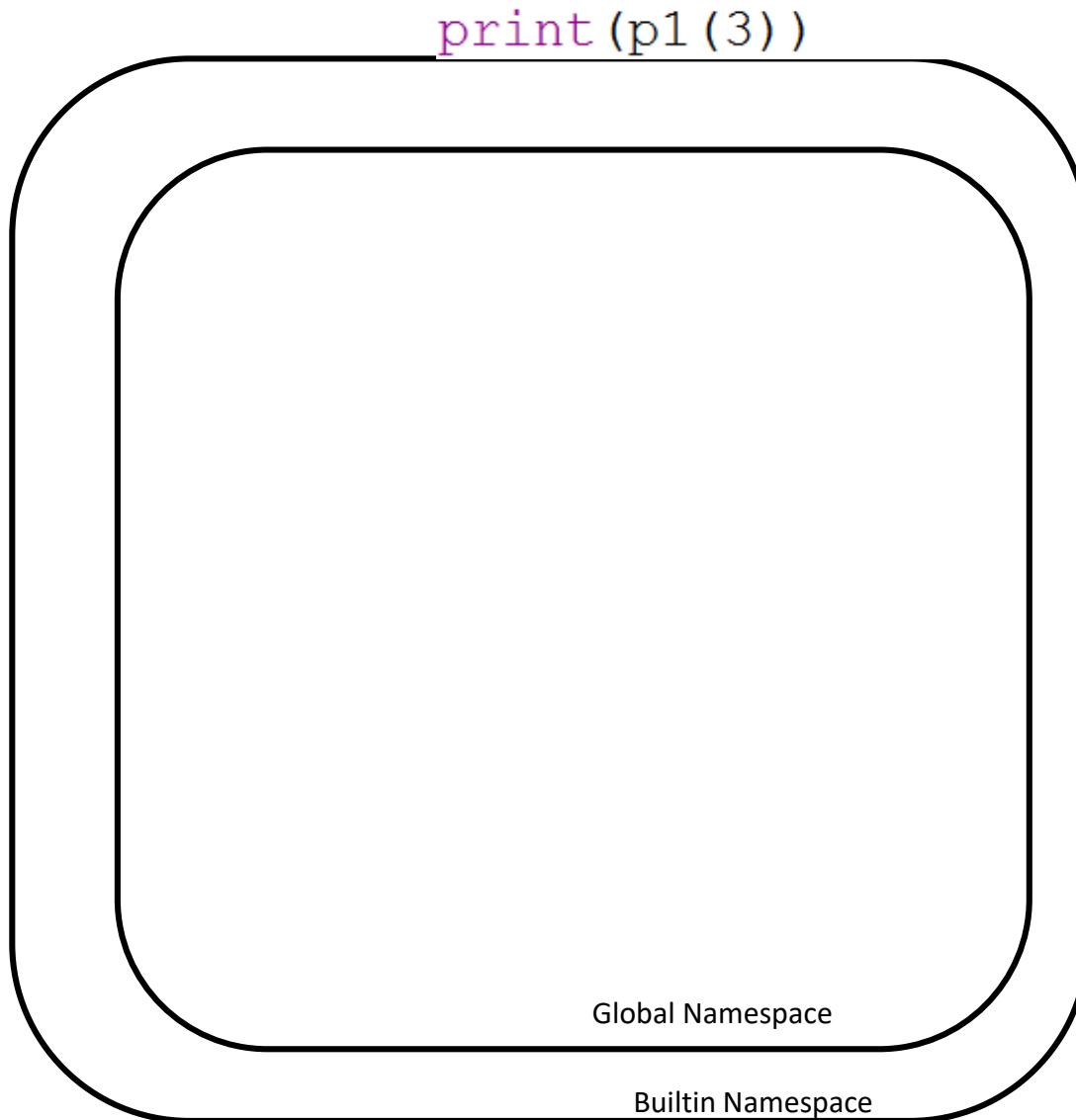


Nested Functions

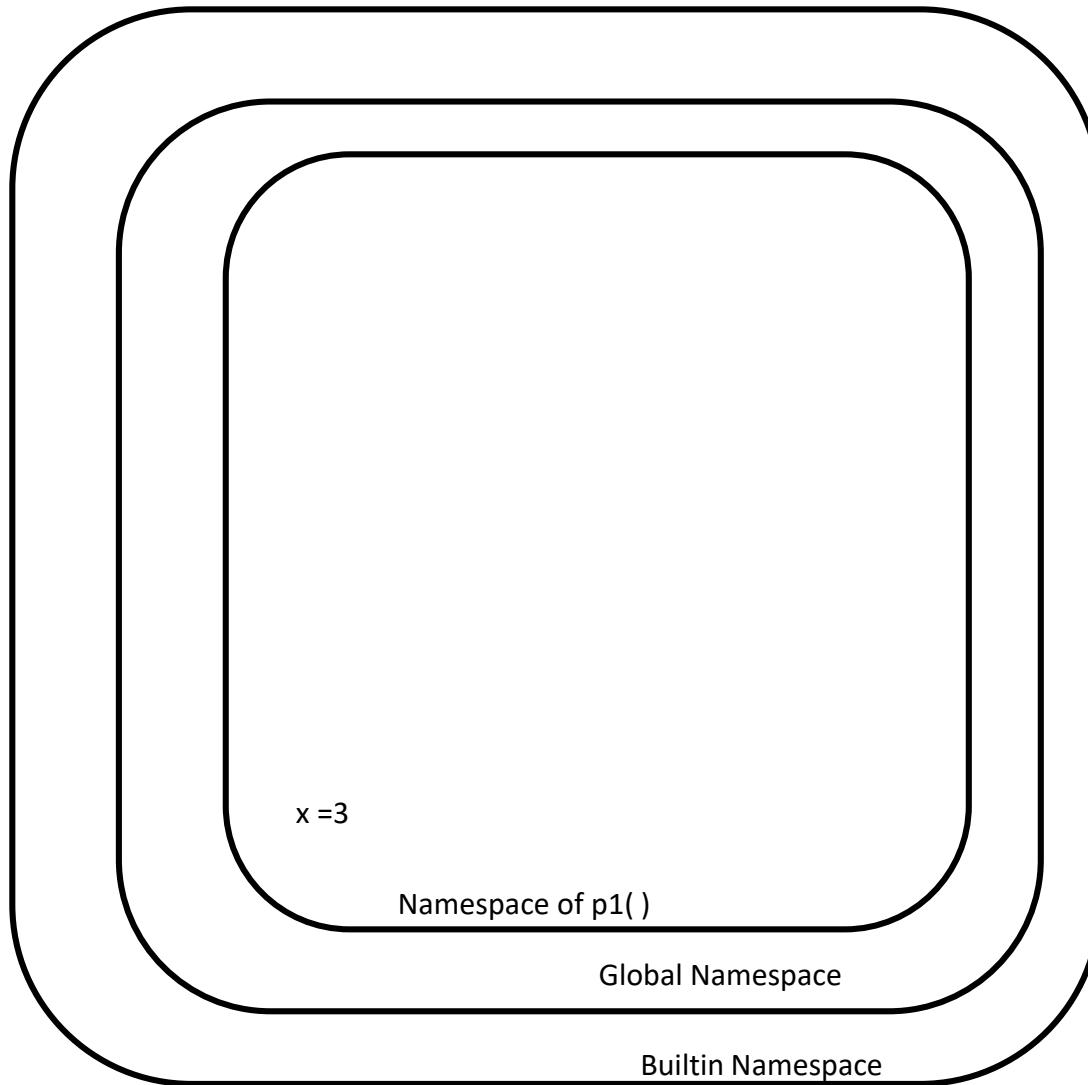


```
print(p1(3))
```

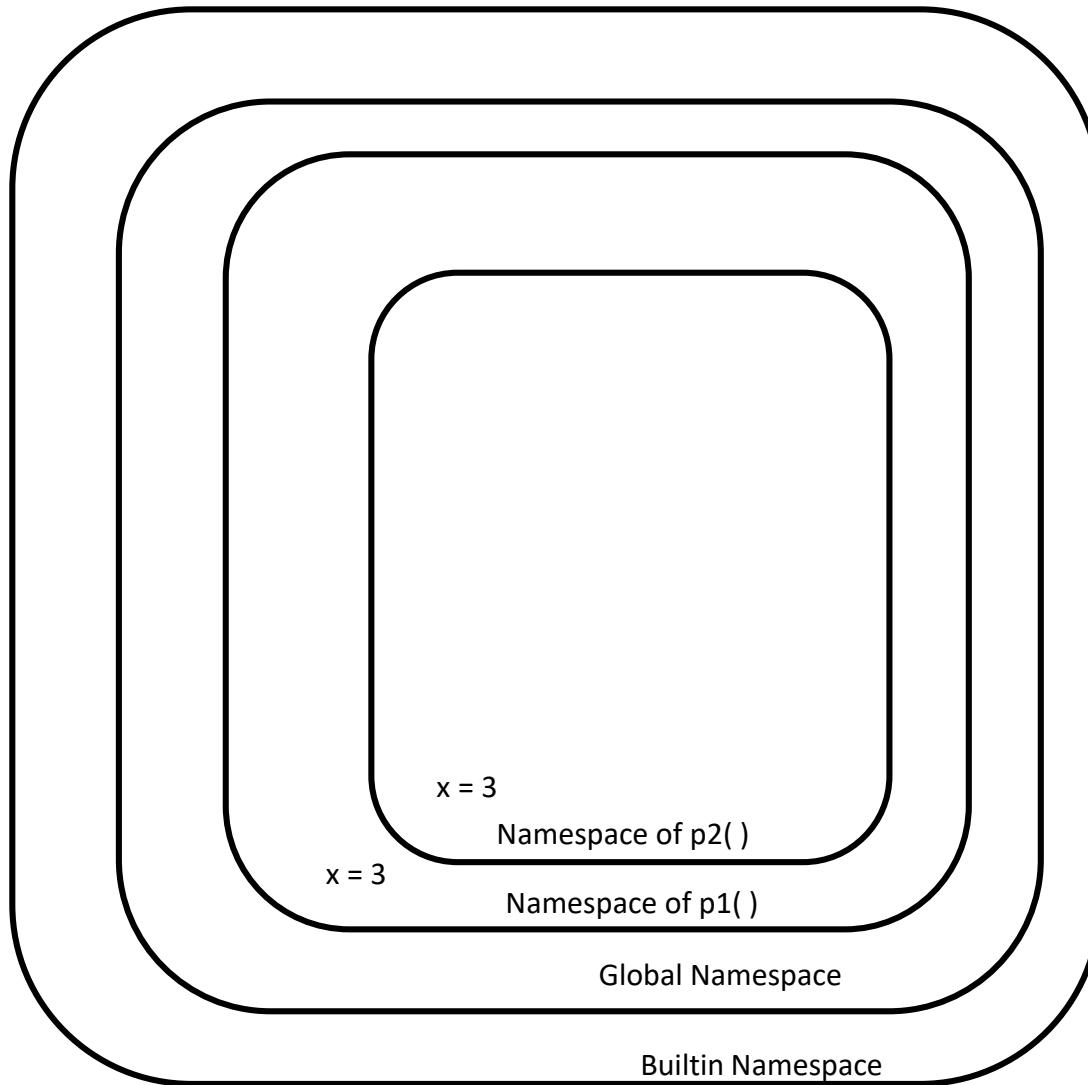
Namespaces: Nested Functions



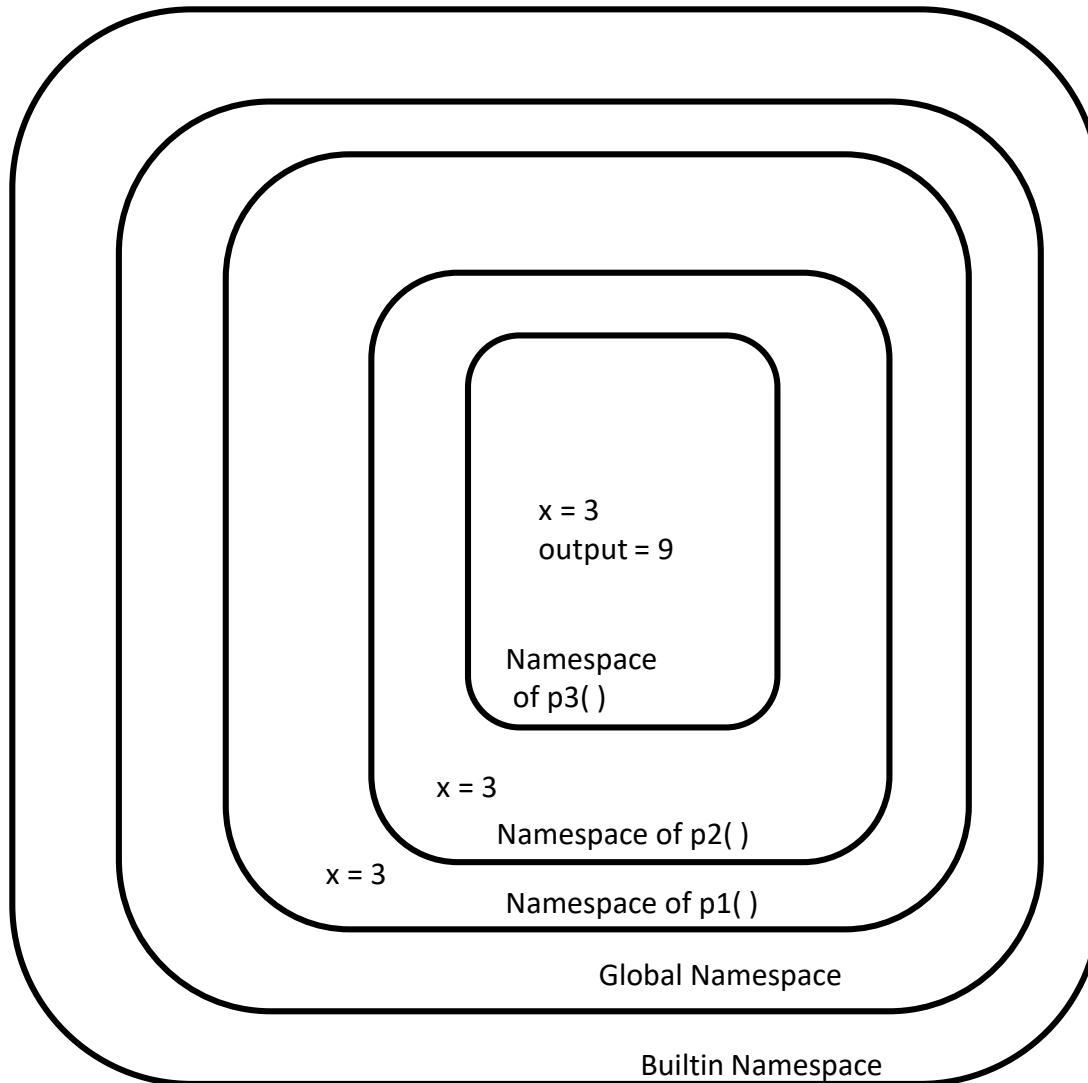
Namespaces: Nested Functions



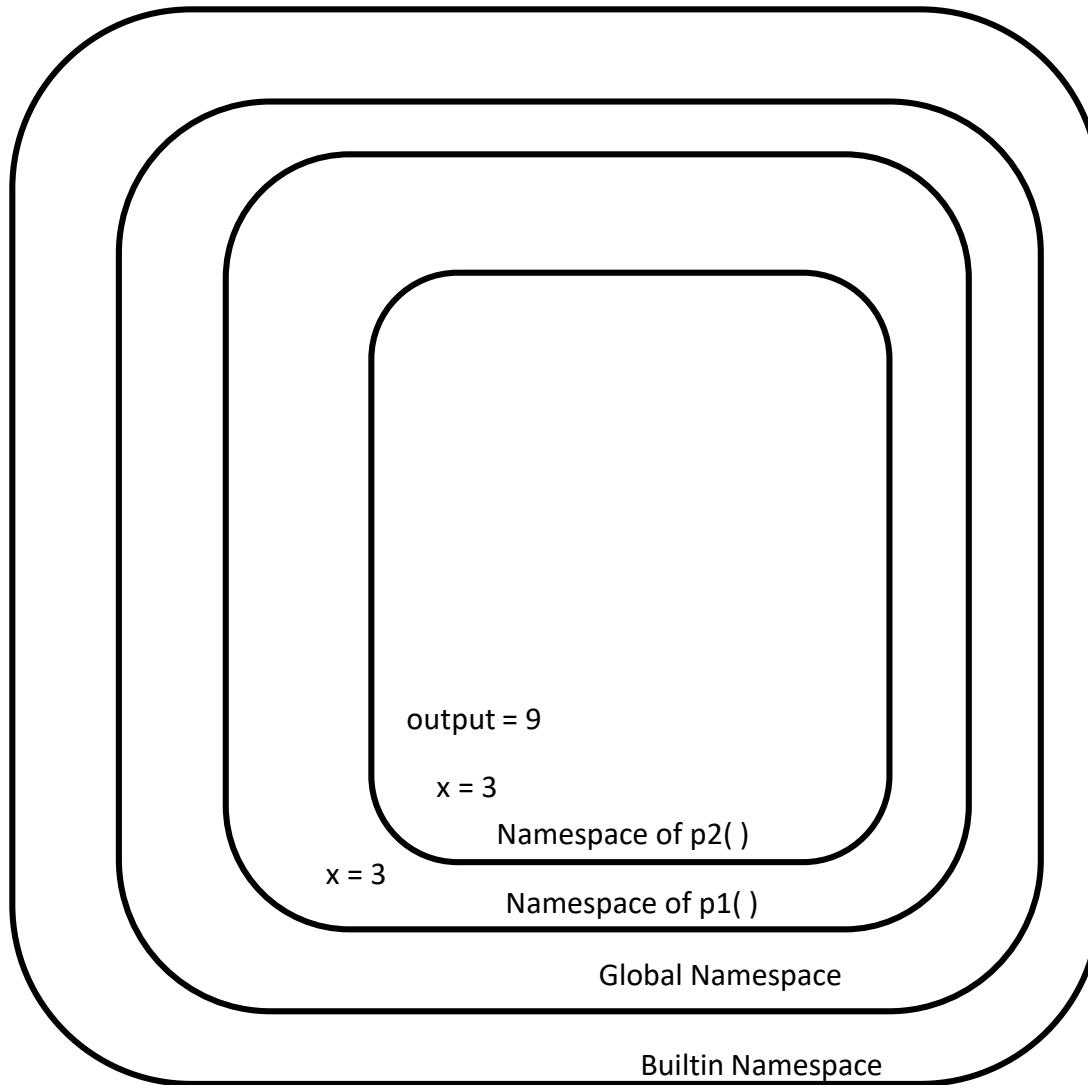
Namespaces: Nested Functions



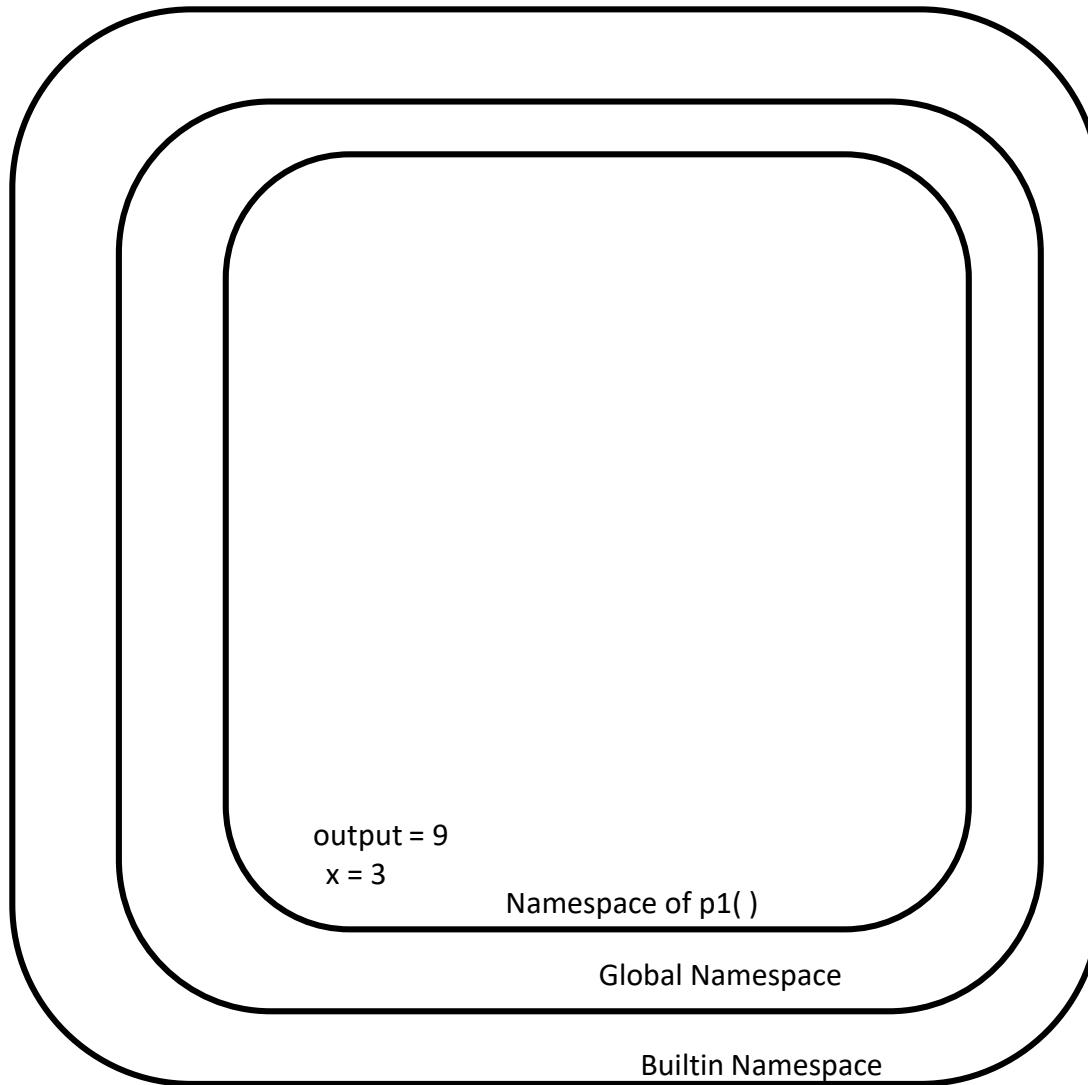
Namespaces: Nested Functions



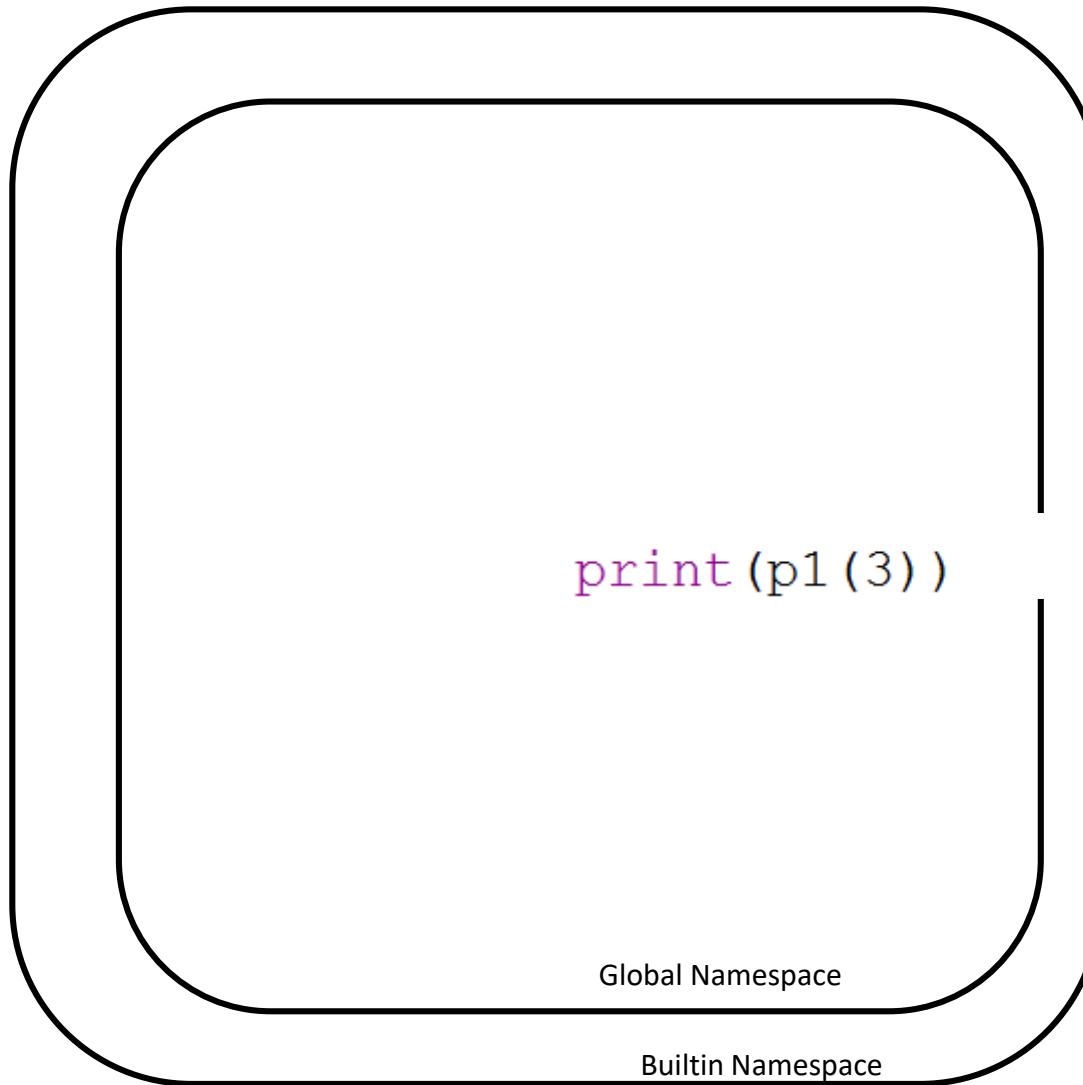
Namespaces: Nested Functions



Namespaces: Nested Functions



Namespaces: Nested Functions



What is the output?

```
def p1(x):
    y = 2
    print('Entering p1')
    def p2(x):
        print('Entering p2')
        z = 4
        def p3(x):
            print('Entering p2')
            output = x**2 +y**2+z**2
            print('Leaving p3')
            return output
        output = p3(x)
        print('Leaving p2')
        return output
    output = p2(x)
    print('Leaving p1')
    return output

print(p1(3))
```

What is the output?

```
def p1(x):
    y = 2
    print('Entering p1')
    def p2(x):
        print('Entering p2')
        z = 4
        def p3(x):
            print('Entering p2')
            output = x**2
            print('Leaving p3')
            return output
        output = p3(x)
        print('Leaving p2')
        return output
    output = p2(x)+z**2
    print('Leaving p1')
    return output

print(p1(3))
```

Where do we use inner functions?

- Higher-Order Functions (Week 5)

Bugs and debugging



The Very First Obstacle of Programming

- Syntax Error

- A syntax error is an error in the source code of a program. Since computer programs **must follow strict syntax** to compile correctly, any aspects of the code that do not conform to the syntax of the programming language will produce a syntax error.

```
>>> x = 10  
SyntaxError: invalid syntax  
>>> |
```



Sometime Errors are Fatal

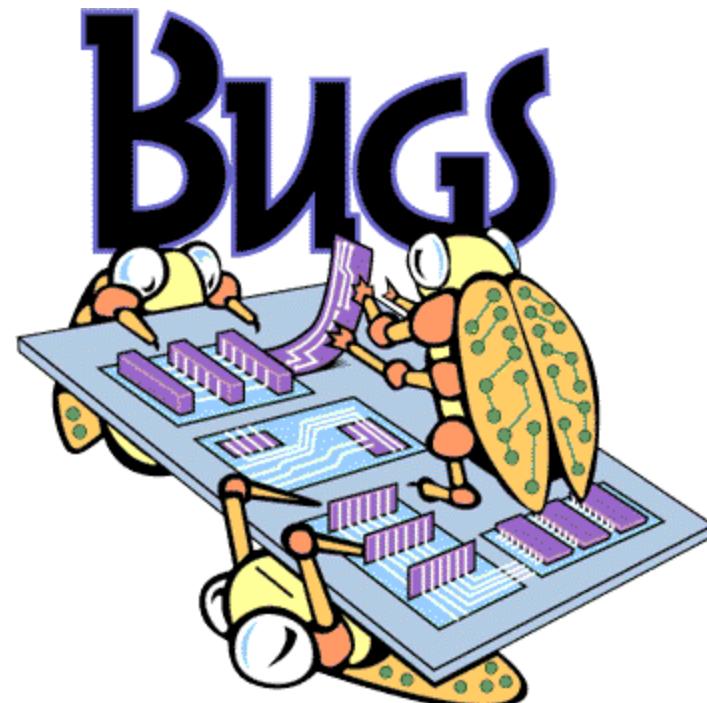
- <https://www.youtube.com/watch?v=VjJgiDuHIRw>



In 1962, a programmer omitted a single hyphen in the code for the Mariner I rocket causing it to explode shortly after take off. This typo cost NASA the today's equivalent of \$630 million dollars.

Bugs?

- In 1947, Grace Murray Hopper was working on the Harvard University Mark II Aiken Relay Calculator (a primitive computer).
- On the 9th of September, 1947, when the machine was experiencing problems, an investigation showed that there was a moth trapped between the points of Relay #70, in Panel F.



Mark I



Bugs?

- The operators removed the moth and affixed it to the log. (See the picture.) The entry reads: "First actual case of bug being found."

9/9	
0800	Autan started
1000	stopped - autan ✓ 13° C (032) MP-MC (033) PRO 2 Relays 6-2 in 033 failed special speed test in relay 11,000 test.
	{ 1.2700 9.037 847 025 9.037 846 995 correct 2.130476415 2.130676415
1100	Started Cosine Tape (Sine check)
1525	Started Multi Adder Test.
1545	 Relay #70 Panel F (moth) in relay.
1630	First actual case of bug being found.
1700	autan starts. closed down.

Humans make mistakes

You are only human

Therefore, you will make mistakes

Debugging

Debugging

- Means to remove errors (“bugs”) from a program.
- After debugging, the program is not necessarily error-free.
 - It just means that whatever errors remain are harder to find.
 - This is especially true for large applications.

W02 debug1.py

```
Python 3.6.0 Shell
File Edit Shell Debug Options Window Help
Python 3.6.0 (v3.6.0:41df79263a11, Dec
v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license"
n.
>>>
RESTART: C:\Users\dcschl\Google Drive\Courses\IT1007\W
02 debug 1.py
>>> p1(1,2)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    p1(1,2)
  File "C:\Users\dcschl\Google Drive\Courses\IT1007\Lectures\W
02 debug 1.py", line 3, in p1
    b = p3(x,y)
  File "C:\Users\dcschl\Google Drive\Courses\IT1007\Lectures\W
02 debug 1.py", line 10, in p3
    return p2(a) + p2(b)
TypeError: p2() missing 1 required positional argument: 'w'
>>> |
```

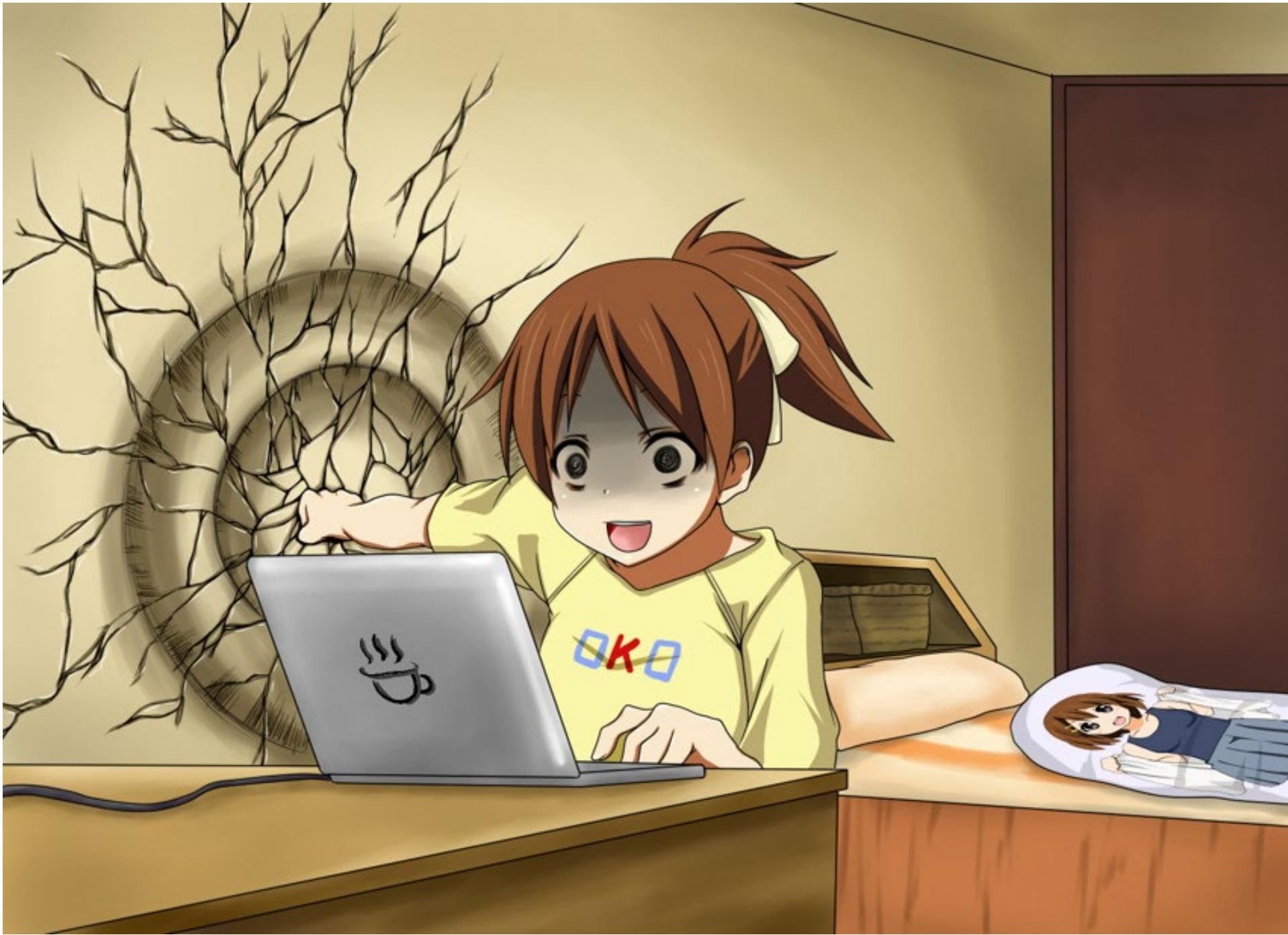


```
W02 debug 1.py - C:\Users\dcsch... - □ ×
File Edit Format Run Options Window Help
def p1(x, y):
    a = p2(x,y)
    b = p3(x,y)
    return a + b

def p2(z, w):
    return z * w

def p3(a, b):
    return p2(a) + p2(b)
```

Ln: 1 Col: 0





KEEP
CALM
AND
DON'T
PANIC

```

>>> p1(1,2) 1
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    p1(1,2)
  File "C:\Users\dcschl\Google Drive\Courses\IT1007\W02\W02 debug 1.py", line 3, in p1
    b = p3(x,y)
  File "C:\Users\dcschl\Google Drive\Courses\IT1007\W02\W02 debug 1.py", line 10, in p3
    return p2(a) + p2(b)
TypeError: p2() missing 1 required positional argument: 'w'
>>> |

```

```

File Edit Format Run Options Window Help
def p1(x, y):
    a = p2(x, y)
    b = p3(x, y) 2
    return a + b

def p2(z, w): ←
    return z * w

def p3(a, b): 3
    return p2(a) + p2(b) 4

Ln: 1 Col: 0

```

Traceback (most recent call last):

- 1 File "<pyshell#0>", line 1, in <module>
- 2 p1(1,2)
- 3 File "C:\Users\dcschl\Google Drive\Courses\IT1007\W02\W02 debug 1.py", line 3, in p1
b = p3(x,y)
- 4 File "C:\Users\dcschl\Google Drive\Courses\IT1007\W02\W02 debug 1.py", line 10, in p3
return p2(a) + p2(b)
- 5 TypeError: p2() missing 1 required positional argument: 'w'



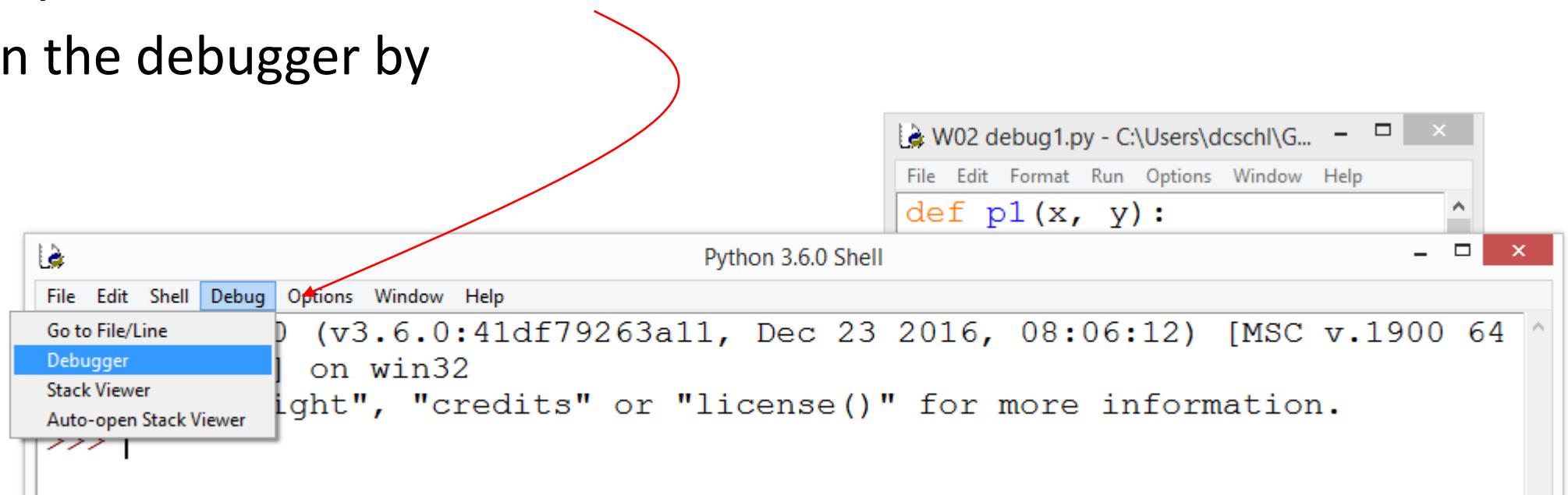


The IDLE Debugger

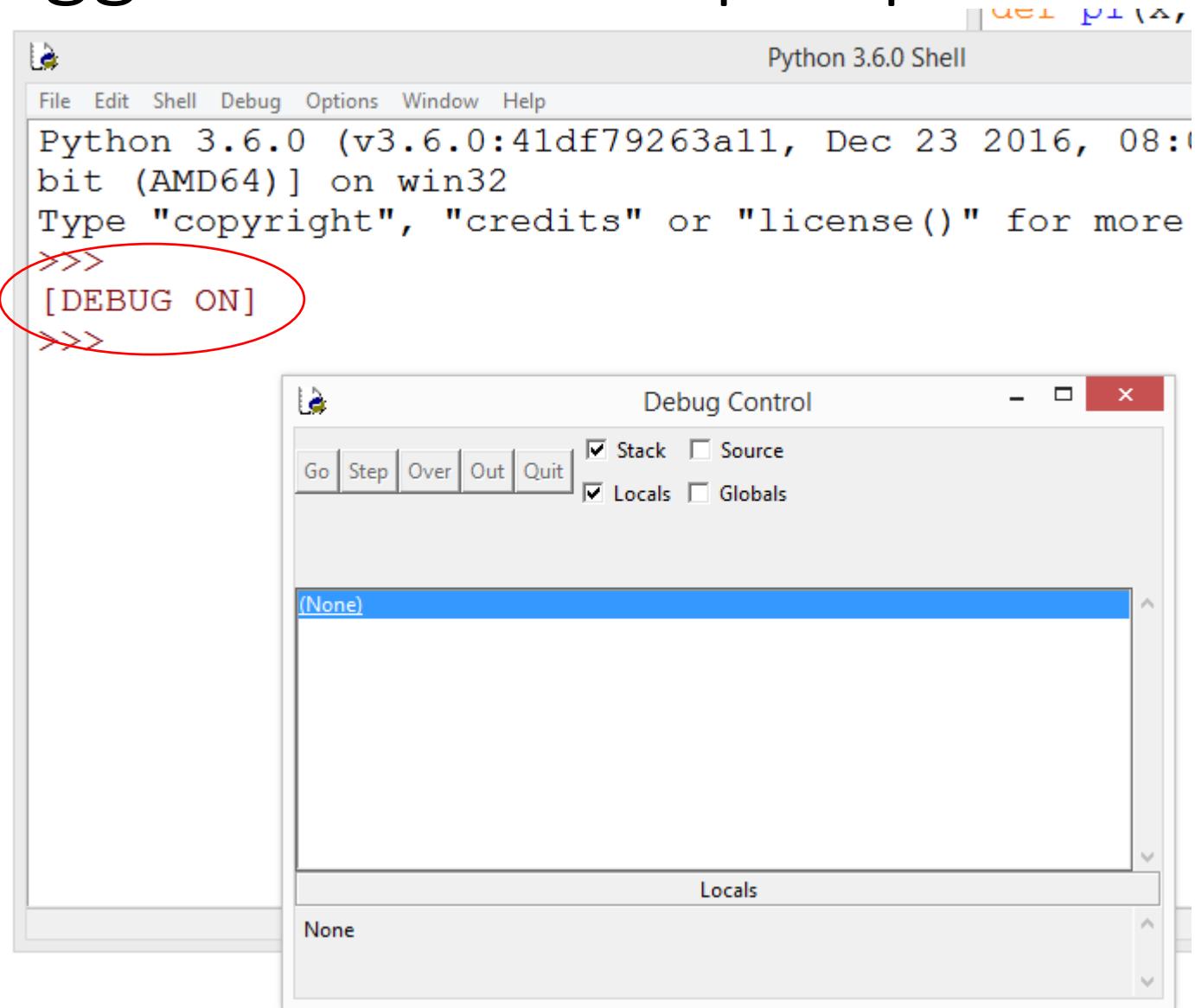


Using the IDLE Debugger

- Load in your source code
- Turn on the debugger by

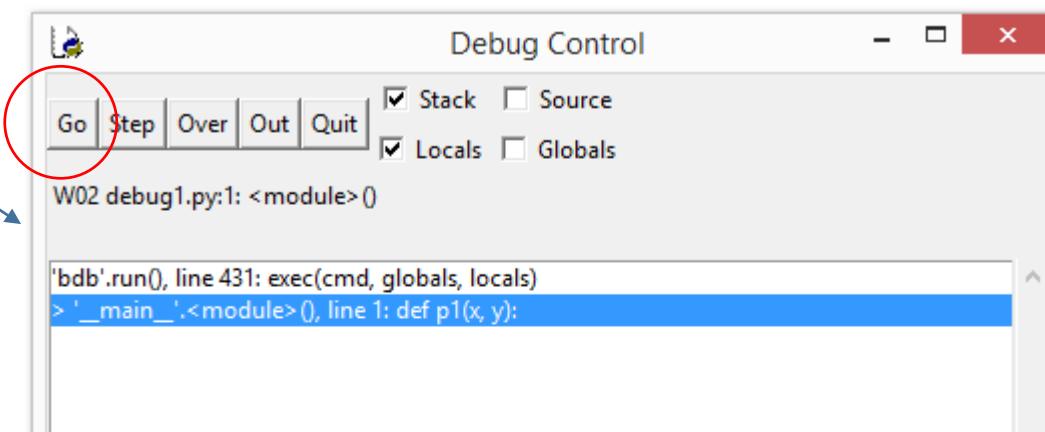


The Debugger Window Pops up



Using the IDLE Debugger

- Go to your source code window to “run”
- Then the debugger will pause the program at the first line of code and wait for you
- You can click the button “Go”
 - That will make the program run
 - At this point we don’t have any error
 - Because by “running” the code, we just define the three functions

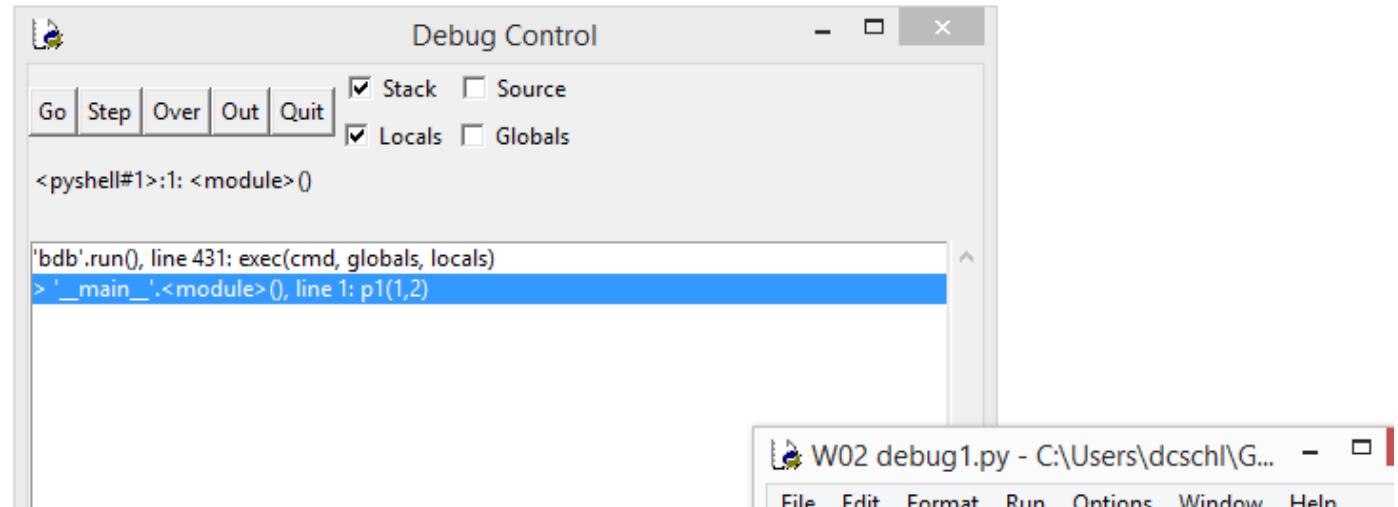


Using the IDLE Debugger

- Let's execute the function in debug mode
- In the shell, type

p1(1,2)

- Then the debugger will pause at the first line of p1
 - If you type “go” now, you will get an error like the last time



Using the IDLE Debugger

- Go
 - Clicking this will run the program until the next break point is reached. You can insert break points in your code by right clicking and selecting Set Breakpoint. Lines that have break points set on them will be highlighted in yellow.
- Step
 - This executes the next statement. If the statement is a function call, it will enter the function and stop at the first line.
- Over
 - This executes the next statement just as Step does. But it does not enter into functions. Instead, it finishes executing any function in the statement and stops at the next statement in the same scope.
- Out
 - This exits the current function and stops in the caller of the current function.
 - After using Step to step into a function, you can use Out to quickly execute all the statements in the function and get back out to the outer function.
- Quit: This terminates execution.

Using the IDLE Debugger

- Currently in line 1
- Click “Step” goes to line 2

The screenshot shows the IDLE debugger interface with two windows. The top window is titled "Debug Control" and displays a stack trace:

```
<pyshell#1>:1: <module>0
'bdb'.run(), line 431: exec(cmd, globals, locals)
> '_main_'.<module>0, line 1: p1(1,2)
```

The bottom window is titled "W02 debug 1.py - C:\Users\dcschl\Google Drive\Courses\IT1007\Lectu..." and contains the source code:

```
def p1(x, y):
    a = p2(x, y)
    b = p3(x, y)
    return a + b

def p2(z, w):
    return z * w

def p3(a, b):
    return p2(a) + p2(b)
```

The status bar at the bottom right indicates "Ln: 1 Col: 0". Below the source code, the variable definitions are shown:

p2	<function p2 at 0x0000007D47B82020>
p3	<function p3 at 0x0000007D47B826A8>

Using the IDLE Debugger

Current position

Step: Go into functions, otherwise “over”

Out: run until
the current
function ends

Over: run until next line

```
def p1(x, y):
    a = p2(x, y)
    b = p3(x, y)
    return a + b

def p2(z, w):
    return z * w

def p3(a, b):
    return p2(a) + p2(b)
```

More Debugging (BuggyAddNum)

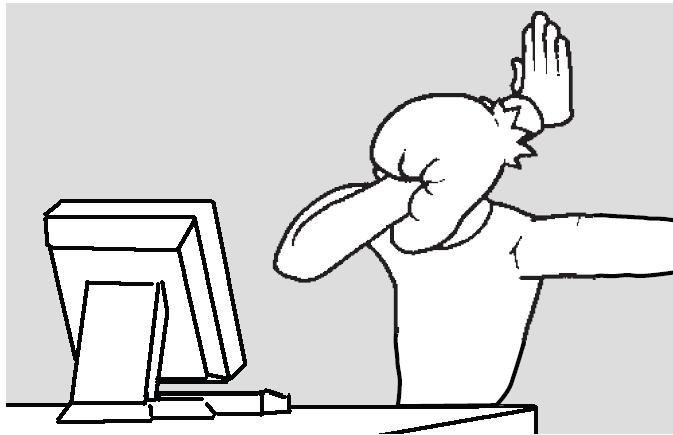
```
import random

def add2Num():
    number1 = random.randint(1, 10)
    number2 = random.randint(1, 10)

    print('What is ' + str(number1) + ' + ' + str(number2) + '?')
    answer = input()
    if answer == number1 + number2:
        print('Correct!')
    else:
        print('Nope! The answer is ' + str(number1 + number2))
```

```
>>> add2Num()
What is 4 + 9?
10
Nope! The answer is 13
>>> |
```

```
>>> add2Num()
What is 6 + 5?
11
Nope! The answer is 11
>>> add2Num()
What is 5 + 9?
14
Nope! The answer is 14
>>> |
```



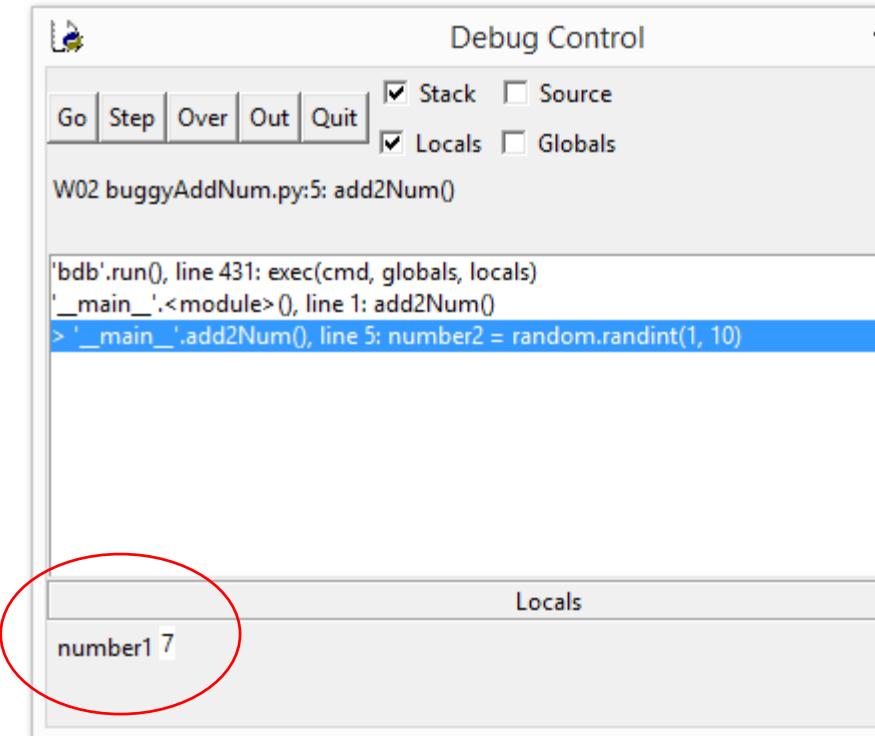
```
import random

def add2Num():
    number1 = random.randint(1, 10)
    number2 = random.randint(1, 10)

    print('What is ' + str(number1) + ' + ' + str(number2) + '?')
    answer = input()
    if answer == number1 + number2:
        print('Correct!')
    else:
        print('Nope! The answer is ' + str(number1 + number2))
```

Turn on Debugger

- After a few steps



The screenshot shows a "Debug Control" window with the following interface elements:

- Buttons: Go, Step, Over, Out, Quit.
- Checkboxes: Stack (checked), Source (unchecked), Locals (checked), Globals (unchecked).
- Text area: W02 buggyAddNum.py:5: add2Num()
- Stack trace:

```
'bdb'.run(), line 431: exec(cmd, globals, locals)
'_main_'.<module>, line 1: add2Num()
> '_main_'.add2Num(), line 5: number2 = random.randint(1, 10)
```
- Locals table:

number1	7
number1	7

A red circle highlights the value "7" in the "number1" row of the locals table.





Debug Control

Go Step Over Out Quit Stack Source
 Locals Globals

W02 buggyAddNum.py:7: add2Num()

```
'bdb'.run(), line 431: exec(cmd, globals, locals)
'_main_'.<module>(), line 1: add2Num()
> '_main_'.add2Num(), line 7: print('What is ' + str(number1) + ' + ' + str(number2) + '?')
```

Locals	
number1	7
number2	1

```
>>> add2Num()
What is 7 + 1?
8|
```

Debug Control

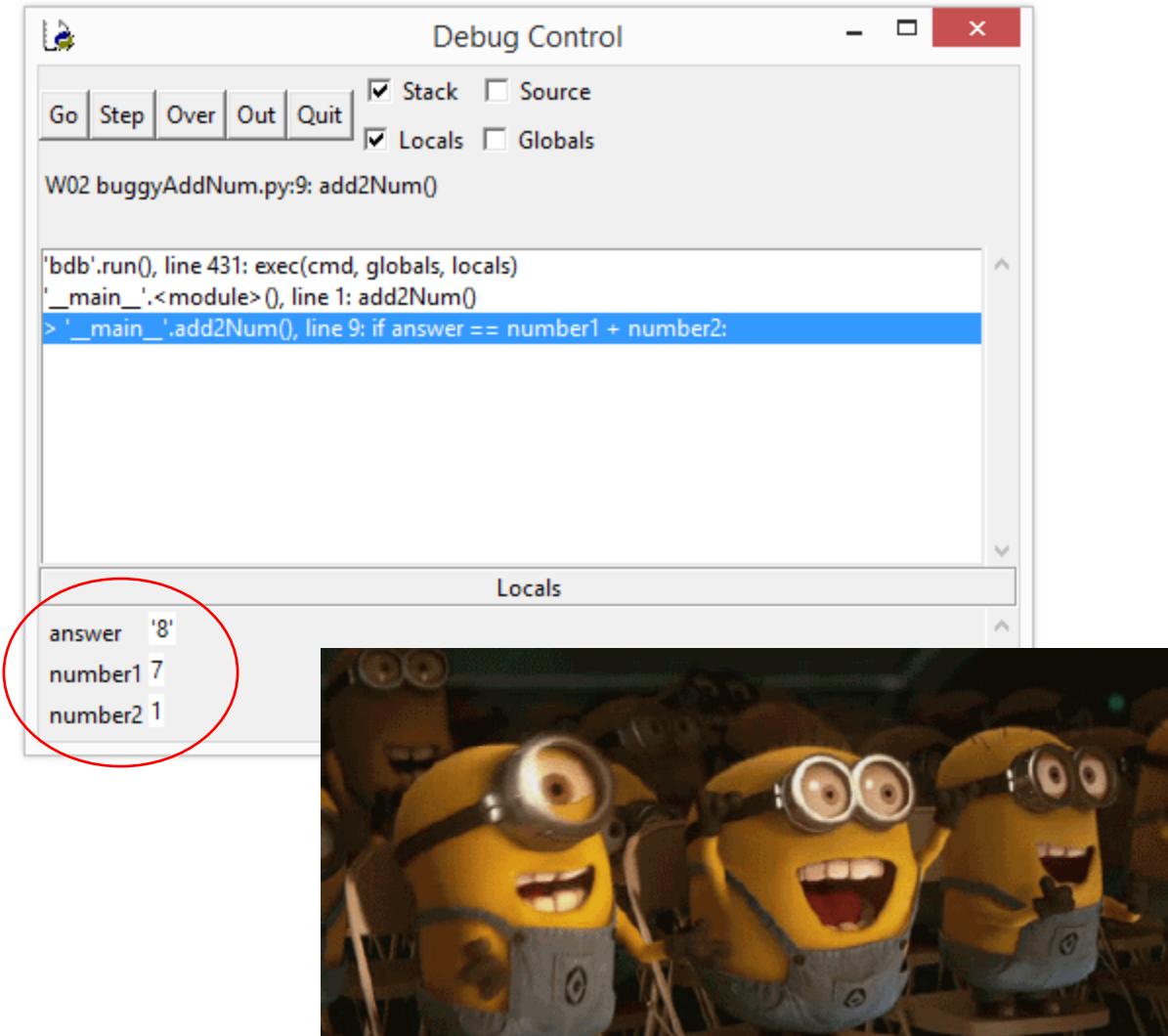
Go Step Over Out Quit Stack Source
 Locals Globals

```
'bdb'.run(), line 431: exec(cmd, globals, locals)
'_main_'.<module>(), line 1: add2Num()
> '_main_'.add2Num(), line 8: answer = input()
```

Locals

```
number1 7
number2 1
```

Using the IDLE Debugger



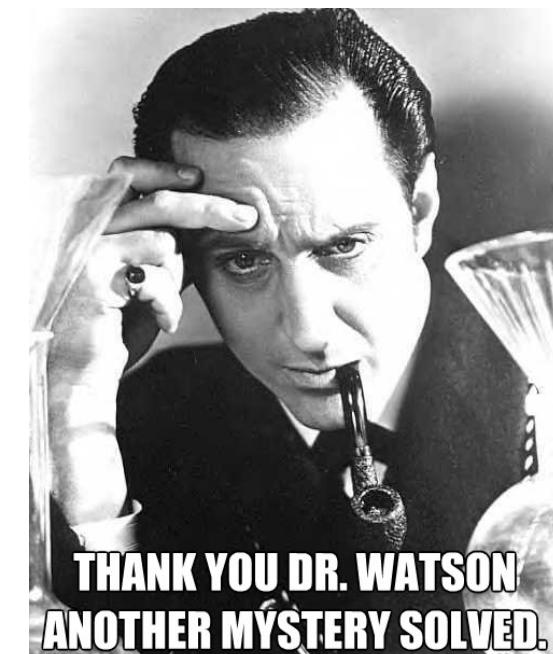
Debug Control

W02 buggyAddNum.py:9: add2Num()

```
'bdb'.run(), line 431: exec(cmd, globals, locals)
'_main_'.<module>(), line 1: add2Num()
> '_main_'.add2Num(), line 9: if answer == number1 + number2:
```

Locals

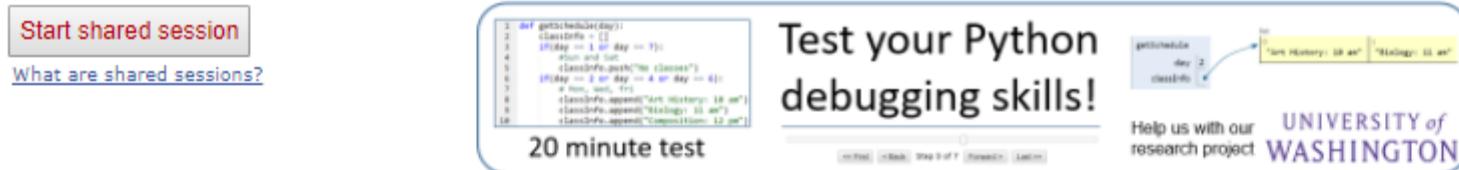
answer	'8'
number1	7
number2	1



THANK YOU DR. WATSON
ANOTHER MYSTERY SOLVED.

giphy.com

Another Debugger: pythontutor.com



Write code in Python 3.6 ▾

```
1 def p1(x, y):
2     a = p2(x,y)
3     b = p3(x,y)
4     return a + b
5
6 def p2(z, w):
7     return z * w
8
9 def p3(a, b):
10    return p2(a) + p2(b)
11
12 p1(1,2)
```

Support our research and practice Python by trying our new [debugging skill test!](#)

[Start shared session](#)

[What are shared sessions?](#)

```
2 def getClassDef(day):
3     classDef = []
4     if day == 1 or day == 7:
5         return classDef
6     else:
7         print("No classes")
8     if day == 2 or day == 4 or day == 6:
9         classDef.append("Art History 10 am")
10    classDef.append("Writing 10 am")
11    classDef.append("Composition 12 pm")
```

Test your Python debugging skills!

20 minute test

pete@pete: ~ % history 10 am % "Writing 10 am"

Help us with our
research project
**UNIVERSITY of
WASHINGTON**

Python 3.6

```
1 def p1(x, y):
2     a = p2(x,y)
3     b = p3(x,y)
4     return a + b
5
6 def p2(z, w):
7     return z * w
8
9 def p3(a, b):
10    return p2(a) + p2(b)
11
12 p1(1,2)
```

[Edit code](#) | [Live programming](#)

→ line that has just executed

→ next line to execute

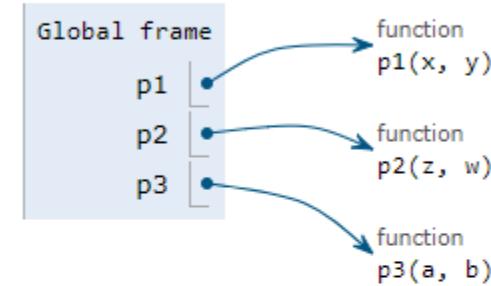
Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

[<< First](#) [< Back](#) Program terminated [Forward >](#) [Last >>](#)

TypeError: p2() missing 1 required positional argument: 'w'

Visualized using [Python Tutor](#) by [Philip Guo \(@pgbovine\)](#)

Frames Objects



Common Types of Errors

Common Types of Errors

- Omitting return statement

```
def square(x):  
    x * x      # no error msg!
```

- Incompatible types

```
x = 5  
def square(x):  
    return x * x  
x + square
```

- Incorrect # args

```
square(3,5)
```

Common Types of Errors

- Syntax

```
def proc(100)
    do_stuff()
    more()
```

- Arithmetic error

```
x = 3
y = 0
x/y
```

- Undeclared variables

```
x = 2
x + k
```

Common Types of Errors

- Infinite loop (from bad inputs)

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        return n * factorial(n-1)
```

fact(2.1)

fact(-1)

Common Types of Errors

- Infinite loop (forgot to decrement)

```
def fact_iter(n):
    counter, result = n, 1
    while counter != 0:
        result *= counter
    return result
```

Common Types of Errors

- Numerical imprecision

```
def foo(n):  
    counter, result = 0,0  
    while counter != n:  
        result += counter  
        counter += 0.1  
    return result
```

```
foo(5)
```

counter never exactly equals n



Common Types of Errors

- Logic

```
def fib(n):  
    if n < 2:  
        return n  
    else:  
        return fib(n-1) + fib(n-1)
```

How to debug?

- Think like a **detective**
 - Look at the clues: error messages, variable values.
 - Eliminate the impossible.
 - Run the program again with different inputs.
 - Does the same error occur again?

How to debug?

- Work backwards
 - From current sub-problem backwards in time
- Use a debugger
 - IDLE has a simple debugger
 - Overkill for our class
- Trace a function
- Display variable values

Displaying variables

```
debug_printing = True
def debug_print(msg):
    if debug_printing:
        print(msg)

def foo(n):
    counter, result = 0,0
    while(counter != n):
        debug_print(f'{counter}, {n}, {result}')
        counter, result = counter + 0.1, result + counter
    return result
```

Example

```
def fib(n):  
    debug_print(f'n:{n}')  
    if n < 2:  
        return n  
    else:  
        return fib(n-1) + fib(n-1)
```

Other tips

- State assumptions clearly.

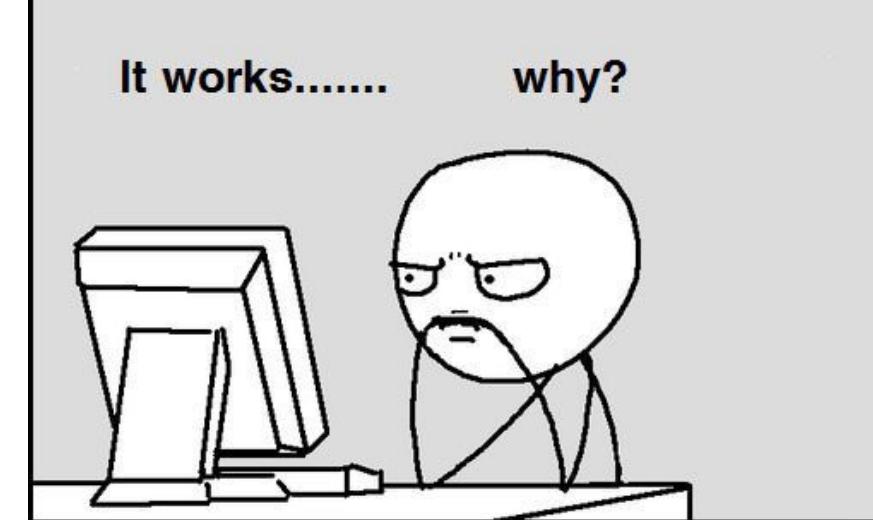
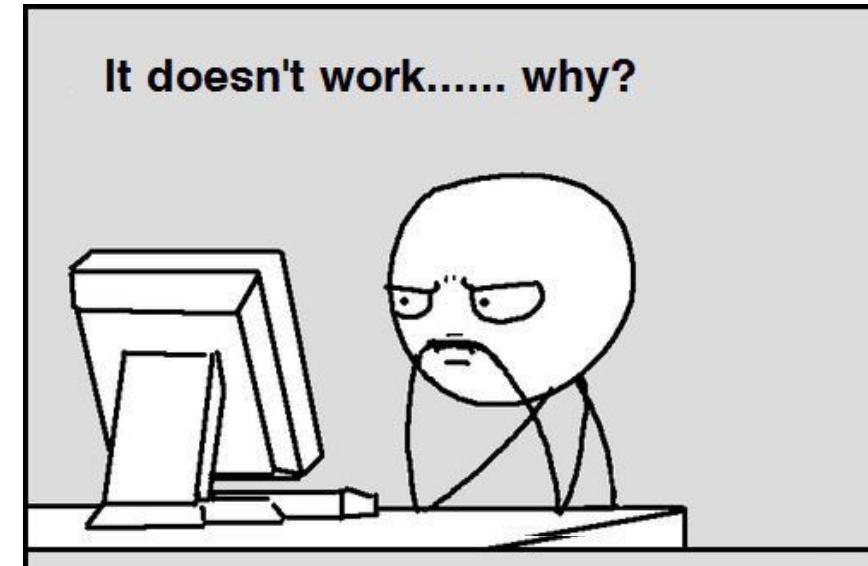
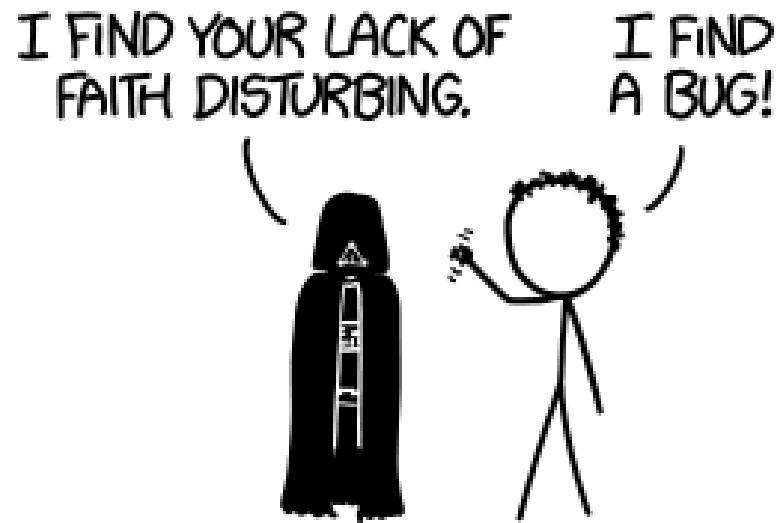
```
def factorial(n): # n integer >= 0
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

- Test each function before you proceed to the next.
 - Remember to test boundary cases

Summary

- Compound data helps us to reason at a higher conceptual level.
- Abstraction barriers separate usage of a compound data from its implementation.
- Only functions at the interface should be used.
- We can choose between different implementations as long as contract is fulfilled.

Debugging is an Art



Maths vs CS vs Engineering

- Three good friends, an engineer, a mathematician and a computer scientist, are driving on a highway that is in the middle of no where. Suddenly one of the tires went flat and they have no spare tire.



Maths vs CS vs Engineering

- Engineer
 - “Let’s use bubble gum to patch the tire and use the strew to inflate it again”
- Mathematician
 - “I can prove that there is a good tire exists in somewhere this continent”
- Computer Scientist
 - “Let’s remove the tire, put it back, and see if it can fix itself again”



Data Collections (Sequences)

It's complicated

Sequence in Python

- Indexed collection
 - Strings
 - Lists
 - Tuples
- Non-indexed collection:
 - Sets
 - Dictionary

Lists and Tuples

- Belongs to a type of data structure called arrays
- Intrinsic ordering
 - Easy to retrieve data from the array
- Lists
 - Mutable
 - Dynamic Arrays
- Tuples
 - Immutable
 - Static Arrays

Lists

- Ordered sequence of data types

- Homogeneous sequence

- Sequence of integers
 - Sequence of floats
 - Sequence of strings
 - Sequence of lists
 - Sequence of functions, etc.

- Heterogeneous sequence

- mix of integers, floats, strings, etc.

```
>>> empty_list = []
>>> empty_list
[]
>>> int_list = [1,2,3,4]
>>> int_list
[1, 2, 3, 4]
>>> float_list = [1.0,2.0,3.0,4.0]
>>> float_list
[1.0, 2.0, 3.0, 4.0]
>>> string_list = ['Hello!', 'Welcome', 'to', 'IT5001']
>>> string_list
['Hello!', 'Welcome', 'to', 'IT5001']
>>> heterogeneous_list = [1,2.0,'Hello', 3+4j]
>>> heterogeneous_list
[1, 2.0, 'Hello', (3+4j)]
```

- Defined using square brackets - []

Lists are Referential Arrays

Python 3.6
[\(known limitations\)](#)

```
1 integer_list = [1,2,3]
2
3 float_list = [1.0, 2.0, 3.0]
4
5 str_list = ['Welcome', 'to', 'IT5001']
```

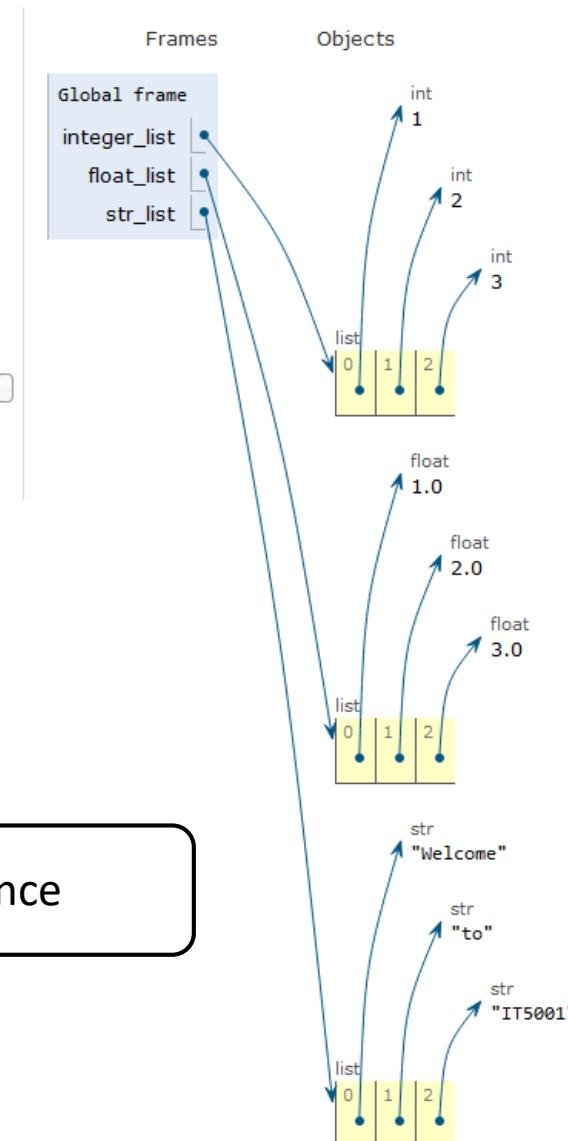
[Edit this code](#)

Just executed
to execute

<< First < Prev Next > >>

Done running (3 steps)

[Initialization](#)



Lists store addresses of its items in sequence

All Indexed Sequences can...

a[i]	return i-th element of a
a[i:j]	returns elements i up to j-1
len(a)	returns numbers of elements in sequence
min(a)	returns smallest value in sequence
max(a)	returns largest value in sequence
x in a	returns True if x is a part of a
a + b	concatenates a and b
n * a	creates n copies of sequence a

```
>>> int_list = [1,2,3,4]
>>> print(f'Element at index = 1 is {int_list[1]}')
Element at index = 1 is 2
>>> print(f'Elements at index = [1,3) are {int_list[1:3]}')
Elements at index = [1,3) are [2, 3]
>>> print(f'Number of Elements = {len(int_list)}')
Number of Elements = 4
>>> print(f'Smallest Element in the List = {min(int_list)}')
Smallest Element in the List = 1
>>> print(f'Largest Element in the List = {max(int_list)}')
Largest Element in the List = 4
>>> print(f'Is element 2 in the list: {2 in int_list}')

Is element 2 in the list: True
>>> another_int_list = [5,6,7,8]
>>> print(f'Concatenated List: {int_list+another_int_list}')
Concatenated List: [1, 2, 3, 4, 5, 6, 7, 8]
>>> print(f'Repeated list: {int_list*3}')
Repeated list: [1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
```

Lists are mutable

- Elements can be replaced
- Elements can be added
- Elements can be removed
 - A specific element
 - If element occurs multiple times, removes first occurrence
 - Element at a specific location (index)
 - From the end of the list
- Elements can be sorted
 - `sort()`
 - `Sorted()`
- Elements can be reversed

Lists are Dynamic-Size Arrays

Elements can be added (appended) to the list

```
>>> id(integer_list)  
1421979130760  
>>> integer_list  
[2, 5, 3, 4]  
>>> integer_list.append(9)  
>>> integer_list  
[2, 5, 3, 4, 9]  
>>> id(integer_list)  
1421979130760
```

Elements can be removed from the list

```
>>> integer_list = [2, 5, 3, 4, 9]  
>>> id(integer_list)  
2378756596296  
>>> integer_list.remove(9)  
>>> integer_list  
[2, 5, 3, 4]  
>>> id(integer_list)  
2378756596296  
>>> my_list = [1, 2, 3, 4, 6, 3]  
>>> my_list.remove(3)  
>>> my_list  
[1, 2, 4, 6, 3]
```

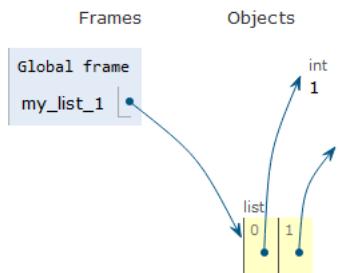
Append Vs Concatenation

Python 3.6
(known limitations)

```
1 my_list_1 = [1,2]
2
3 my_list_2 = my_list_1.append(3)
4
5 my_list_3 = my_list_1 + [3]
```

[Edit this code](#)

st executed
execute



1

Python 3.6
(known limitations)

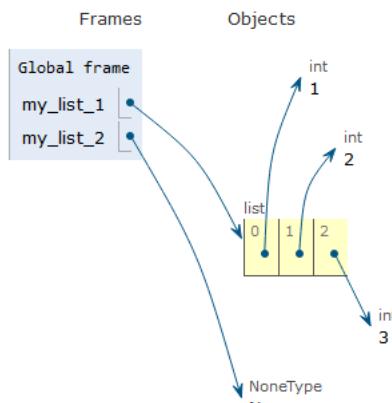
```
1 my_list_1 = [1,2]
2
3 my_list_2 = my_list_1.append(3)
4
5 my_list_3 = my_list_1 + [3]
```

[Edit this code](#)

st executed
execute

<< First < Prev Next > Last >>

Step 3 of 3



2

Append Vs Concatenation

Python 3.6
([known limitations](#))

```
1 my_list_1 = [1,2]
2
3 my_list_2 = my_list_1.append(3)
4
5 my_list_3 = my_list_1 + [3]
```

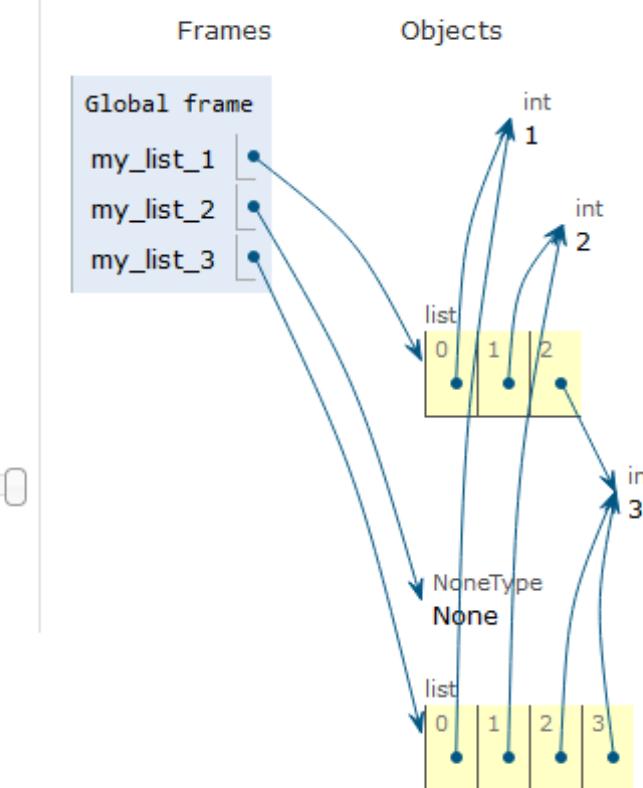
[Edit this code](#)

Just executed
to execute

<< First < Prev Next > >>

Done running (3 steps)

[Initialization](#)



Append Vs Concatenation

```
>>> my_list_1 = [1,2,3,4]
>>> my_list_2 = [5,6,7,8]
>>> my_list_1.append(my_list_2)
>>> my_list_1
[1, 2, 3, 4, [5, 6, 7, 8]]
>>> my_list_1 = my_list_1 + my_list_2
>>> my_list_1
[1, 2, 3, 4, [5, 6, 7, 8], 5, 6, 7, 8]
```

Strings to Lists and Vice-Versa

Python 3.6
([known limitations](#))

```
1 my_string = "IT5001"
2
3 my_list = list(my_string)
```

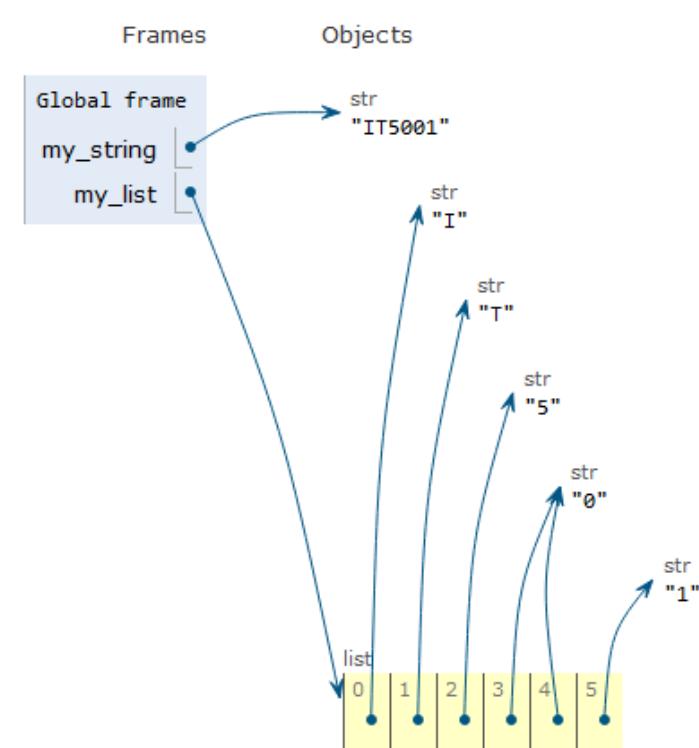
[Edit this code](#)

Just executed
None to execute

<< First < Prev Next > >>

Done running (2 steps)

[visualization](#)



Strings to Lists and Vice-Versa

Python 3.6
[\(known limitations\)](#)

```
1 my_list = ['a','b','c','d']
2 my_string = str(my_list)
→ 3 a = my_string[0]
```

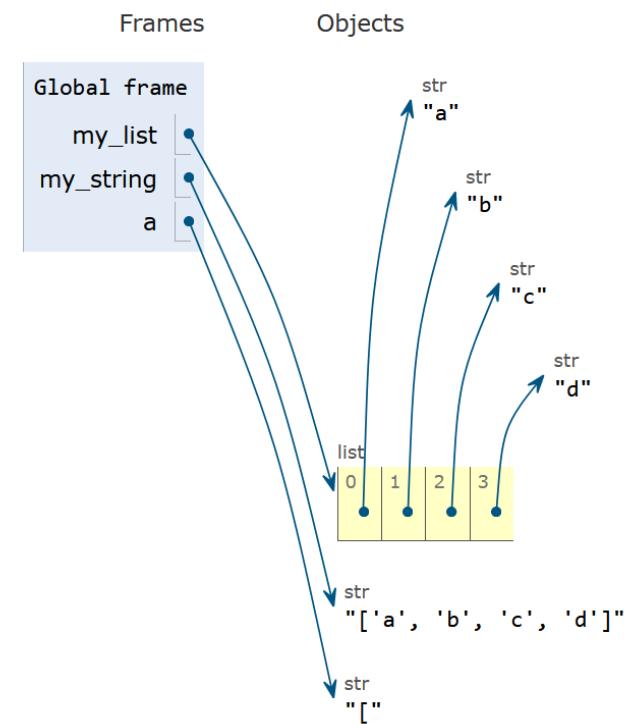
[Edit this code](#)

→ line that just executed
→ next line to execute

<< First < Prev Next > Last >>

Done running (3 steps)

[Customize visualization](#)



Aliasing vs Cloning

Aliasing

Python 3.6
(known limitations)

```
1 integer_list_1= [1,2,3,4]
2
3
4 integer_list_2 = integer_list_1
5
6
7 integer_list_3 = integer_list_1[:]
```

[Edit this code](#)

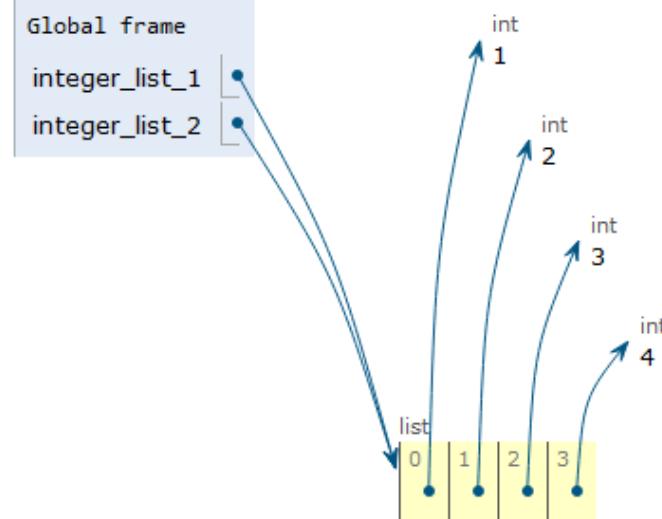
it just executed
ie to execute

[**<< First**](#) [**< Prev**](#) [**Next >**](#) [**Last >>**](#)

Step 3 of 3

Frames

Objects



Aliasing vs Cloning

Python 3.6
([known limitations](#))

```
1 integer_list_1= [1,2,3,4]
2
3
4 integer_list_2 = integer_list_1
5
6
7 integer_list_3 = integer_list_1[:]
```

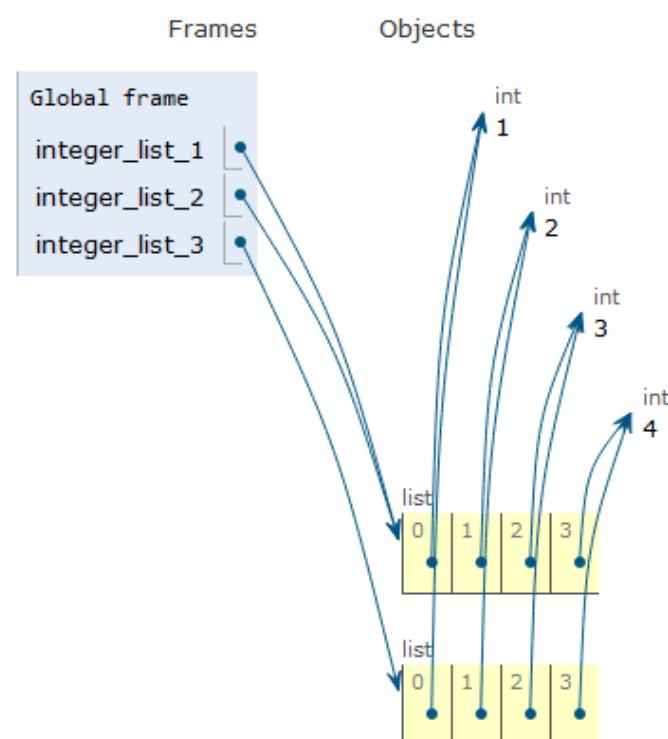
[Edit this code](#)

just executed
to execute

[<< First](#) [< Prev](#) [Next >](#) [Last >>](#)

Done running (3 steps)

Cloning



Sort vs Sorted

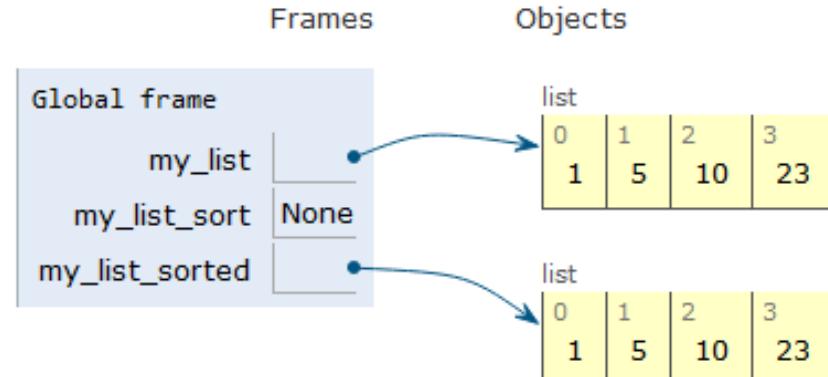
- `sort()` method mutates the list
- `sorted()` method creates a new sorted list without mutating the original list

Python 3.6
(known limitations)

```
1 my_list = [10, 1, 23, 5]
2 my_list_sort = my_list.sort()
3 my_list_sorted = sorted(my_list)
```

[Edit this code](#)

executed
execute



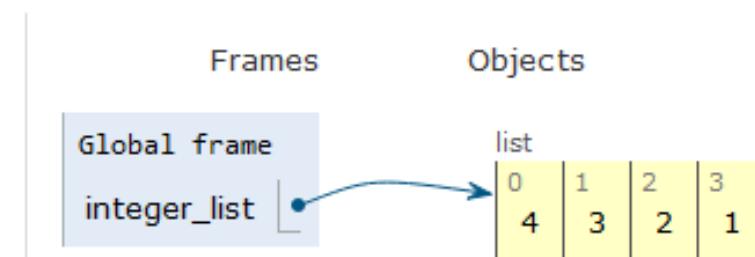
Reverse

- reverse() method mutates the list

Python 3.6
[\(known limitations\)](#)

```
1 integer_list = [1,2,3,4]
2
→ 3 integer_list.reverse()
```

[Edit this code](#)



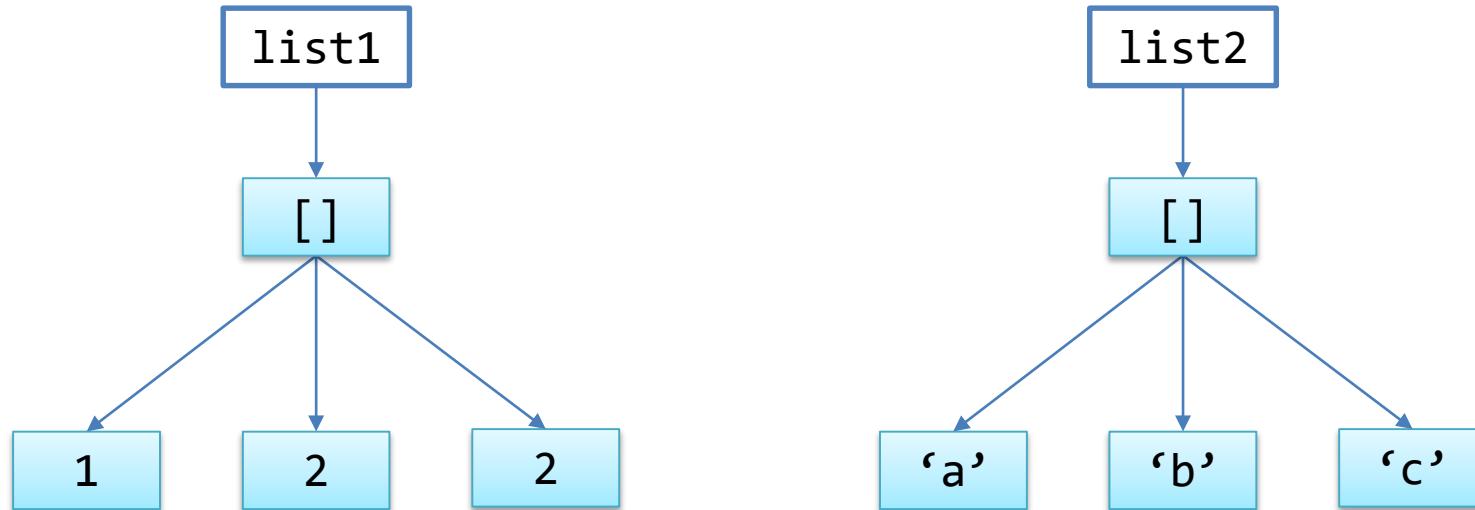
Lists of Anything

- A list of
 - Lists?

```
>>> list1 = [1,2,3]
>>> list2 = ['a','b','c']
>>> list3 = [list1,list2]
>>> list3
[[1, 2, 3], ['a', 'b', 'c']]
>>> list4 = [True,list3,list1]
>>> list4
[True, [[1, 2, 3], ['a', 'b', 'c']], [1, 2, 3]]
```

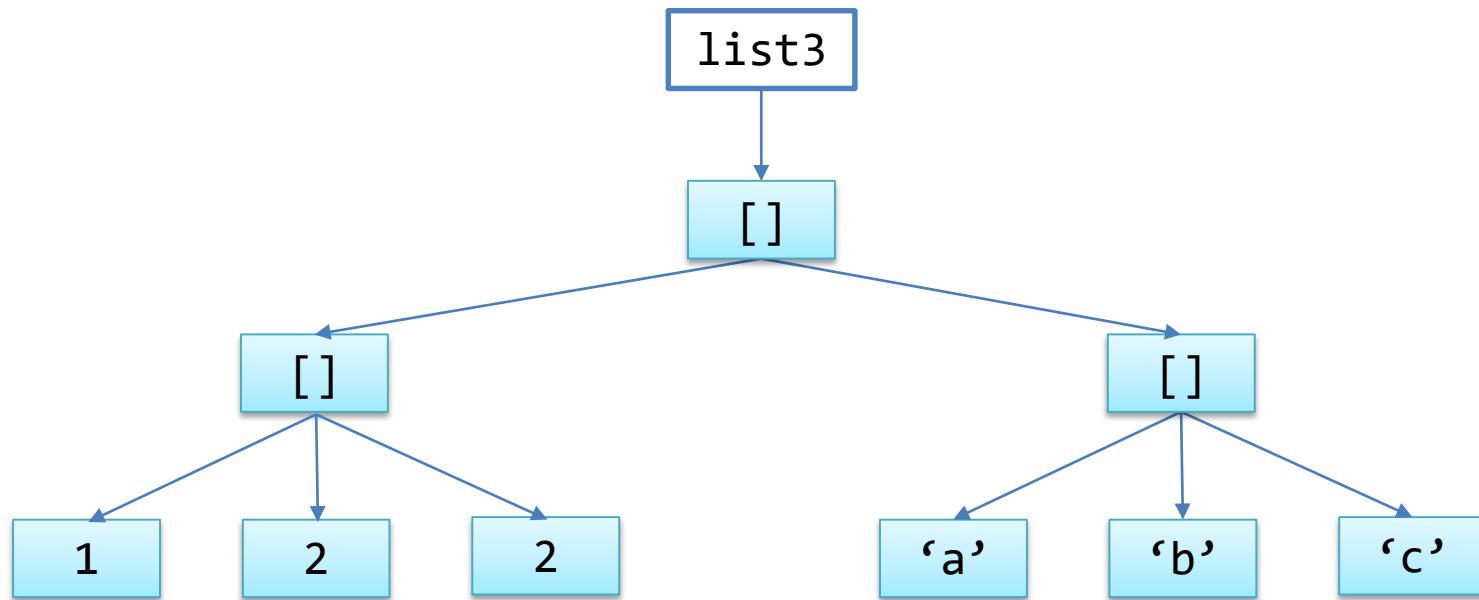
Block Diagram

```
>>> list1 = [1,2,3]  
>>> list2 = ['a','b','c']
```



Block Diagram

```
>>> list3 = [list1,list2]  
>>> list3  
[[1, 2, 3], ['a', 'b', 'c']]
```



Lists are Iterable

due to this method

```
>>> my_list = [1,2,3,4]
>>> dir(my_list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__has__',
 '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
 '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__',
 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
 'remove', 'reverse', 'sort']
```

```
>>> for e in my_list:
    print(e)
```

1
2
3
4

For loop

```
>>> for i in range(0,5):  
    print (i)
```

0
1
2
3
4

```
>>> for i in [0,1,2,3,4]:  
    print(i)
```

0
1
2
3
4

For Loop

```
my_list = [1,2,3,4]
```

```
for i,j in enumerate(my_list):  
    print(f'Element at index {i} is {j}')
```

Output:

```
Element at index 0 is 1  
Element at index 1 is 2  
Element at index 2 is 3  
Element at index 3 is 4
```

Never do this

```
myList = [1, 2, 3, 4]  
  
for ele in myList:  
    myList.remove(ele)
```

Why?

Mutation and Iteration

- Avoid mutating a list while iterating over the list

```
myList = [1, 2, 3, 4]  
  
for ele in myList:  
    myList.remove(ele)
```

- *next()* method uses an index to retrieve the elements from *myList*
- *remove()* method mutates the *myList*, but the index in *next()* method will not be updated

Example: Find Max in A List of No.

```
list1 = [2,101,3,1,6,33,22,4,99,123,55]
```

```
def findMax(lst):
    maxsofar = lst[0]
    for i in lst:
        if i > maxsofar:
            maxsofar = i
    return maxsofar
```

```
>>> print(findMax(list1))
123
```

- Is there any potential problem?

Example: Find all Even Numbers

```
def findAllEvenNo(lst):
    output = []
    for i in lst:
        if i % 2 == 0:
            output.append(i)
    return output

>>> print(findAllEvenNo(list1))
[2, 6, 22, 4]
```

List Comprehensions

- Provides a concise way to apply an operation to the items in iterable object and store the result as a list
- Syntax:
 - $[expr \text{ for } elem \text{ in } iterable \text{ if } test]$
- Returns an iterable

List Comprehension

- Todo:
 - create a list:

```
a_list = [1,2,3,4,5,6,..... , 100]
```

- You can

```
>>> a_list = []
>>> for i in range(1,101):
    a_list.append(i)
```

```
>>> a_list
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58,
59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72,
73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86,
87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100
]
```

List Comprehension

- Or

The item really in the list

every i between 1 and
101 (exclusive)

```
>>> b_list = [ i for i in range(1,101) ]  
>>> b_list  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,  
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,  
31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,  
45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58,  
59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72,  
73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86,  
87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100  
]
```

$$b = \{i | i \in [1, 101)\}$$

Compare to
ordinary math
equation

List Comprehension

- How do I produce a list of first 10 squared numbers?

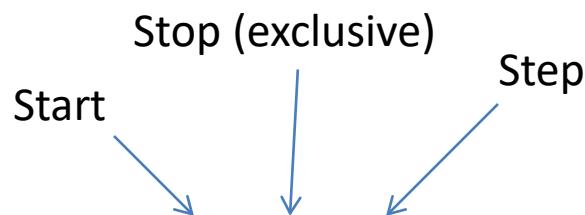
```
>>> d_list = [i*i for i in range(1,11)]  
>>> d_list  
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

$$b = \{i^2 | i \in [1,101)\}$$

Compare to
ordinary math
equation

List Comprehension

- How do I produce a list of odd numbers less than 100
 - Like string slicing



```
>>> c_list = [i for i in range(1, 101, 2)]
>>> c_list
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29,
31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57,
59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85,
87, 89, 91, 93, 95, 97, 99]
```

List Comprehension

- How do I produce a list of **even** numbers less than 100
 - Similar to the previous one but start with 2
 - Or

```
>>> c2_list = [i for i in range(1,101) if i not in c_list]
>>> c2_list
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100]
```

Advance: Generate Prime Numbers

- Let's generate all the prime numbers < 50
- First, generate all the non-prime numbers < 50

```
>>> for i in range(2,8):  
    print([j for j in range(i*2, 50, i)])
```

i is from 2 to 7
($7 = \sqrt{50}$)

get all the multiples of i
from $2*i$ to 49

```
[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,  
34, 36, 38, 40, 42, 44, 46, 48]  
[6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48]  
[8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48]  
[10, 15, 20, 25, 30, 35, 40, 45]  
[12, 18, 24, 30, 36, 42, 48]  
[14, 21, 28, 35, 42, 49]
```

Advance: Generate Prime Numbers

- Let's generate all the prime numbers < 50
- First, generate all the non-prime numbers <50

```
i is from 2 to 7  
get all the multiples of i  
from 2*i to 49  
  
>>> nonprime =[j for i in range(2,8) for j in range(i*2, 50, i)]  
>>> nonprime  
[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36,  
, 38, 40, 42, 44, 46, 48, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33,  
36, 39, 42, 45, 48, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 1  
0, 15, 20, 25, 30, 35, 40, 45, 12, 18, 24, 30, 36, 42, 48, 14, 2  
1, 28, 35, 42, 49]  
  
i = 2  
i = 3  
i = 4
```

Generate Prime Numbers

- Let's generate all the prime numbers < 50
- First, generate all the non-prime numbers < 50
- Prime numbers are the numbers NOT in the list above

```
>>> nonprime =[j for i in range(2,8) for j in range(i*2, 50, i)]  
>>> prime = [x for x in range(1,50) if x not in nonprime]  
>>> prime  
[1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Generator Expressions

- Provides a generator that can be used to iterate over without explicitly generating the list of items
- Syntax:
 - $(\text{expr} \text{ for } \text{elem} \text{ in } \text{iterable} \text{ if } \text{test})$
- Returns an iterator
- Requires less memory than list
 - Check!

Generator Expressions

```
num = 4
# square each term using list comprehension
my_square_list = [x**2 for x in range(num)]
my_square_generator = (x**2 for x in range(num))

for k in my_square_list:
    print(k)

for k in my_square_generator:
    print(k)
```

Output:

```
0
1
4
9
0
1
4
9
```

Which is better?

Generator Expressions

```
num = 10**4
# square each term using list comprehension
my_square_list = [x**2 for x in range(num)]
my_square_generator = (x**2 for x in range(num))

import sys
print(f'Size of my_square_list {sys.getsizeof(my_square_list)}')
print(f'Size of my_square_generator {sys.getsizeof(my_square_generator)}')
```

Output:

```
Size of my_square_list 87624
Size of my_square_generator 120
```

Sequence in Python

- Indexed collection
 - Strings
 - Lists
 - **Tuples**
- Non-indexed collection:
 - Sets
 - Dictionary

Tuples

- A static and an immutable array/list

- Syntax:

- `int_tuple = (1,2,3)`
- `float_tuple = (1.0,2.0,3.0)`
- `str_tuple = ('hi','IT5001')`
- `mixed_tuple = (1,1.0,'IT5001')`

Task	Syntax
Return i-th element	<code>a[i]</code>
Return elements from i to j-1	<code>a[i:j]</code>
Return number of elements	<code>len(a)</code>
Return smallest value in sequence	<code>min(a)</code>
Return largest value in sequence	<code>max(a)</code>
Returns if an element is part of sequence	<code>x in a</code>
Concatenates two sequences	<code>a + b</code>
Creates n copies of a sequence	<code>a * n</code>

Tuples: Example 1

Python 3.6
[\(known limitations\)](#)

```
1 my_tuple_1 = (1,2,3,4)
2
3 my_tuple_2 = my_tuple_1
```

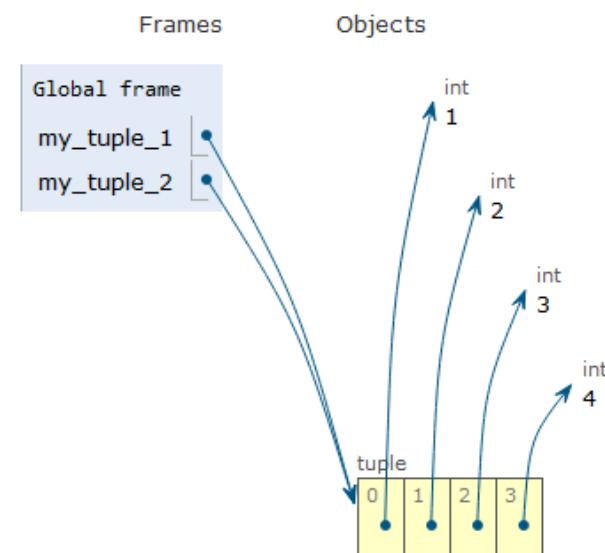
[Edit this code](#)

→ line that just executed
→ next line to execute

<< First < Prev Next > Last >>

Done running (2 steps)

[Customize visualization](#)



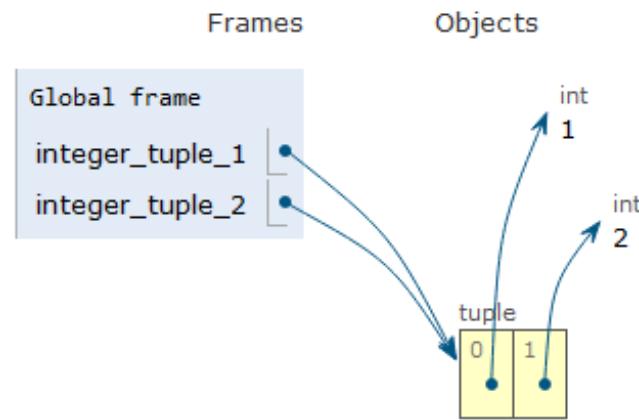
Tuples: Example 2

Python 3.6
(known limitations)

```
1 integer_tuple_1= (1,2)
2
3 integer_tuple_2 = integer_tuple_1
4
5 integer_tuple_1 + (3,4)
6
7 integer_tuple_1 = integer_tuple_1 + (3,4)
```

[Edit this code](#)

that just executed
: line to execute



Immutable:

creates a new tuple – but not assigned

Tuples: Example 2

Python 3.6
([known limitations](#))

```
1 integer_tuple_1= (1,2)
2
3 integer_tuple_2 = integer_tuple_1
4
5 integer_tuple_1 + (3,4)
6
→ 7 integer_tuple_1 = integer_tuple_1 + (3,4)
```

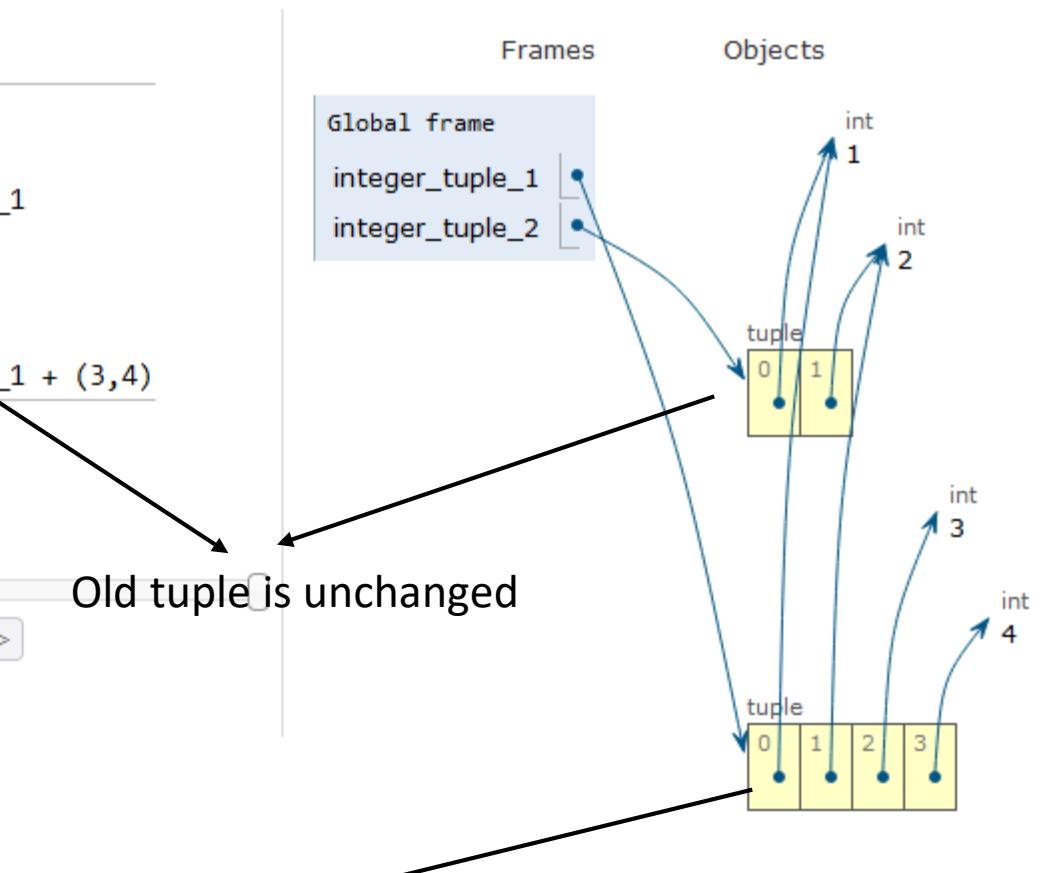
[Edit this code](#)

at just executed
line to execute

<< First < Prev Next > > Last >>

Done running (4 steps)

[visualization](#)



Old tuple is unchanged

Creates a new tuple and assigned to `integer_tuple_1`

Lists and Tuples

- Similarities:
 - List and Tuple are
 - Indexed
 - Iterable
 - Both can store heterogeneous data types
- Differences:
 - List is mutable
 - Tuple is immutable

Tuple

- A Tuple is basically a list but
 - CANNOT be modified

```
>>> a_tuple = (12, 13, 'dog') ← Tuples use '(' and ')'  
>>> a_tuple[1] ← Lists use '[' and ']'  
13  
>>> a_tuple[1] = 9  
Traceback (most recent call last):  
  File "<pyshell#130>", line 1, in <module>  
    a_tuple[1] = 9  
TypeError: 'tuple' object does not support item assignment  
>>> a_tuple.append(1)  
Traceback (most recent call last):  
  File "<pyshell#131>", line 1, in <module>  
    a_tuple.append(1)  
AttributeError: 'tuple' object has no attribute 'append'  
>>>
```

Tuple

- A Tuple is basically a list but
 - CANNOT be modified

```
>>> t1 = (1,2,3)
>>> t1.append(3)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    t1.append(3)
AttributeError: 'tuple' object has no attribute 'append'
>>> t1.remove(1)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    t1.remove(1)
AttributeError: 'tuple' object has no attribute 'remove'
```

For a Singleton of List and Tuple...

```
>>> a_list = [3,5,8]
>>> print(a_list)
[3, 5, 8]
>>> type(a_list)
<class 'list'>
```

```
>>> a_tuple=(3,5,8)
>>> print(a_tuple)
(3, 5, 8)
>>> type(a_tuple)
<class 'tuple'>
```

- a list with only one element
- a tuple with only one element

```
>>> b_list = [3]
>>> print(b_list)
[3]
>>> type(b_list)
<class 'list'>
>>> |
```

```
>>> b_tuple=(3)
>>> print(b_tuple)
3
>>> type(b_tuple)
<class 'int'> !!!
```

A Tuple with only one element

```
>>> b_tuple=(3)
>>> print(b_tuple)
3
>>> type(b_tuple)
<class 'int'>
```

- Correct way

```
>>> c_tuple = (3,)
>>> print(c_tuple)
(3,)
>>> type(c_tuple)
<class 'tuple'>
>>> c_tuple[0]
3
```

Note the
comma
here

But then, why use Tuple? Or List?

Or when to use Tuple? When to use
List?

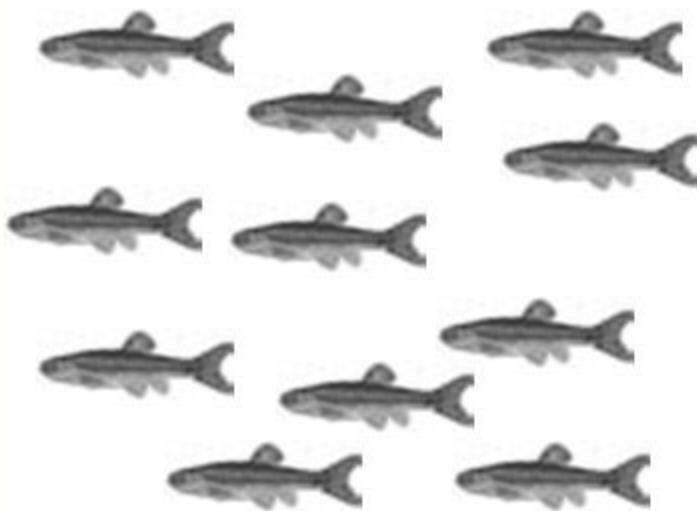
English Grammar

- Which sentence is grammatically correct?
 - “I have more than one fish. Therefore, I have many *fish*”
 - “I have more than one fish. Therefore, I have many *fishes*”
- Both of them are grammatically correct!
 - But they mean different things

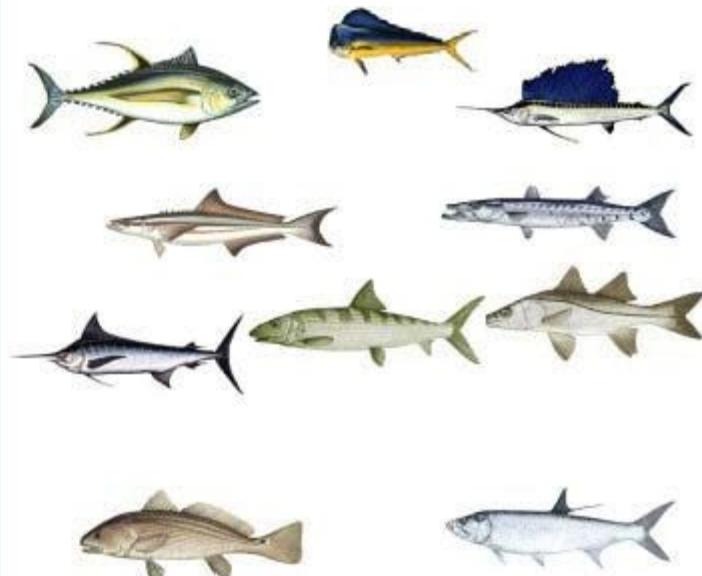
Fish vs Fishes

- The plural of fish is usually *fish*.
- When referring to more than one species of fish, especially in a scientific context, you can use *fishes* as the plural.

Fish vs. Fishes



"This tank is full of fish."



"The ocean is full of fishes."

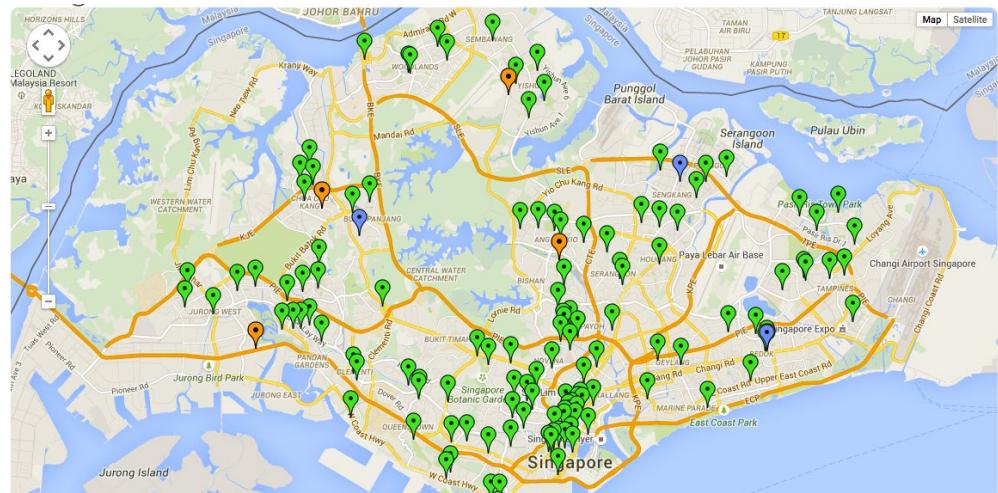
List vs Tuple, Cultural Reason

- List
 - Usually stores a **large** collection of data with the **same type (homogenous)**
 - E.g. List of 200 student names in a class
- Tuple
 - Usually stores a **small** collections of items with **various data types/concepts (heterogeneous)**
 - E.g. A single student record with name (string), student number(string) and mark(integer)

But, violating this “culture” will NOT cause any syntax error

An Example

- To store the data on a map
 - These are the locations of **100** nice restaurants in Singapore
 - The location of each restaurant is recorded as the coordinates value of x and y
 - (100,50)
 - (30, 90)
 - (50, 99)
 - etc...

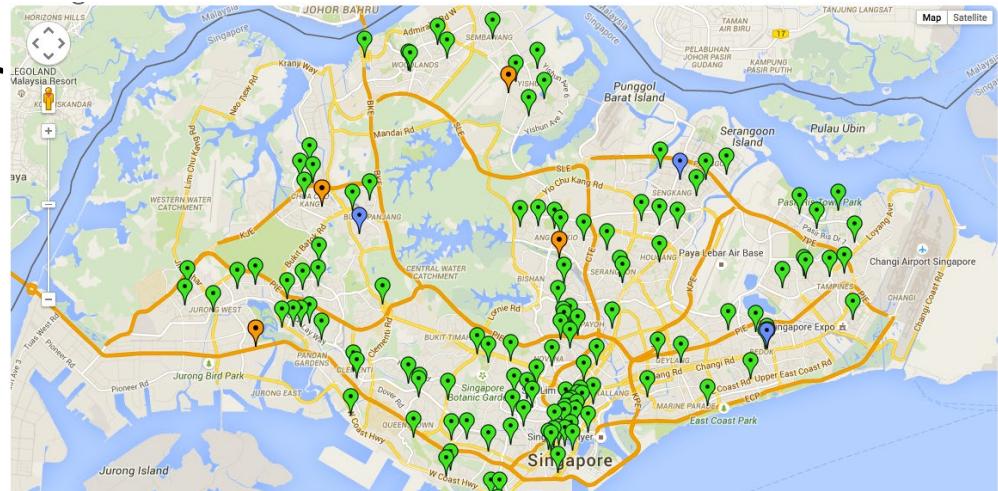


An Example

- I will code like this

```
locations_of_nice_restaurants = [ (100, 50),  
                                  (30, 90), (50, 90) ]
```

- Is it
 - a tuple of tuples,
 - a tuple of lists,
 - a list of tuples, or
 - a list of lists?



Find all the restaurants near me

- I will code like this

```
locations_of_nice_restaurants = [(100,50),  
                                  (30,90), (50,90)]
```

shortened the name

```
def find_restaurants(my_current_pos):  
    locations = generate_list()  
    output_list = []  
  
    for loc in locations:  
        if distance(my_current_pos, loc) < DISTANCE_RANGE:  
            output_list.append(loc)  
  
    return output_list
```

```
def find_restaurants(my_current_pos):  
    locations = generate_list()  
    output_list = []  
  
    for loc in locations:  
        if distance(my_current_pos, loc) < DISTANCE_RANGE:  
            output_list.append(loc)  
  
    return output_list  
  
def generate_list():  
    output_list = []  
    for i in range(NO_RESTAURANTS):  
        output_list.append((random.randint(1,SIZE_OF_SG),  
                            random.randint(1,SIZE_OF_SG)))  
  
    return output_list  
  
def distance(p1, p2):  
    return sqrt( square(p1[0]-p2[0]) + square(p1[1]-p2[1]))  
  
def square(x):  
    return x * x
```

Just a fake function
to generate the list
for this demo

A list

A tuple

```
def find_restaurants(my_current_pos):  
    locations = generate_list()  
    output_list = []  
  
    for loc in locations:  
        if distance(my_current_pos, loc) < DISTANCE_RANGE:  
            output_list.append(loc)  
  
    return output_list
```

```
>>> find_restaurants((50, 50))  
[(45, 52), (59, 47), (51, 41)]  
>>> find_restaurants((50, 50))  
[(55, 48), (54, 55)]  
>>> find_restaurants((50, 50))  
[(51, 58), (45, 47)]  
>>> find_restaurants((50, 50))  
[(43, 55), (48, 43), (43, 48), (54, 43)]
```

Challenge:
Find the nearest THREE restaurants

Instead of ALL

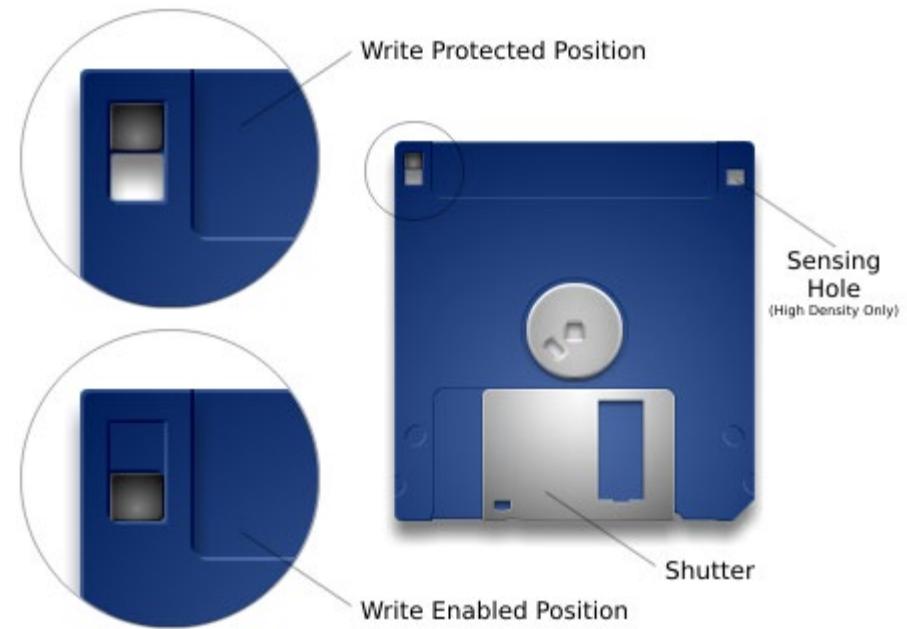
List vs Tuple, Cultural Reason

- List
 - Usually stores a **large** collection of data with the **same type (homogenous)**
 - E.g. List of 200 student names in a class
- Tuple
 - Usually stores a **small** collections of items with **various data types/concepts (heterogeneous)**
 - E.g. A single student record with name (string), student number(string) and mark(integer)

But, violating this “culture” will NOT cause any syntax error

List vs Tuple, Technical Reasons

- Immutable vs mutable
 - Tuple is Write protected (Immutable)
- List can be changed within a function
 - NOT passed by value
 - Mutable



Recap: Primitive Data Types

```
x = 0
```

```
def changeValue(n):  
    n = 999  
    print(n)
```

```
changeValue(x)  
print(x)
```

- The `print()` in “changeValue” will print 999
- But how about the last `print(x)`?
 - Will x becomes 999?
- (So actually this function will NOT change the value of x)

Recap: Primitive Data Types

```
x = 0
```

```
def changeValue(n):  
    n = 999  
    print(n)
```

```
changeValue(x)  
print(x)
```

- n is another copy of x
- You can deem it as

```
def changeValue(x):  
    n = x  
    n = 999  
    print(n)
```

Pass By Values

But for List

- Mutable!

```
>>> l = [1, 2, 3]
>>> changeSec(l)
Inside function
[1, 'changed!', 3]
>>> print(l)
[1, 'changed!', 3] !!!
```

```
def changeSec(a):
    a[1] = 'changed!'
    print('Inside function')
    print(a)
```

Sequence in Python

- Indexed collection
 - Strings
 - Lists
 - Tuples
- Non-indexed collection:
 - Sets
 - Dictionary

Sets

- A set is an **unordered** collection of **immutable** elements with **no duplicate** elements
 - Unordered: You cannot get a single element by its index like `s[2]`
 - No duplicate: every element exists only once in a set

```
>>> set1 = {1,2,3,4,5,6,7,8,1,2,3}  
>>> set1  
{1, 2, 3, 4, 5, 6, 7, 8}
```

Tuples use '(' and ')'
Lists use '[' and ']'
Sets use '{' and '}'

Python
Removes
duplicates
for you

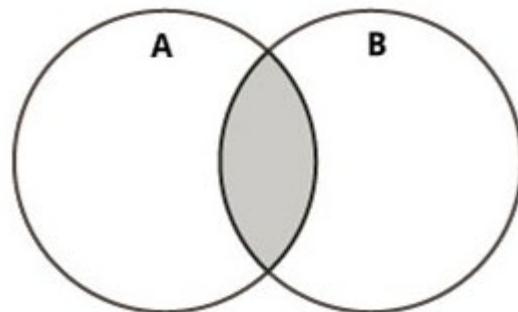
Sets

- Some operations are not available because sets are NOT indexed

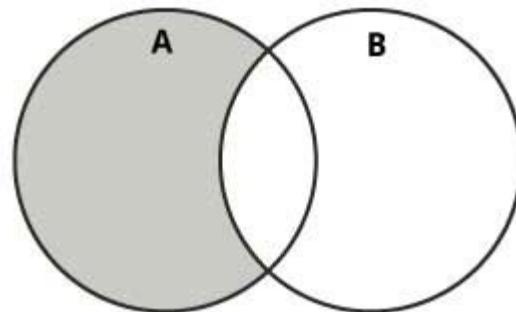
a[i]	return i-th element of a
a[i:j]	returns elements i up to j - 1
len(a)	returns numbers of elements in sequence
min(a)	returns smallest value in sequence
max(a)	returns largest value in sequence
x in a	returns True if x is a part of a
a + b	concatenates a and b
n * a	creates n copies of sequence a

Set Operations

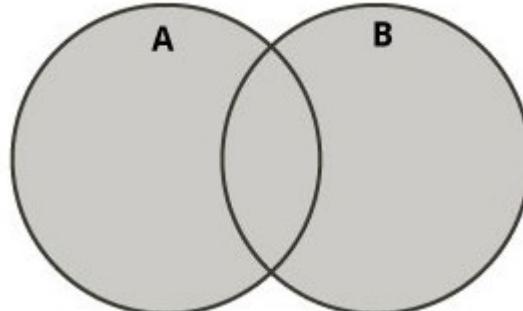
- Intersection



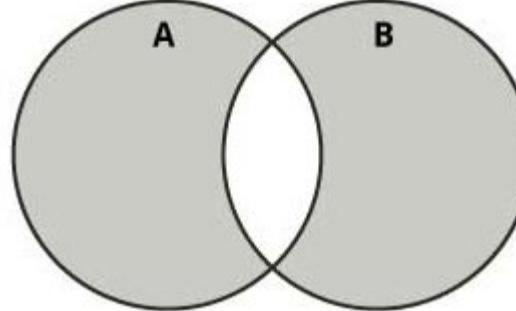
- $A - B$



- Union



- Symmetric Difference



Sets

- Usual set operations

```
>>> setA = {1,2,3,4}  
>>> setB = {3,4,5,6}  
>>> setA | setB ← Union  
{1, 2, 3, 4, 5, 6}  
>>> setA & setB ← Intersection  
{3, 4}  
>>> setA - setB ← A - B  
{1, 2}  
>>> setA ^ setB ← (A | B) - A & B  
{1, 2, 5, 6}
```

Sets

```
>>> setA.remove(1) ← Remove like a list
>>> setA
{2, 3, 4}
>>> setA.remove(1) ← But error if element missing
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    setA.remove(1)
KeyError: 1
>>> setA.discard(1) ← But we can use
>>> | discard instead
```

Sets are Iterable

```
my_set = {1, 2, 3}           1  
for i in my_set:  
    print(i)                 2  
                            3
```

Set from List and Vice-Versa

```
>>> my_list = [1,2,3]
>>> my_set = set(my_list)
>>> my_set
{1, 2, 3}
```

```
>>> my_set = {4,5,6}
>>> my_list = list(my_set)
>>> my_list
[4, 5, 6]
```

Sequence in Python

- Indexed collection
 - Strings
 - Lists
 - Tuples
- Non-indexed collection:
 - Sets
 - Dictionary

Dictionary

e•merge (ĕ-mûrj') *v.* **e•merged**, **e•merging**.

1. To rise up or come forth into view; appear. 2. To come into existence. 3. To become known or evident. [Lat. *emergere*.] —**e•mer'gence** *n.* —**e•mer'gent** *adj.*

e•merg•ency (ĕ-mûr'jĕn-sĕ) *n.*, *pl.* -**ies**. An unexpected situation or occurrence that demands immediate attention.

e•mer•itus (ĕ-mĕr'i-tăs) *adj.* Retired but retaining an honorary title: *a professor emeritus*. [Lat., p.p. of *emereri*, to earn by service.]

em•er•y (ĕm'ĕ-rĕ, ĕm'rĕ) *n.* A fine-grained impure corundum used for grinding and polishing. [< Gk *smuris*.]

e•met•ic (ĕ-mĕt'ik) *adj.* Causing vomiting. [< Gk. *emein*, to vomit.] —**e•met'ic**, *n.*

—emia suff. Blood: *leukemia*. [< Gk. *haima*, blood.]

em•i•grate (ĕm'i-grăt') *v.* -**grat•ed**, -**grat•ing**. To leave one country or region to settle in another. [Lat. *emigrare*.] —**em'i-grant** *n.* —**em'i-gra'tion** *n.*

é•mi•gré (ĕm'i-gră') *n.* An emigrant, esp. a refugee from a revolution. [Fr.]

em•i•nence (ĕm'ĕ-nĕns) *n.* 1. a position of great distinction or superiority. 2. A rise or elevation of ground; hill.

em•i•nent (ĕm'ĕ-nĕnt) *adj.* 1. Outstanding, as in reputation; distinguished. 2. Towering above others; projecting. [< Lat. *eminere*, to stand out.] —**em'i•nently** *adv.*

em•phatic (ĕm-făt'ik) *adj.* Expressed or performed with emphasis. [< Gk. *emphatikos*.] —**em•phat'ically** *adv.*

em•physe•ma (ĕm'fi-sĕ'mă) *n.* A disease in which the air sacs of the lungs lose their elasticity, resulting in an often severe loss of breathing ability. [< Gk. *emphusēma*.]

em•pire (ĕm'pir') *n.* 1. A political unit, usu. larger than a kingdom and often comprising a number of territories or nations, ruled by a single central authority. 2. Imperial dominion, power, or authority. [< Lat. *imperium*.]

em•pir•ical (ĕm-pir'i-kăl) *adj.* Also **em•pir•ic** (-pir'ik). 1. Based on observation or experiment. 2. Relying on practical experience rather than theory. [< Gk. *empeirikos*, experienced.] —**em•pir'ically** *adv.*

em•pir•ic•ism (ĕm-pir'i-siz'ĕm) *n.* 1. The view that experience, esp. of the senses, is the only source of knowledge. 2. The employment of empirical methods, as in science. —**em•pir'ic•ist** *n.*

em•place•ment (ĕm-plăs'mĕnt) *n.* 1. A prepared position for guns within a fortification. 2. Placement. [Fr.]

em•ploy (ĕm-ploï') *v.* 1. To engage or use the services of. 2. To put to service; use. 3. To devote or apply (one's time or energies) to an activity. —**n.** Employment. [< Lat. *implicare*, to involve.] —**em•ploy'a•ble** *adj.*

em•ploy•ee (ĕm-ploï'ĕ, ĕm'ploï-ĕ') *n.* Also **em•ploy•e**. One who works for another.

Word

Its meaning

Word

Its meaning

ā pat ā pay ā care ā father ē pet ē be ī pit ī tie ī pier ō pot ō toe ō paw, for ō noise
ōō took ōō boot ōō out th̄ thin th̄ this ū cut ū urge yoo abuse zh̄ vision ō about, item,
edible, gallop, circus

Dictionary

- You search for the word in the dictionary
- Then look for its meaning



- Each word has a **correspondent** meaning

Python Dictionary

- You search for the **key** in the dictionary
- Then look for its **value**



- Each key has a **correspondent value**

```
>>> students = {'A100000X': 'John', 'A123456X': 'Peter',  
'A999999X': 'Paul'}  
>>> students['A123456X']  
'Peter'
```

key : value
pair

Tuples use '(' and ')'
Lists use '[' and ']'
Sets and Dict use '{' and '}'

An Example

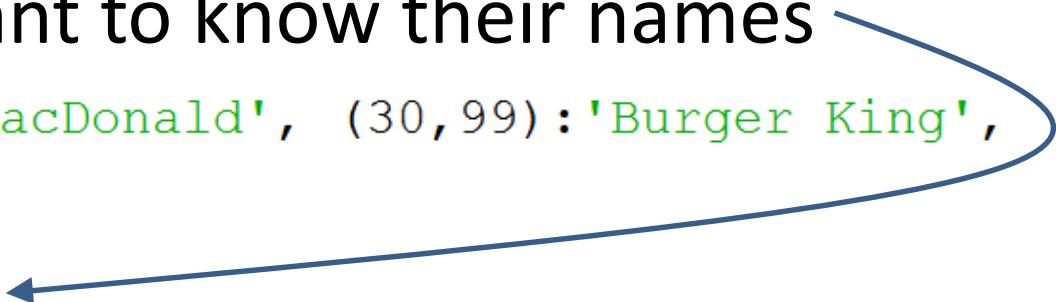
- To store the data on a map
 - These are the locations of **100** nice restaurants in Singapore
 - The location of each restaurant is recorded as the coordinates value of x and y **and name**
 - (10,20):Pizza Hut



Python Dictionary

- Key: location
- Value: restaurant name
- After you searched for the nearest restaurants, you want to know their names

```
>>> locations = { (10,30) : 'MacDonald', (30,99) : 'Burger King',  
(22,33) : 'Pizza Hut'}  
>>>  
>>> locations[ (22,33) ]  
'Pizza Hut'
```



Recap: List

- Or tuples

```
>>> vm = ['M&M', 'Twix', 'Milky Way', 'Oreo']  
>>> vm[1]  
'Twix'
```



Index:
From 0 to $\text{len}(a)-1$

Input a number



Output an item



But when you go to Japan

- You are not inputting a number (index)!

```
>>> vmj = {'Beef noodle small':290, 'Beef noodle big':390}  
>>> vmj['Beef noodle small']  
290
```

Input a ~~number~~ a name



Output an item



To set up a dictionary

- Each pair has a key and a value

```
>>> vmj = {'Beef noodle small':290, 'Beef noodle big':390}
```

The code defines a dictionary named `vmj` with two entries. Each entry consists of a string key and an integer value. The keys are 'Beef noodle small' and 'Beef noodle big', and the values are 290 and 390 respectively. The entire dictionary is enclosed in curly braces. Below the dictionary, blue curly braces group each key-value pair together, with the word 'key' under the first brace and 'value' under the second.

What is Dictionary?

- Key is on the left, Value on the right

```
>>> my_dictionary = {'a':1, 'b':2}  
>>> my_dictionary['b']  
2
```

- Summary: A data structure used for
“When I give you X, give me Y”
- Can store any type
- Called HashTable in some other languages

How is a Dictionary Useful?

- Keep Track of Things by Key!
 - Eg, keeping track of stocks of fruits

```
my_stock = {"apples":450,"oranges":412}
```

```
my_stock["apples"]
```

```
>>> 450
```

```
my_stock["apples"] + my_stock["oranges"]
```

```
>>> 862
```

How is a Dictionary Useful?

- Keep Track of Things by Key!
 - When you want to get an associated operation
(eg, alphabets to numeric integers)

```
my_alphabet_index = {'a':1,'b':2... 'z':26}
my_alphabet_index['z']
>>> 26
```

Dictionary Methods

- Access (VERY FAST! - Almost instant!)
- Assignment
- Removal
- Other Dictionary Methods

Dictionary Access

```
>>> my_fruit_inventory = {"apples":450,"oranges":200}  
>>> my_fruit_inventory["apples"]  
450  
>>> my_fruit_inventory.get("apples")  
450  
>>> my_fruit_inventory["pears"]  
KeyError!  
>>> my_fruit_inventory.get("pears")  
None
```

****Cannot access keys which don't exist!****

- Accessing with [] will crash if does not exist
- Accessing with .get() will NOT crash if key does not exist

Dictionary Assignment

```
>>> my_fruit_inventory[“pears”] = 100  
>>> print(my_fruit_inventory)  
{“apples”:450, “oranges”:200, “pears”:100}
```

- Caution: This OVERWRITES existing values!

```
>>> my_fruit_inventory[“oranges”] = 100  
>>> print(my_fruit_inventory)  
{“apples”:450, “oranges”:100, “pears”:100}
```

Dictionary Removal

```
>>> my_fruit_inventory =  
{“apples”:450,“oranges”:200}  
  
>>> my_fruit_inventory.pop(“apples”)  
>>> print(my_fruit_inventory)  
{‘oranges’:200}
```

- OR

```
>>> del my_fruit_inventory[“apples”]
```

Other Dictionary Methods

.clear()

- clear all

.copy()

- make a copy

.keys()

- return all keys

.values()

- return all values

.items()

- return all keys + values

Dictionary is Iterable

```
my_dict = {'a':1, 'b':2}
```

```
for key in my_dict:  
    print(key)
```

```
for key in my_dict:  
    print(key, my_dict[key])
```

```
for key, value in my_dict.items():  
    print(key, value)
```

```
a  
b  
a 1  
b 2  
a 1  
b 2
```

Sequence in Python

- Indexed
 - Strings
 - Lists
 - Tuples
- Non-indexed collection:
 - Sets
 - Dictionary



NUS | Computing

National University
of Singapore

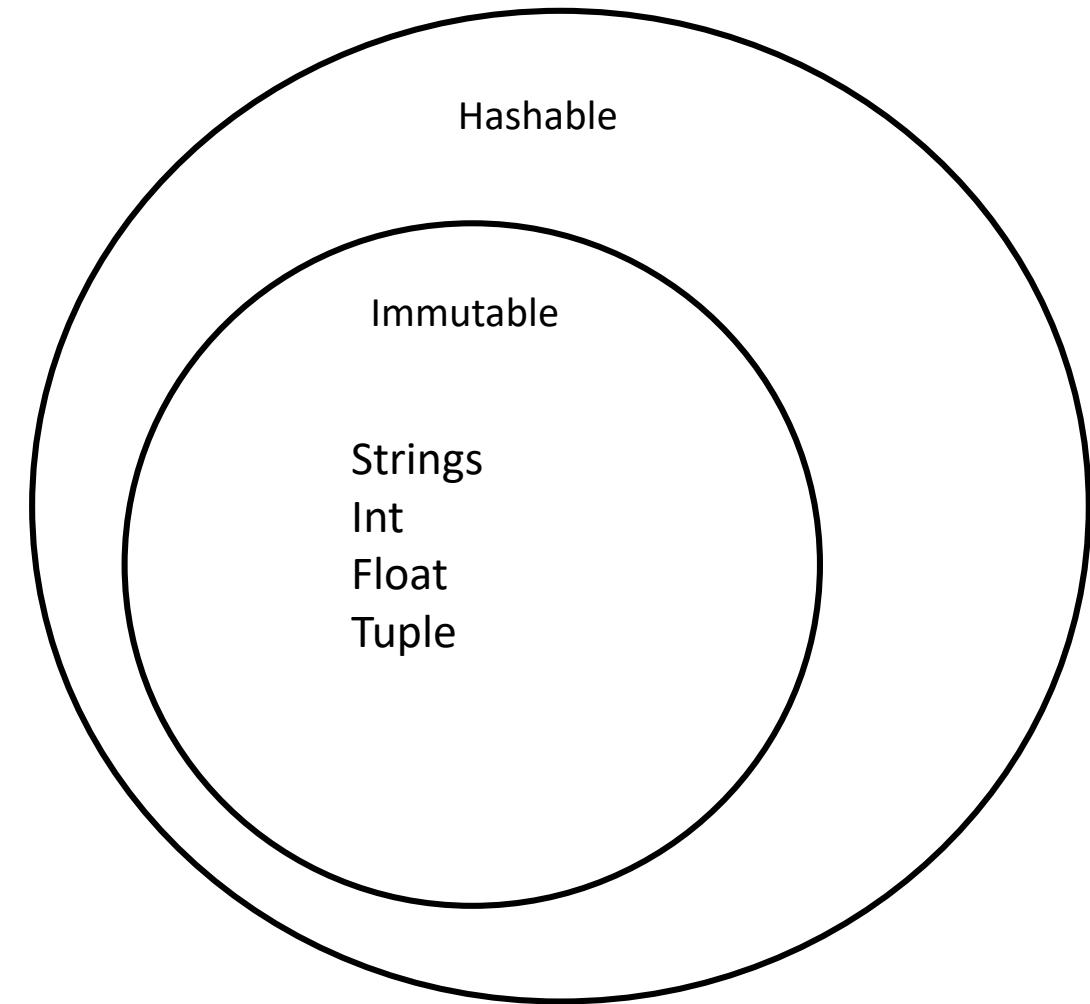
IT5001 Software Development Fundamentals

8. Sequences Contd.

Sirigina Rajendra Prasad

Python: Hashability and Immutability

Data Type	Immutable	Hashable
Integer	Yes	Yes
Float	Yes	Yes
String	Yes	Yes
Tuple	Yes	Yes
List	No	No
Set	No	No
Dictionary	No	No



All immutable objects are Hashable but not vice-versa

Returning Multiple Values

```
def f():
    return 1, 2
```



Treated as a tuple

```
x = f()
print(x)
print(type(x))
x, y = f()
print(x, y)
print(type(x), type(y))
```



Unpacking of tuple

Output:

```
(1, 2)
<class 'tuple'>
1 2
<class 'int'> <class 'int'>
```

If you don't know number of arguments

Unknown number of positional arguments

```
def f(*args):
    print(f'Type of input arguments: {type(args)}')
    for i in args:
        print(i)

f(1, 2, 3, 4)
```

Output:

```
Type of input arguments: <class 'tuple'>
1
2
3
4
```

If you don't know number of arguments

Unknown number of keyword arguments

```
def f(**kwargs):  
    print(f'Type of input arguments: {type(kwargs)}')  
    for i,j in kwargs.items():  
        print(f'{i} = {j}')
```

```
f(arg_1 = 1,arg_2 = 2,arg_3 = 3)
```

Output:

```
Type of input arguments: <class 'dict'>  
arg_1 = 1  
arg_2 = 2  
arg_3 = 3
```

Accessing Global Variables

- Can a function access (read) a global variable?
 - Variable is Immutable
 - Yes
 - Variable is Mutable
 - Yes

```
def my_func_1(y):  
    return x+y  
x = 2  
print(x)  
print(my_func_1(4))  
print(x)  
  
x = [2,3]  
print(x)  
print(my_func_1([1,4]))  
print(x)
```

Output:

```
2  
6  
2  
[2, 3]  
[2, 3, 1, 4]  
[2, 3]
```

Modifying Global Variables

- Can a function modify a global variable with variable declared **global** within local function?
 - Yes, for both mutable and immutable variables

```
def my_func_2(y):  
    global x  
    x = x+y  
    return x
```

```
x = 2  
print(x)  
my_func_2(3)  
print(x)
```



```
x = [2,3]  
print(x)  
my_func_2([4,5])  
print(x)
```

```
x = (2,3)  
print(x)  
my_func_2((4,5))  
print(x)
```

Output:

```
2  
5  
[2, 3]  
[2, 3, 4, 5]  
(2, 3)  
(2, 3, 4, 5)
```

Modifying Global Variables

- Can a function modify a global variable with variable **not** declared as global within local function?
 - No, for immutable variables
 - Yes, for mutable variables with **only** the methods that **mutate data** (append, sort, etc.)

```
def my_func_3(y):
    return x.append(y)

x = [2, 3]
print(x)
print(my_func_3([4, 5]))
print(x)
```

No assignment to variable *x*

Output:

[2, 3]
None
[2, 3, [4, 5]]



Scope

- Passing a mutable variable as argument

Python 3.6
[\(known limitations\)](#)

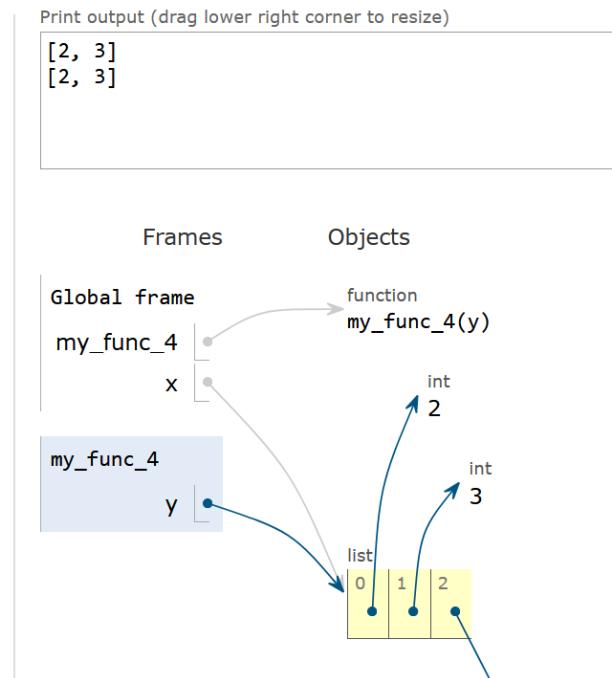
```
1 def my_func_4(y):
2     print(y)
3     y.append(4)
4     print(y)
5
6 x = [2,3]
7 print(x)
8 my_func_4(x)
9 print(x)
```

[Edit this code](#)

→ line that just executed
→ next line to execute

<< First < Prev Next > >>
Step 8 of 10

[Customize visualization](#)



Output:

```
[2, 3]
[2, 3]
[2, 3, 4]
[2, 3, 4]
```

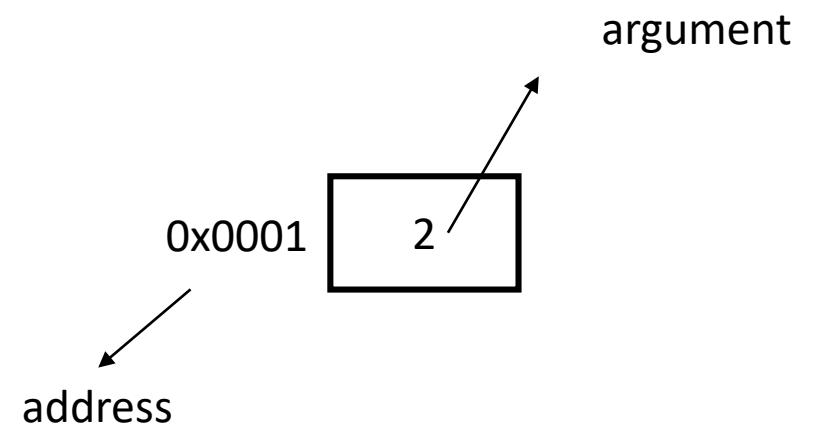
How are arguments passed to functions?

- Pass-by-Value

- An independent (duplicate) copy of argument is passed as input
- Not good if the argument size is very large
 - Requires additional memory and execution time

- Pass-by-Reference

- Address of argument is passed and function access the value from address
- Efficient for arguments of very large size



How about Python?

- Pass-by-Value or Pass-by-Reference?
 - Neither of them
- Python passes objects by assignment
 - Pass-by-Assignment
- Only exception is with *global* keyword

With **global** keyword

```
def square_1():
    global x
    x = x**2
equivalent to pass-by-reference
```

```
x=2
square_1()
```

Pass-by-Assignment

- Best Practice
 - Pass value by assignment, modify, and reassign

```
def square_2(y):  
    return y**2
```

```
x = 2  
x = square_2(x)
```

```
def modifyTup(t):  
    return t + (999,)
```

```
>>> tup = (1,2,3,4,5)  
>>> tup = modifyTup(tup)
```

Pass-by-Assignment

- For mutable variables

```
def my_func_1(y):  
    return x+y
```

```
x = [2, 3]  
print(x)  
print(my_func_1([1, 4]))  
print(x)
```

```
def my_func_3(y):  
    return x.append(y)
```

```
x = [2, 3]  
print(x)  
print(my_func_3([4, 5]))  
print(x)
```

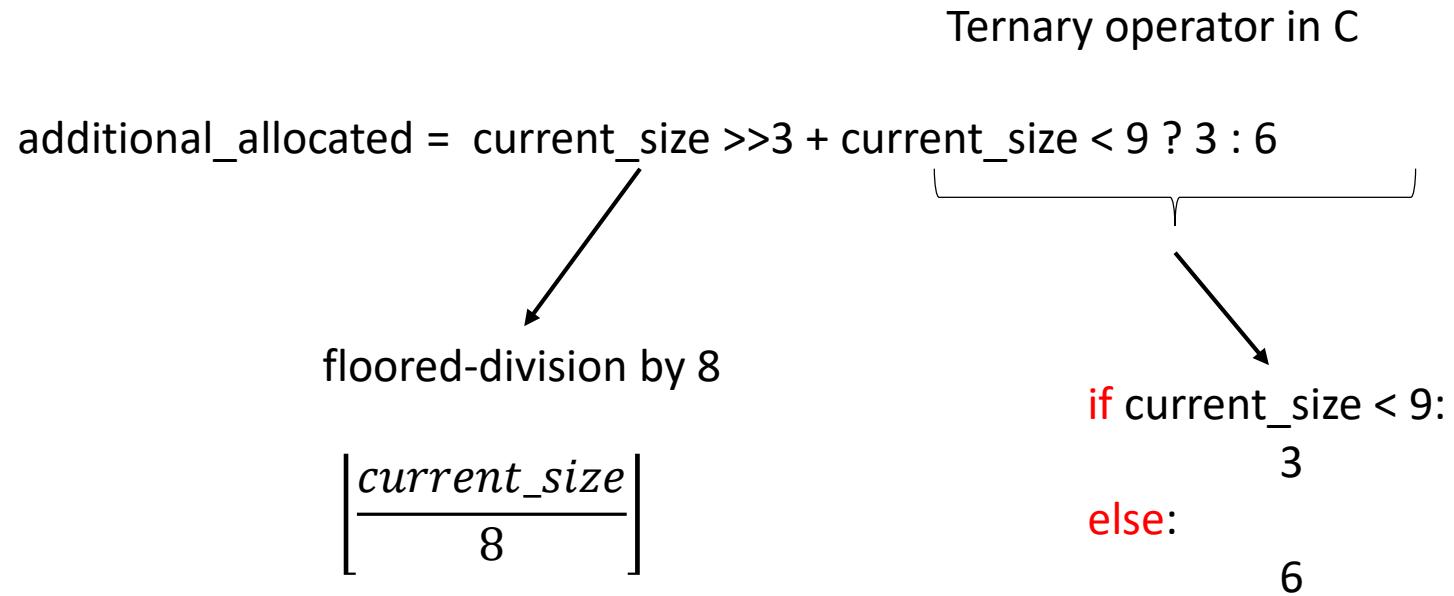
Effect is similar to pass-by-value

Effect is similar to pass-by-reference

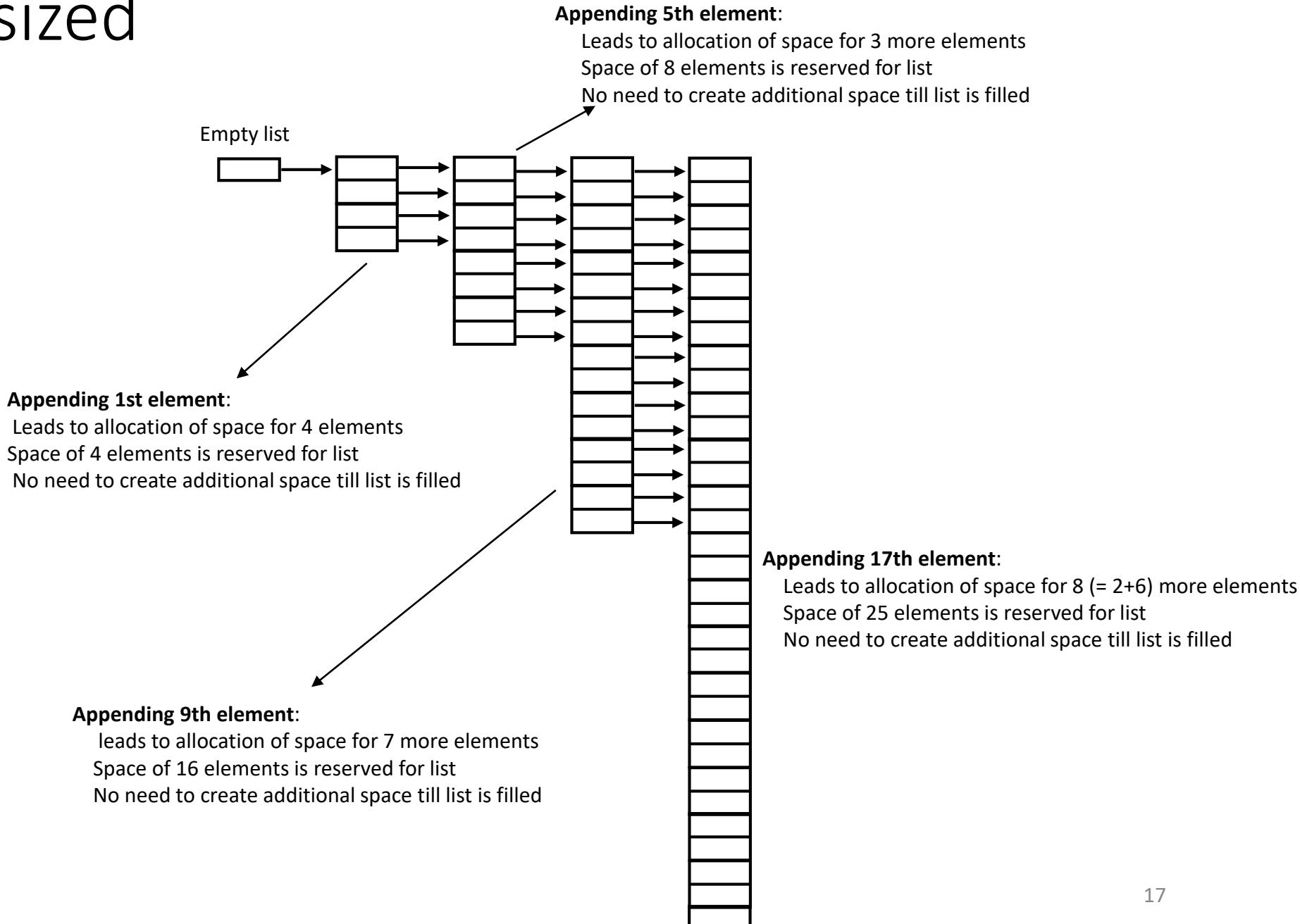
Scope: Summary

- Parameter Passing
 - Pass-by-Assignment
- Immutable Variable
 - Can access global variable
 - Cannot modify global variable unless declared global
- Mutable Variable
 - Can access global variable
 - Can modify the variable
 - Need to use **global** keyword for methods that do not mutate objects
 - No need of **global** keywords for methods that mutate objects

How are lists resized with *append*?



How are lists resized with *append*?

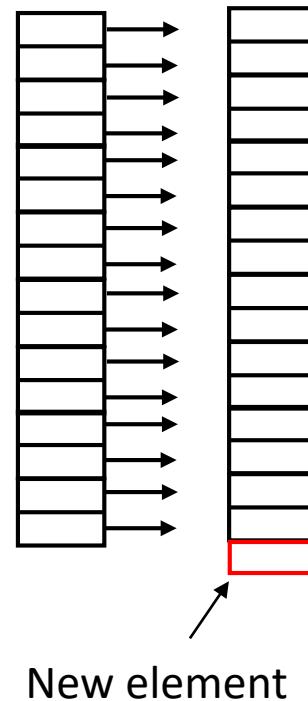


How are lists resized with *append*?

```
import sys
my_list = []
my_list_length = []
my_list_size = []
limit= 100
for k in range(limit):
    my_list_length.append(len(my_list))
    my_list_size.append(sys.getsizeof(my_list))
    my_list.append(k)
```

How are lists resized with *concatenation*?

- Always create a new list
 - Copies the contents of old list and new list combined



List Append Vs Concatenation

```
from time import time
import sys
my_list_append = []
my_list_append_length = []
my_list_append_size = []

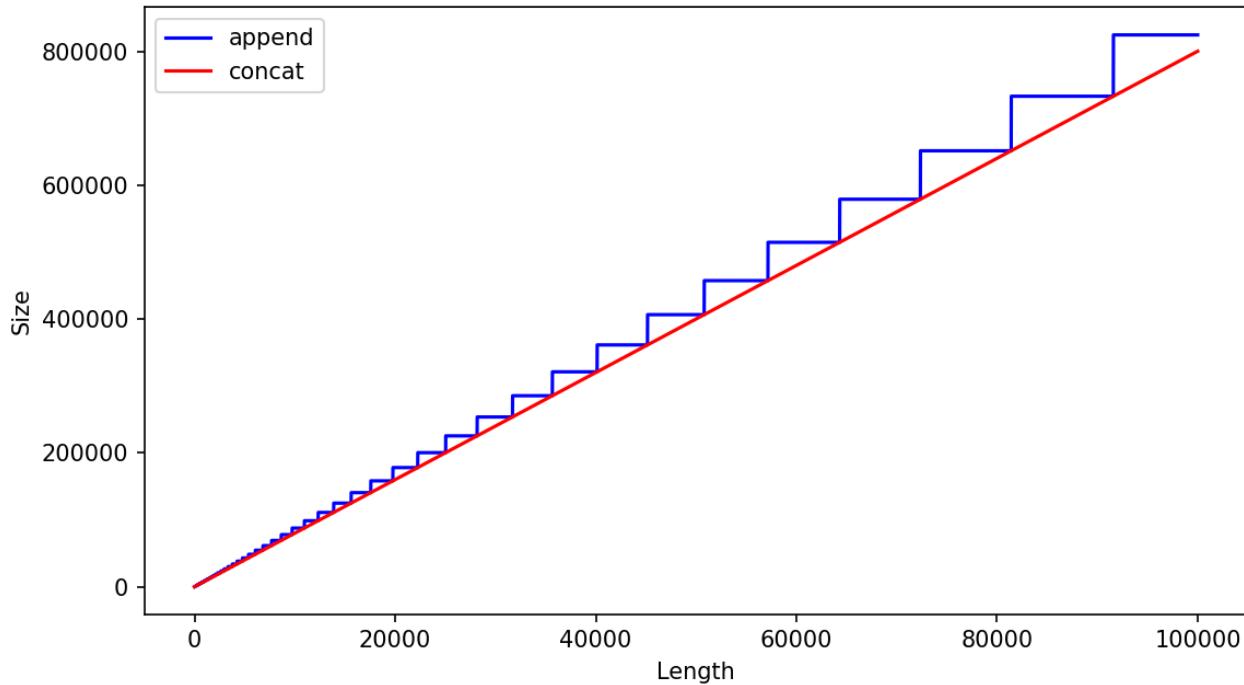
start= 0
end= 10**5

append_start_time = time()
for k in range(start,end):
    my_list_append_length.append(len(my_list_append))
    my_list_append_size.append(sys.getsizeof(my_list_append))
    my_list_append.append(k)
append_end_time = time()

my_list_concat = []
my_list_concat_length = []
my_list_concat_size = []
concat_start_time = time()
for k in range(start,end):
    my_list_concat_length.append(len(my_list_concat))
    my_list_concat_size.append(sys.getsizeof(my_list_concat))
    my_list_concat = my_list_concat + [k]
concat_end_time = time()

print(f'Append: {round(append_end_time-append_start_time,4)} s')
print(f'Concatenation: {round(concat_end_time - concat_start_time,4)} s')
from matplotlib import pyplot
pyplot.plot(my_list_append_length, my_list_append_size,color = 'b')
pyplot.plot(my_list_concat_length, my_list_concat_size,color='r')
pyplot.xlabel('Length')
pyplot.ylabel('Size')
pyplot.legend(['append','concat']))
pyplot.show()
```

List Append Vs Concatenation

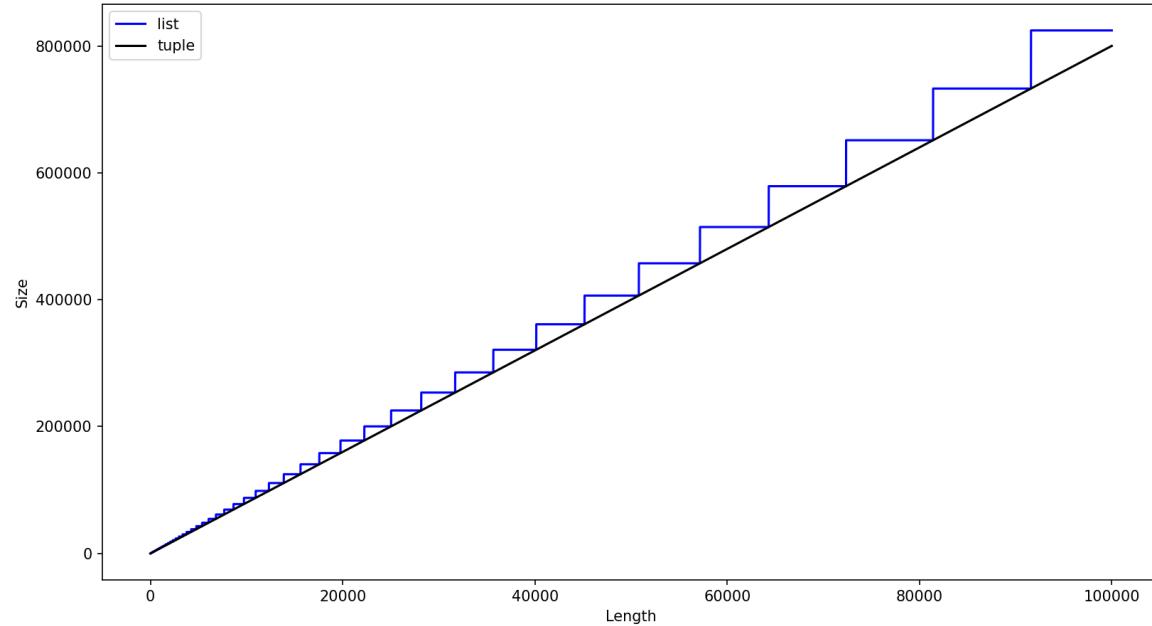


Append: 0.0316s

Concatenation: 20.7656s

Concatenation is slow as it creates new list for every new concatenation

List Append Vs Tuple Concatenation

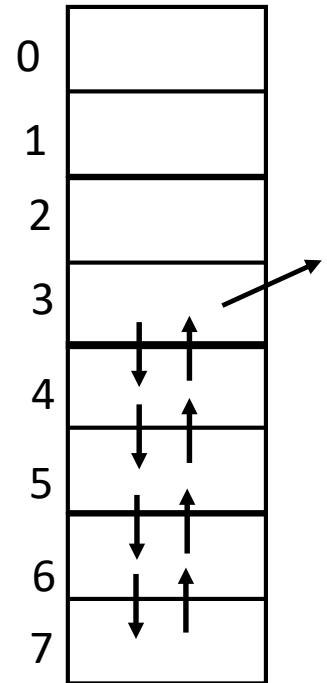


Append: 0.0211s

Concatenation: 20.3768s

Modifying tuple is slow as it creates new tuple for every new concatenation

Insertion/Deletion in lists?



Inserting of new element or
Deletion of an element
leads to relocation of
elements in subsequent indices

List Vs Tuple: Conclusion

Tuple Version



Preserve input



Must return result



Slower

List Version



May or may not preserve input

- May modify input



May or may not need to return result



Faster (append)



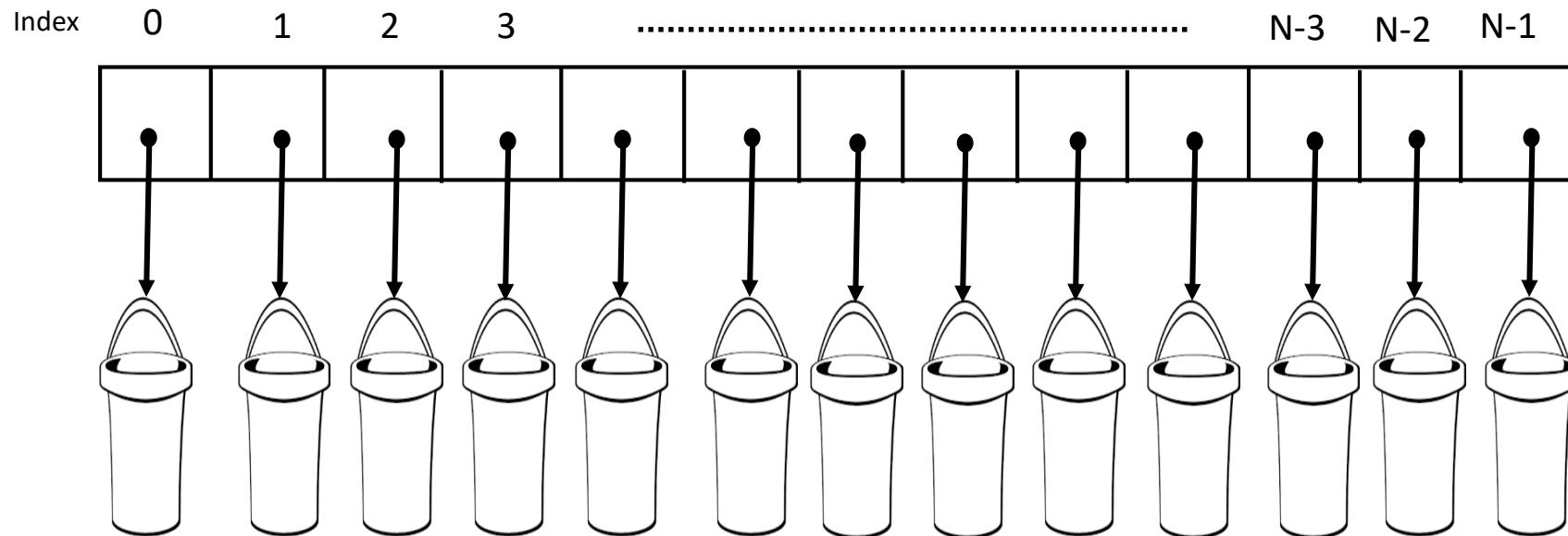
Slower (concatenation)

Miscellaneous

Dictionary – Bucket Array

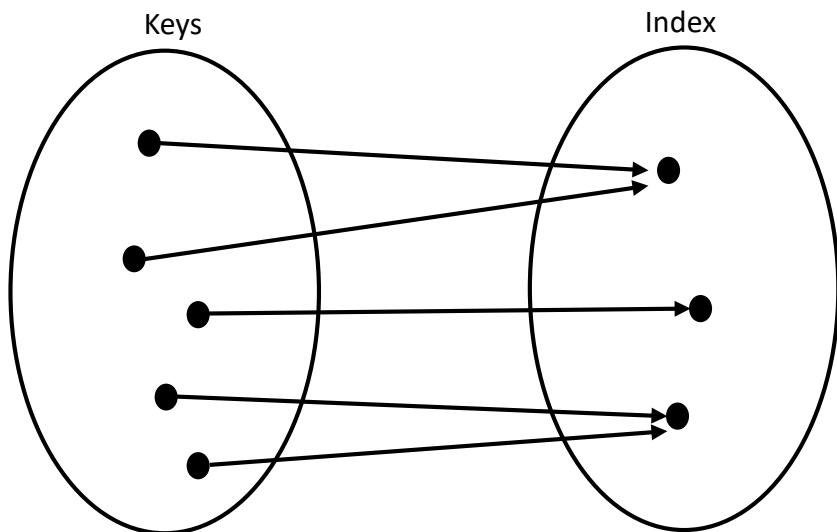
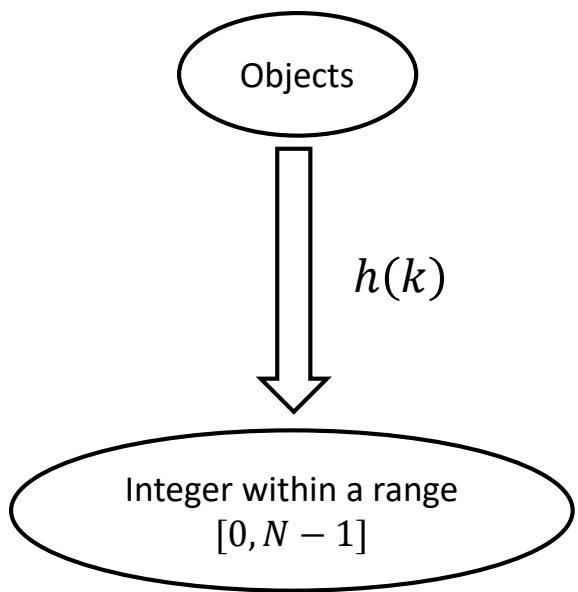
$$A(h(k)) = v$$

Key (k) is mapped to an index (0 to $N-1$) and Value (v) is stored in the corresponding bucket

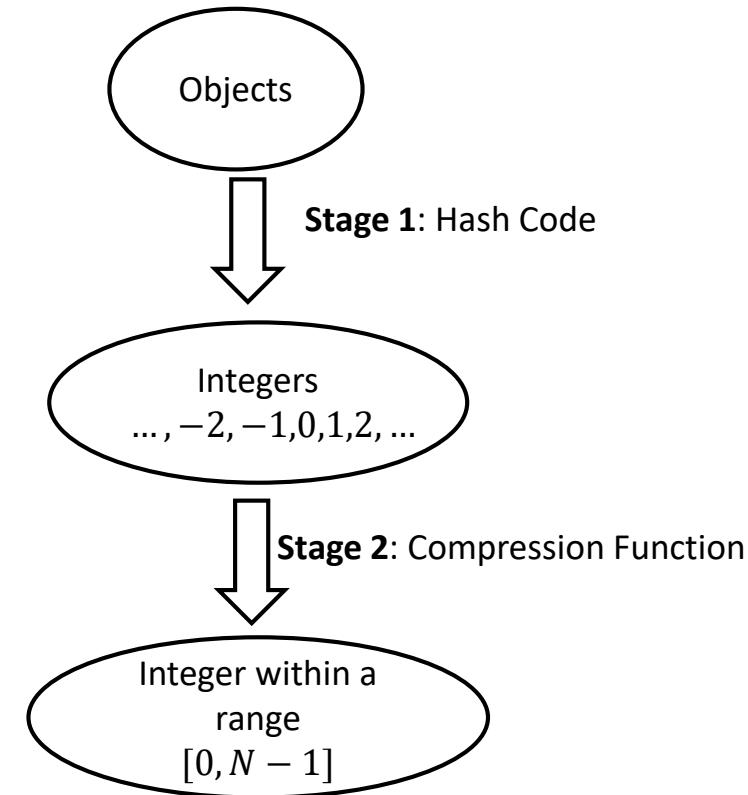
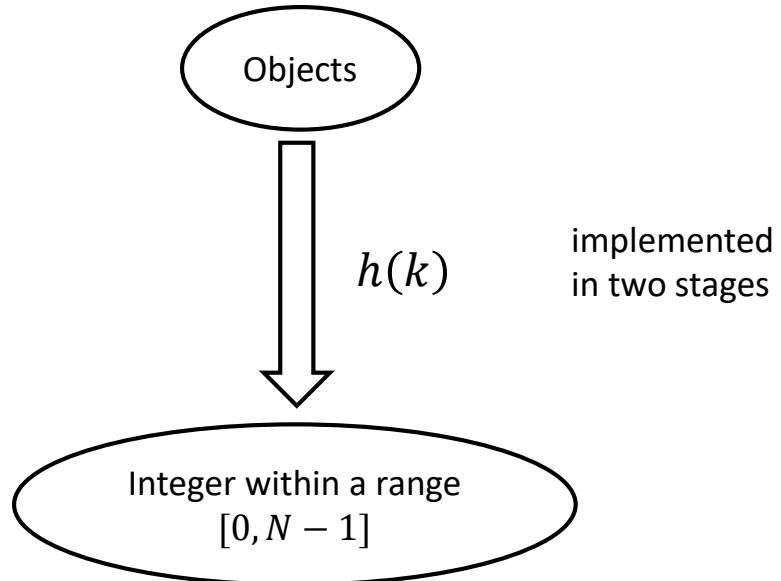


Key to index mapping is done through Hash Function ($h(k)$)

Hash Function - $h(k)$



Hash Function - $h(k)$



Hash Code

- Bit representation as hash codes
- Polynomial hash codes
 - $x_0a^{n-1} + x_1a^{n-2} + \dots + x_{n-2}a + x_{n-1}$
 - x_i : Coefficients
 - a : Constant
- Cyclic-shift hash codes

Compression Functions

- Division Method $i \bmod N$
- Multiply-Add-and-Divide (MAD) Method

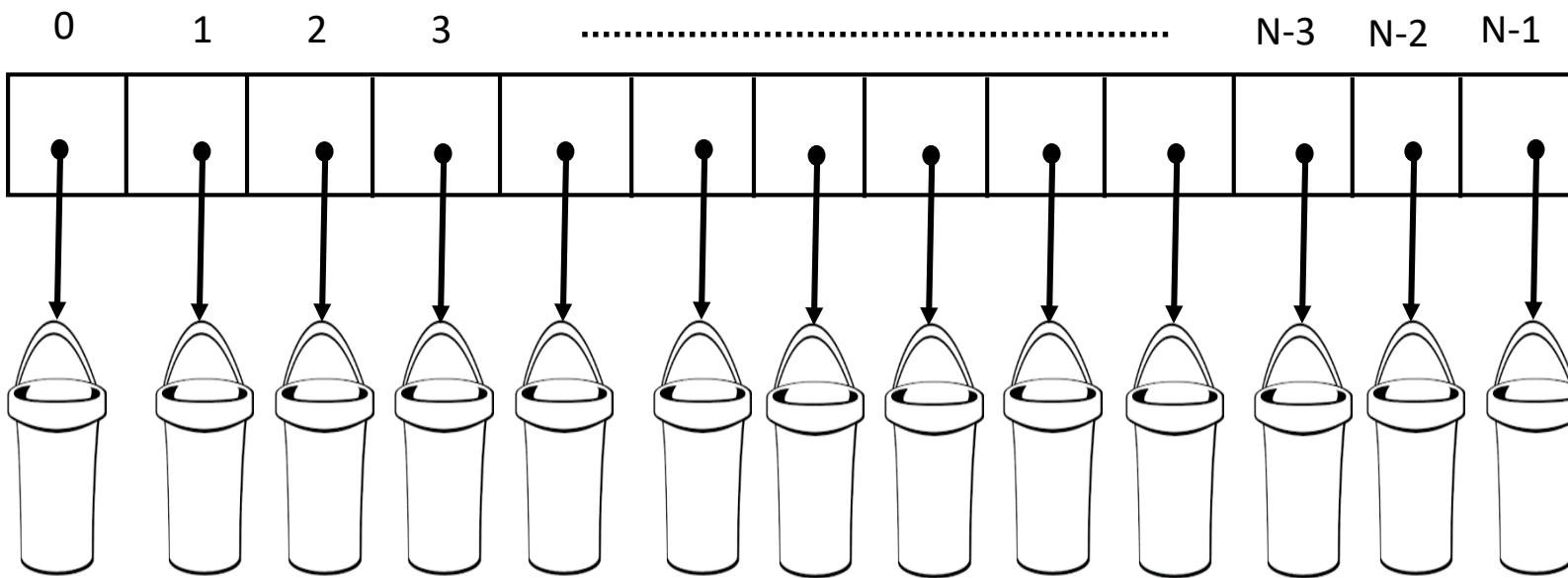
$$[(ai + b) \bmod p] \bmod N$$

Collisions

Collision occurs if multiple keys produce same hash value



$$h(k_1) = h(k_2)$$

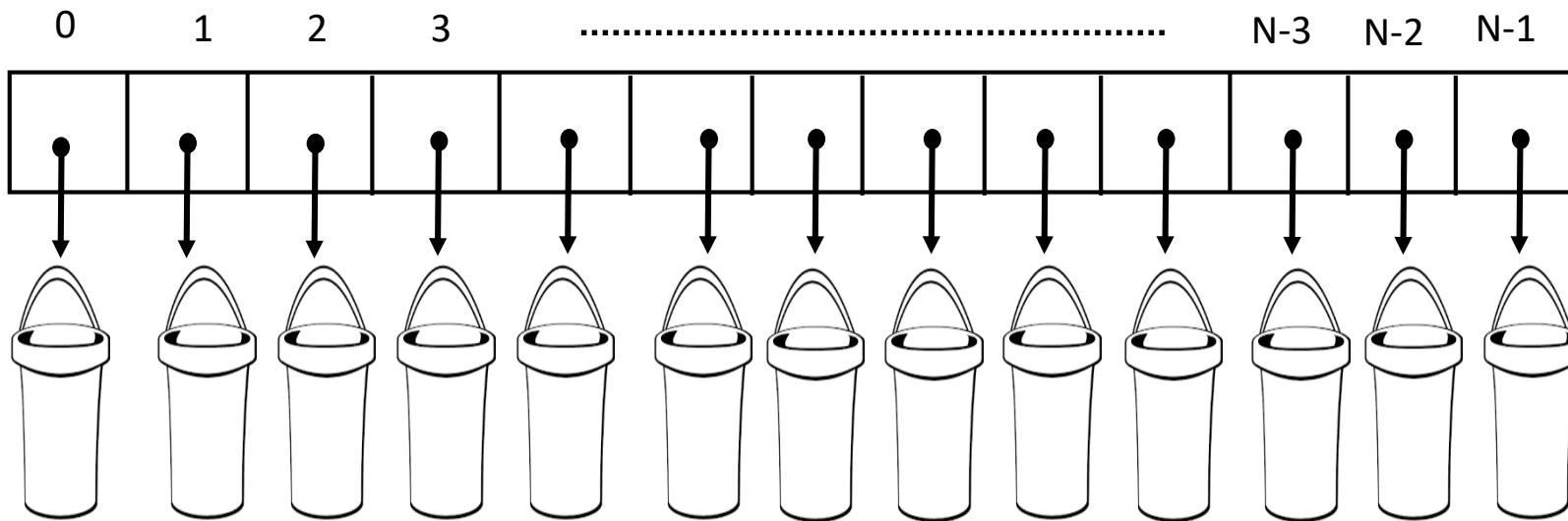


Collision-Handling

- Separate Chaining
- Open Addressing
 - Linear Probing
 - Quadratic Probing
 - Double Hashing

Collision-Handling

- Separate Chaining



- Each bucket contains a list of values v_i whose $h(k_i)$ are same
 - Requires additional *list* data structure
 - Slows down the access as the *list* need to be searched for the key

Open Addressing

- Linear Probing
- Quadratic Probing
- Double Hashing



NUS | Computing

National University
of Singapore

IT5001 Software Development Fundamentals

8. Anonymous Functions

Sirigina Rajendra Prasad

Functions

- Traditional Form
 - $\text{square}(x) \mapsto x^2$
- Anonymous Form:
 - $x \mapsto x^2$

Anonymous Functions

- Also known as lambda (λ) functions
 - Functions without name
- Python Syntax:
 - `lambda args: expression`

Functions: Traditional Maths vs λ -Calculus

Mathematics

- Abstraction
 - *square*: $x \mapsto x^2$
 - $\text{square}(x) = x^2$
- Application
 - $\text{square}(5) = 5^2 = 25$

λ -Calculus

- Abstraction
 - $x \mapsto x^2$
 - $\lambda x. x^2$
- Application
 - $(\lambda x. x^2)5 = 5^2 = 25$

Functions: λ -Calculus vs Python

λ -Calculus

- Abstraction
 - $x \mapsto x^2$
 - $\lambda x. x^2$
- Application
 - $(\lambda x. x^2)5 = 5^2 = 25$

Python

- Abstraction

```
>>> lambda x: x**2
```

```
<function <lambda> at 0x000002557882E708>
```

- Application

```
>>> (lambda x: x**2) (5)  
25
```

or

```
>>> f = lambda x: x**2+1  
>>> f(5)
```

26

→ Abstraction

→ Application

Examples: Identity Function

λ -Calculus

- λ -Calculus
 - $(\lambda x. x)$
 - Takes a variable x as input argument
 - returns x

Python

```
>>> lambda x:x
<function <lambda> at 0x000002557882E3A8>
>>> (lambda x:x) ('abc')
'abc'
>>> (lambda x:x) (10)
10
```

Examples: Constant Function

λ -Calculus

- λ -Calculus
 - $(\lambda x. y)$
 - Takes a variable x and return y

Python

```
>>> (lambda x: 'abc')(5)
'abc'
>>> (lambda x: 'abc')(10)
'abc'
```

Alternative in Python:

```
>>> (lambda : 10)()
10
>>> x = lambda : 10
>>> x()
10
```

Anonymous Functions: More Examples

- Can pass multiple arguments as inputs

```
>>> lambda x, y, z: x+y+z
<function <lambda> at 0x0000021EAA3B9DC8>
>>> (lambda x, y, z: x+y+z) (4, 5, 9)
18
```

```
>>> my_list = [1, 2, 3]
>>> (lambda x, y, z: x+y+z) (my_list)
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    (lambda x, y, z: x+y+z) (my_list)
TypeError: <lambda>() missing 2 required positional arguments: 'y' and 'z'
>>> (lambda my_list: my_list[0]+my_list[1]+my_list[2]) (my_list)
```



NUS | Computing

National University
of Singapore

IT5001 Software Development Fundamentals

9a. Higher Order Functions

Functions in Python

- Functions can be
 - Assigned to variables
 - Passed as arguments to functions
 - Returned from functions

“Callability”

- Normal variables are NOT **callable**

```
>>> x = 1
>>> x()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    x()
TypeError: 'int' object is not callable
```

- A function is **callable**

```
>>> def f():
        print("Hello")
```

```
>>> f()
Hello
```

Assignments

- Normal variables can store values

```
>>> x = 1  
>>> y = x  
>>> x = 2
```

- Can a variable store **a function**?!

```
>>> def f():  
    print("Hello")
```

```
>>> x = f  
>>> x()  
Hello
```

- Can!!!!!!

Assignments

- The **function** f is stored in the **variable** x
 - So x is a function, same as f

```
>>> def f():  
    print("Hello")
```

```
>>> x = f
```

```
>>> x()
```

Hello

See the difference

```
>>> def f2():
        return 999
```

With '()

```
>>> x = f2()
>>> print(x)
999 ← values
```

types

Without '()

```
>>> y = f2
>>> print(y)
<function f2 at 0x0000007ACE8C5A60>
>>> type(y)
<class 'function'>
```

Assigning to a variable

```
def inc_func(x):
    return x+1

my_func = inc_func

print(id(my_func) == id(inc_func))
print(f'ID of inc_func is: {id(inc_func)}')
print(f'ID of my_func is: {id(my_func)}')
print(f'inc_func(1) returns {inc_func(1)} ')
print(f'my_func(1) returns {my_func(1)} ')
```

Output:

```
True
ID of inc_func is: 1608860195432
ID of my_func is: 1608860195432
inc_func(1) returns 2
my_func(1) returns 2
```

Functions can be stored in variables

```
>>> from math import cos, sin, tan  
>>> f_1 = cos  
>>> f_1(0) ←  
1.0  
>>> print(f_1)  
<built-in function cos> ←
```

Equivalent
to $\cos(0)$

The type is
“function”

```
>>> def f():  
    print("Hello")  
>>> print(f)  
<function f at 0x000000F9F93F4950>
```

Functions as elements in Lists/Tuples

```
from math import cos, sin, tan
def inc_func(x):
    return x+1
my_list = [cos, sin, tan, inc_func, print]
x = my_list[3](1)
print(x)
```

Output:

2

```
from math import cos, sin, tan, pi
my_func_list = [cos, sin, tan]
theta = pi/3
output = [func(theta) for func in my_func_list]
print(output)
```

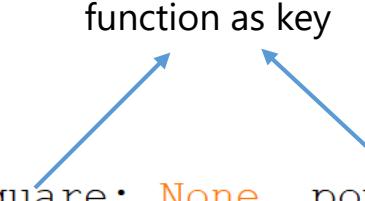
Output:

[0.5000000000000001, 0.8660254037844386, 1.7320508075688767] 9

Functions as elements in Dictionaries

```
def square(n):  
    return n**2  
  
def power(n, k):  
    return n**k  
  
my_func_dict = {square: None, power: 5}  
  
output = []  
for func, parameter in my_func_dict.items():  
    output.append(func(2) if parameter == None else func(2, parameter))  
  
print(output)
```

function as key



Functions as input arguments

```
from math import sqrt
```

```
def distance_1(x,y):  
    return sqrt(square(x[0]-y[0])+square(x[1]-y[1]))
```

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

Function calling
other function

```
def distance_2(x,y):  
    def square(x):  
        return x**2  
    return sqrt(square(x[0]-y[0])+square(x[1]-y[1]))
```

Nested function

```
def distance_3(x,y,square):  
    return sqrt(square(x[0]-y[0])+square(x[1]-y[1]))
```

```
x = (0,0)  
y = (2,2)
```

Function as input argument

```
print(distance_1(x,y))  
print(distance_2(x,y))  
print(distance_3(x,y,square))
```

Output: 2.8284271247461903

2.8284271247461903

2.8284271247461903

Functions that return functions

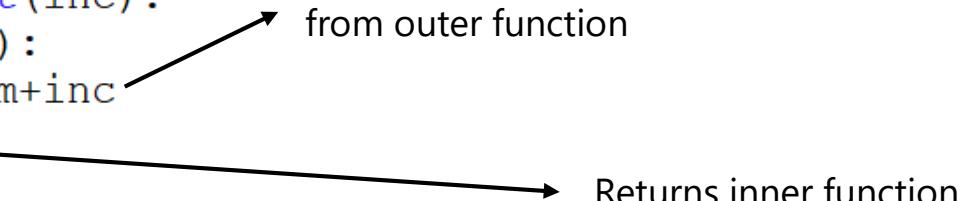
- Functions can return inner functions as output
- Inner functions serve many purposes
 - Closures
 - Decorators

Closures

- Closure:
 - Returns inner functions
 - Function plus the environment (state) in which they execute together
 - Preserve function state across function calls

- Example:

```
>>> def generate_increment(inc):           Access to variable  
    def increment(num):                  from outer function  
        return num+inc  
    return increment  
  
>>> increment_by_2 = generate_increment(2)  
>>> increment_by_2(10)  
12  
>>> increment_by_10 = generate_increment(10)  
>>> increment_by_10(23)  
33  
>>> increment_by_2(23)  
25
```



The diagram consists of two arrows. One arrow points from the text "Access to variable from outer function" to the line "return num+inc". Another arrow points from the text "Returns inner function" to the line "return increment".

Closures

- Create Functions to Power a Number

```
def make_power_func(n):
    return lambda x:x**n

square = make_power_func(2)
cube = make_power_func(3)
square_root = make_power_func(0.5)

>>> print(square(3))
9
>>> print(cube(2))
8
>>> print(square_root(16))
4.0
```

Decorators

- Decorator is a closure
 - Additionally, outer function accepts a function as input argument
- Modify input function's behaviour with an inner function without explicitly changing input function's code
- Example

```
def deco(func):  
    def wrapper():  
        #statements  
        func()  
        pass  
    return wrapper
```

```
def f():  
    pass  
  
f = deco(f)
```

```
def deco(func):  
    def wrapper():  
        #statements  
        func()  
        pass  
    return wrapper
```

```
@deco  
def f():  
    pass
```

Function Composition

- In math, we can do something like

$$\log(\sin(x))$$

```
>>> def f():
        print("Hello")
```

```
>>> def do_twice(x):
        x()
        x()
```

```
>>> do_twice(f)
Hello
Hello
```

Equivalent to

```
>>> def do_twice(x):
        f()
        f()
```

Mix and Match

```
>>> def add1to(x):  
        return x + 1
```

A function

```
>>> def square(x):  
        return x * x
```

A variable
(can be a
function
too!)

```
>>> def do_3_times(f, n):  
        return f(f(f(n)))
```

```
>>> do_3_times(add1to, 2)
```

5

```
>>> do_3_times(square, 2)
```

256

Equivalent to

```
>>> def do_3_times(f, n):  
        add1to(add1to(add1to(2)))
```

Examples

The “Powerful” Lambda

```
>>> def add1(x):  
        return x+1  
  
>>> add1(9)  
10  
>>> func = lambda x: x +1  
>>> func(9)  
10
```

```
>>> def aFunctionAddN(n):  
        return lambda x: x + n  
  
>>> f1 = aFunctionAddN(10)  
>>> f1(1)  
11  
>>> f1(2)  
12  
>>> f2 = aFunctionAddN(99)  
>>> f2(1)  
100  
>>> f2(f1(3))  
112
```

Agar Agar (Anyhow) Derivative

- We know that, given a function f , the derivative of f is

$$\frac{df(x)}{dx} = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

- But, if we have very small number dx

$$\frac{df(x)}{dx} \approx \frac{f(x + dx) - f(x)}{dx}$$

Agar Agar (Anyhow) Derivative

- We know that, given a function f , the derivative of f is

$$\frac{df(x)}{dx} = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

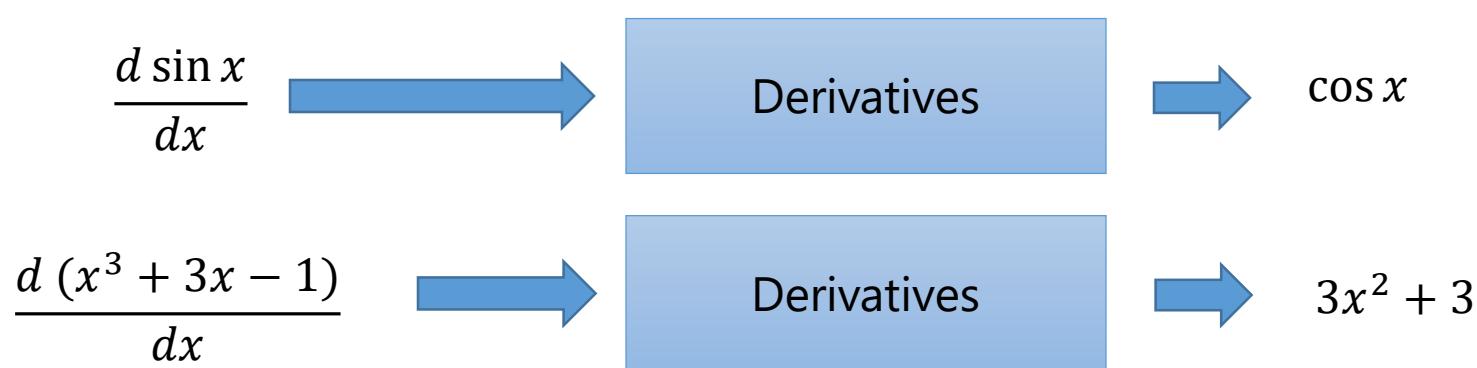
- $\frac{d \sin x}{dx} = \cos x$

- $\frac{d (x^3 + 3x - 1)}{dx} = 3x^2 + 3$

Agar Agar (Anyhow) Derivative

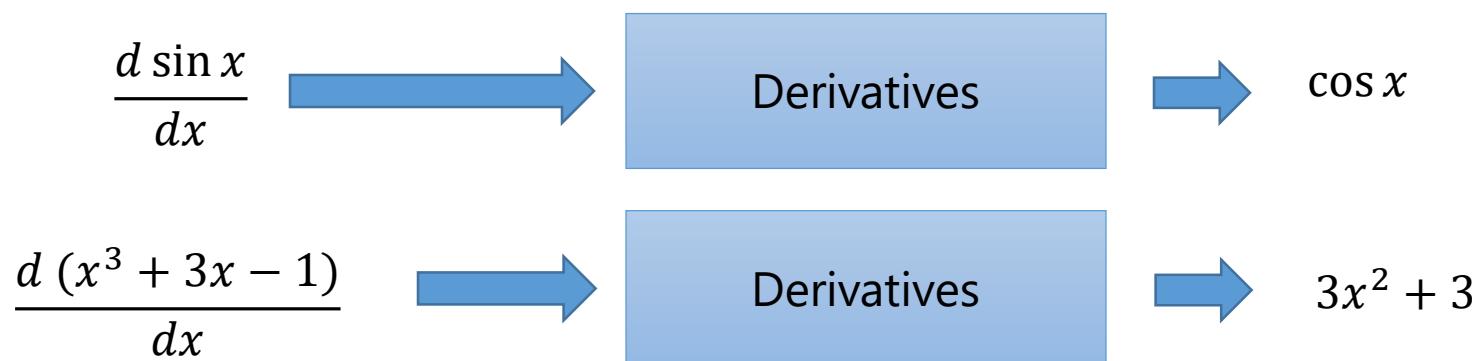
- We know that, given a function f , the derivative of f is

$$\frac{df(x)}{dx} = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$



Agar Agar (Anyhow) Derivative

- Its input is a function
 - And output another function



Agar Agar (Anyhow) Derivative

- We know that, given a function f , the derivative of f is

$$\frac{df(x)}{dx} = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

- But, if we have very small number dx

$$\frac{df(x)}{dx} \approx \frac{f(x + dx) - f(x)}{dx}$$

Take in a function,
returning another
function

```
>>> def deriv(f):
    dx = 0.000000001
    return lambda x: (f(x+dx)-f(x))/dx

>>> cos(0.123)
0.9924450321351935
>>> func = deriv(sin)
>>> func(0.123)
0.9924450428133723
```

Agar Agar (Anyhow) Derivative

- We know that, given a function f , the derivative of f is

$$\frac{df(x)}{dx} = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

- But, if we have very small number dx

$$\frac{df(x)}{dx} \approx \frac{f(x + dx) - f(x)}{dx}$$

```
>>> def f(x):  
    return x**3+3*x-1  
  
>>> deriv(f)(9)  
246.00001324870388  
>>> x = 9  
>>> 3*x**2 +3  
246
```

Agar Agar (Anyhow) Derivative

```
>>> def deriv(f):
    dx = 0.000000001
    return lambda x: (f(x+dx)-f(x))/dx

>>> cos(0.123)
0.9924450321351935
>>> func = deriv(sin)
>>> func(0.123)
0.9924450428133723

>>> def f(x):
    return x**3+3*x-1

>>> deriv(f)(9)
246.00001324870388
>>> x = 9
>>> 3*x**2 +3
246
```

Take in a function,
returning another
function

Agar Agar (Anyhow) Derivative

```
>>> def deriv(f):
    dx = 0.000000001
    return lambda x: (f(x+dx)-f(x))/dx

>>> cos(0.123)
0.9924450321351935
>>> func = deriv(sin)
>>> func(0.123)
0.9924450428133723
```

$$\frac{d \sin x}{dx}$$

Derivatives

$$\cos x$$

$$\frac{d (x^3 + 3x - 1)}{dx}$$

Derivatives

$$3x^2 + 3$$

Application Example of deriv()

Example: Newton's method

- To compute root of function $g(x)$, i.e. find x such that $g(x) = 0$
1. Anyhow assume the answer $x = \text{something}$
 2. If $g(x) \approx 0$ then stop: answer is x , return x
 3. Otherwise
 - $x = x - g(x)/\text{deriv}(x)$
 4. Go to step 2

Example: Newton's method

- To compute root of function $g(x)$, i.e. find x such that $g(x) = 0$
1. Anyhow assume the answer $x = \text{something}$
 2. If $g(x) \approx 0$ then stop: answer is x , return x
 3. Otherwise
 - $x = x - g(x)/\text{deriv}(x)$
 4. Go to step 2

```
def newtonM(g):  
    x = 999 #doesn't matter  
    err = 0.0000000001  
    while (abs(g(x))>err):  
        x = x - g(x)/deriv(g)(x)  
    return x
```

Example: Newton's method

- To compute the root of function $g(x)$, i.e. find x such that $g(x) = 0$

```
def deriv(f):
    dx = 0.000000001
    return lambda x: (f(x+dx)-f(x))/dx

def newtonM(g):
    x = 999 #doesnt matter
    err = 0.0000000001
    while(abs(g(x))>err):
        x = x - g(x)/deriv(g)(x)
    return x
```

Example: Newton's method

- Example: Square root of a number A
➤ It's equivalent to solve the equation: $x^2 - A = 0$

```
>>> def my_own_sqrt(A):
    return newtonM(lambda x:x*x-A)

>>> x = my_own_sqrt(10)
>>> x * x
9.99999999999998
```

Example: Newton's method

- Example: Compute $\log_{10}(A)$
 - Solve the equation: $10^x - A = 0$

```
>>> def my_own_log10(N):
    return newtonM(lambda x: 10**x - N)

>>> my_own_log10(100)
2.0000000000000013
>>> x = my_own_log10(234)
>>> 10 ** x
234.000000000000892
```

```
>>> def my_own_log10(N):
    return newtonM(lambda x: 10**x - N)

>>> my_own_log10(100)
2.000000000000013
>>> x = my_own_log10(234)
>>> 10 ** x
234.00000000000892
```

You can solve any equation!

.... that Newton Method can solve.



NUS | Computing

National University
of Singapore

IT5001 Software Development Fundamentals

9a. Higher Order Functions

Functions in Python

- Functions can be
 - Assigned to variables
 - Passed as arguments to functions
 - Returned from functions

“Callability”

- Normal variables are NOT **callable**

```
>>> x = 1
>>> x()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    x()
TypeError: 'int' object is not callable
```

- A function is **callable**

```
>>> def f():
        print("Hello")
```

```
>>> f()
Hello
```

Assignments

- Normal variables can store values

```
>>> x = 1  
>>> y = x  
>>> x = 2
```

- Can a variable store **a function**?!

```
>>> def f():  
    print("Hello")
```

```
>>> x = f  
>>> x()  
Hello
```

- Can!!!!!!

Assignments

- The **function** f is stored in the **variable** x
 - So x is a function, same as f

```
>>> def f():  
    print("Hello")
```

```
>>> x = f
```

```
>>> x()
```

Hello

See the difference

```
>>> def f2():
        return 999
```

With '()

```
>>> x = f2()
>>> print(x)
999 ← values
```

types

Without '()

```
>>> y = f2
>>> print(y)
<function f2 at 0x0000007ACE8C5A60>
>>> type(y)
<class 'function'>
```

Assigning to a variable

```
def inc_func(x):
    return x+1

my_func = inc_func

print(id(my_func) == id(inc_func))
print(f'ID of inc_func is: {id(inc_func)}')
print(f'ID of my_func is: {id(my_func)}')
print(f'inc_func(1) returns {inc_func(1)} ')
print(f'my_func(1) returns {my_func(1)} ')
```

Output:

```
True
ID of inc_func is: 1608860195432
ID of my_func is: 1608860195432
inc_func(1) returns 2
my_func(1) returns 2
```

Functions can be stored in variables

```
>>> from math import cos, sin, tan  
>>> f_1 = cos  
>>> f_1(0) ←  
1.0  
>>> print(f_1)  
<built-in function cos> ←
```

Equivalent
to $\cos(0)$

The type is
“function”

```
>>> def f():  
    print("Hello")  
>>> print(f)  
<function f at 0x000000F9F93F4950>
```

Functions as elements in Lists/Tuples

```
from math import cos, sin, tan
def inc_func(x):
    return x+1
my_list = [cos, sin, tan, inc_func, print]
x = my_list[3](1)
print(x)
```

Output:

2

```
from math import cos, sin, tan, pi
my_func_list = [cos, sin, tan]
theta = pi/3
output = [func(theta) for func in my_func_list]
print(output)
```

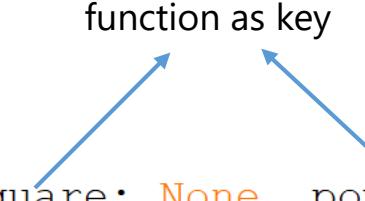
Output:

[0.5000000000000001, 0.8660254037844386, 1.7320508075688767] 9

Functions as elements in Dictionaries

```
def square(n):  
    return n**2  
  
def power(n, k):  
    return n**k  
  
my_func_dict = {square: None, power: 5}  
  
output = []  
for func, parameter in my_func_dict.items():  
    output.append(func(2) if parameter == None else func(2, parameter))  
  
print(output)
```

function as key



Functions as input arguments

```
from math import sqrt
```

```
def distance_1(x,y):  
    return sqrt(square(x[0]-y[0])+square(x[1]-y[1]))
```

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

Function calling
other function

```
def distance_2(x,y):  
    def square(x):  
        return x**2  
    return sqrt(square(x[0]-y[0])+square(x[1]-y[1]))
```

Nested function

```
def distance_3(x,y,square):  
    return sqrt(square(x[0]-y[0])+square(x[1]-y[1]))
```

```
x = (0,0)  
y = (2,2)
```

Function as input argument

```
print(distance_1(x,y))  
print(distance_2(x,y))  
print(distance_3(x,y,square))
```

Output: 2.8284271247461903
2.8284271247461903
2.8284271247461903

Functions that return functions

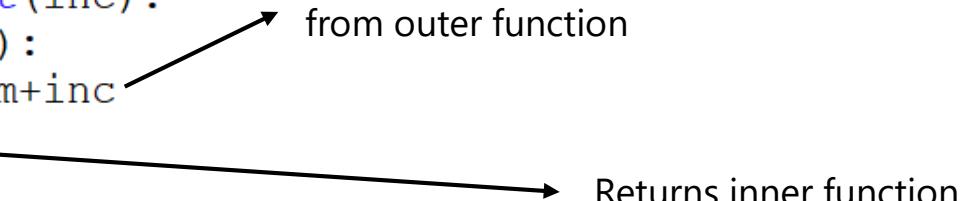
- Functions can return inner functions as output
- Inner functions serve many purposes
 - Closures
 - Decorators

Closures

- Closure:
 - Returns inner functions
 - Function plus the environment (state) in which they execute together
 - Preserve function state across function calls

- Example:

```
>>> def generate_increment(inc):           Access to variable  
    def increment(num):                  from outer function  
        return num+inc  
    return increment  
  
>>> increment_by_2 = generate_increment(2)  
>>> increment_by_2(10)  
12  
>>> increment_by_10 = generate_increment(10)  
>>> increment_by_10(23)  
33  
>>> increment_by_2(23)  
25
```



The diagram consists of two arrows. One arrow points from the annotation "Access to variable from outer function" to the line "return num+inc". Another arrow points from the annotation "Returns inner function" to the line "return increment".

Closures

- Create Functions to Power a Number

```
def make_power_func(n):
    return lambda x:x**n

square = make_power_func(2)
cube = make_power_func(3)
square_root = make_power_func(0.5)

>>> print(square(3))
9
>>> print(cube(2))
8
>>> print(square_root(16))
4.0
```

Decorators

- Decorator is a closure
 - Additionally, outer function accepts a function as input argument
- Modify input function's behaviour with an inner function without explicitly changing input function's code
- Example

```
def deco(func):  
    def wrapper():  
        #statements  
        func()  
        pass  
    return wrapper
```

```
def f():  
    pass  
  
f = deco(f)
```

```
def deco(func):  
    def wrapper():  
        #statements  
        func()  
        pass  
    return wrapper
```

```
@deco  
def f():  
    pass
```

Function Composition

- In math, we can do something like

$$\log(\sin(x))$$

```
>>> def f():
        print("Hello")
```

```
>>> def do_twice(x):
        x()
        x()
```

```
>>> do_twice(f)
Hello
Hello
```

Equivalent to

```
>>> def do_twice(x):
        f()
        f()
```

Mix and Match

```
>>> def add1to(x):  
        return x + 1
```

A function

```
>>> def square(x):  
        return x * x
```

A variable
(can be a
function
too!)

```
>>> def do_3_times(f, n):  
        return f(f(f(n)))
```

```
>>> do_3_times(add1to, 2)
```

5

```
>>> do_3_times(square, 2)
```

256

Equivalent to

```
>>> def do_3_times(f, n):  
        add1to(add1to(add1to(2)))
```

Examples

The “Powerful” Lambda

```
>>> def add1(x):  
        return x+1  
  
>>> add1(9)  
10  
>>> func = lambda x: x +1  
>>> func(9)  
10
```

```
>>> def aFunctionAddN(n):  
        return lambda x: x + n  
  
>>> f1 = aFunctionAddN(10)  
>>> f1(1)  
11  
>>> f1(2)  
12  
>>> f2 = aFunctionAddN(99)  
>>> f2(1)  
100  
>>> f2(f1(3))  
112
```

Agar Agar (Anyhow) Derivative

- We know that, given a function f , the derivative of f is

$$\frac{df(x)}{dx} = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

- But, if we have very small number dx

$$\frac{df(x)}{dx} \approx \frac{f(x + dx) - f(x)}{dx}$$

Agar Agar (Anyhow) Derivative

- We know that, given a function f , the derivative of f is

$$\frac{df(x)}{dx} = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

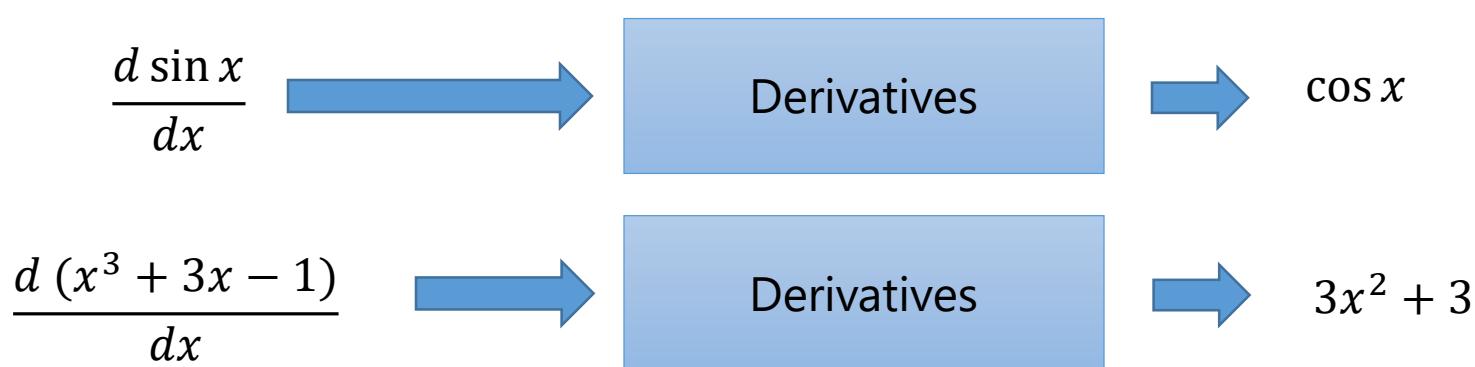
- $\frac{d \sin x}{dx} = \cos x$

- $\frac{d (x^3 + 3x - 1)}{dx} = 3x^2 + 3$

Agar Agar (Anyhow) Derivative

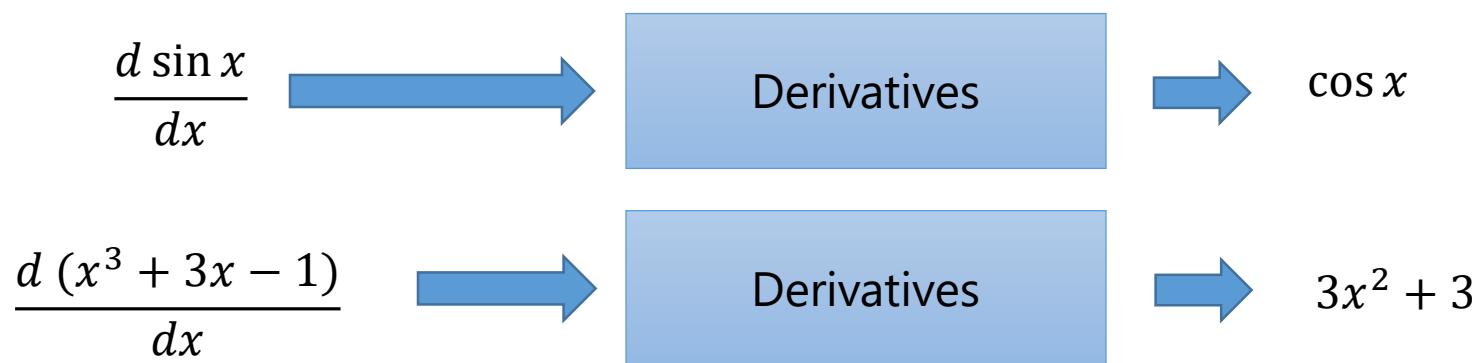
- We know that, given a function f , the derivative of f is

$$\frac{df(x)}{dx} = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$



Agar Agar (Anyhow) Derivative

- Its input is a function
 - And output another function



Agar Agar (Anyhow) Derivative

- We know that, given a function f , the derivative of f is

$$\frac{df(x)}{dx} = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

- But, if we have very small number dx

$$\frac{df(x)}{dx} \approx \frac{f(x + dx) - f(x)}{dx}$$

Take in a function,
returning another
function

```
>>> def deriv(f):
    dx = 0.000000001
    return lambda x: (f(x+dx)-f(x))/dx

>>> cos(0.123)
0.9924450321351935
>>> func = deriv(sin)
>>> func(0.123)
0.9924450428133723
```

Agar Agar (Anyhow) Derivative

- We know that, given a function f , the derivative of f is

$$\frac{df(x)}{dx} = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

- But, if we have very small number dx

$$\frac{df(x)}{dx} \approx \frac{f(x + dx) - f(x)}{dx}$$

```
>>> def f(x):  
    return x**3+3*x-1  
  
>>> deriv(f)(9)  
246.00001324870388  
>>> x = 9  
>>> 3*x**2 +3  
246
```

Agar Agar (Anyhow) Derivative

```
>>> def deriv(f):
    dx = 0.000000001
    return lambda x: (f(x+dx)-f(x))/dx

>>> cos(0.123)
0.9924450321351935
>>> func = deriv(sin)
>>> func(0.123)
0.9924450428133723

>>> def f(x):
    return x**3+3*x-1

>>> deriv(f)(9)
246.00001324870388
>>> x = 9
>>> 3*x**2 +3
246
```



Take in a function,
returning another
function

Agar Agar (Anyhow) Derivative

```
>>> def deriv(f):
    dx = 0.000000001
    return lambda x: (f(x+dx)-f(x))/dx

>>> cos(0.123)
0.9924450321351935
>>> func = deriv(sin)
>>> func(0.123)
0.9924450428133723
```

$$\frac{d \sin x}{dx}$$

Derivatives

$$\cos x$$

$$\frac{d (x^3 + 3x - 1)}{dx}$$

Derivatives

$$3x^2 + 3$$

Application Example of deriv()

Example: Newton's method

- To compute root of function $g(x)$, i.e. find x such that $g(x) = 0$
1. Anyhow assume the answer $x = \text{something}$
 2. If $g(x) \approx 0$ then stop: answer is x , return x
 3. Otherwise
 - $x = x - g(x)/\text{deriv}(x)$
 4. Go to step 2

Example: Newton's method

- To compute root of function $g(x)$, i.e. find x such that $g(x) = 0$
1. Anyhow assume the answer $x = \text{something}$
 2. If $g(x) \approx 0$ then stop: answer is x , return x
 3. Otherwise
 - $x = x - g(x)/\text{deriv}(x)$
 4. Go to step 2

```
def newtonM(g):  
    x = 999 #doesn't matter  
    err = 0.0000000001  
    while (abs(g(x))>err):  
        x = x - g(x)/deriv(g)(x)  
    return x
```

Example: Newton's method

- To compute the root of function $g(x)$, i.e. find x such that $g(x) = 0$

```
def deriv(f):
    dx = 0.000000001
    return lambda x: (f(x+dx)-f(x))/dx

def newtonM(g):
    x = 999 #doesnt matter
    err = 0.0000000001
    while(abs(g(x))>err):
        x = x - g(x)/deriv(g)(x)
    return x
```

Example: Newton's method

- Example: Square root of a number A
➤ It's equivalent to solve the equation: $x^2 - A = 0$

```
>>> def my_own_sqrt(A):
    return newtonM(lambda x:x*x-A)

>>> x = my_own_sqrt(10)
>>> x * x
9.99999999999998
```

Example: Newton's method

- Example: Compute $\log_{10}(A)$
 - Solve the equation: $10^x - A = 0$

```
>>> def my_own_log10(N):
    return newtonM(lambda x: 10**x - N)

>>> my_own_log10(100)
2.0000000000000013
>>> x = my_own_log10(234)
>>> 10 ** x
234.000000000000892
```

```
>>> def my_own_log10(N):
    return newtonM(lambda x: 10**x - N)

>>> my_own_log10(100)
2.000000000000013
>>> x = my_own_log10(234)
>>> 10 ** x
234.00000000000892
```

You can solve any equation!

.... that Newton Method can solve.

Lambda functions

```
>>> f = lambda a, b: lambda x: b(b(a))  
>>> f('b', lambda a: a * 3)(lambda a: a[:1])
```

Lambda functions

```
>>> f = lambda a, b: lambda x: b(b(a))
```

Abstraction is right associative

$$f = (\text{lambda } a, b: (\text{lambda } x: b(b(a))))$$

```
>>> f('b', lambda a: a * 3)(lambda a: a[:1])
```

Application is left associative

$$(f('b', \text{lambda } a: a^*3))(\text{lambda } a: a[:1])$$

Lambda functions

$f = (\lambda a, b: (\lambda x: b(b(a))))$

$(f('b', \lambda a: a^3))(\lambda a: a[:1])$

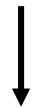
$\lambda x: ((\lambda a: a^3)(\lambda a: a^3)('b'))$

$\lambda x: ((\lambda a: a^3) ('bbb'))$

$\lambda x: ('bbbbbbbb')$

$\left(f('b', \text{lambda } a: a^*3) \right) \left((\text{lambda } a: a[:1]) \right)$

$(\text{lambda } x: 'bbbbbbbb') (\text{lambda } a: a[:1])$



'bbbbbbbb'

(Text) File Input/Output





Data.gov.sg

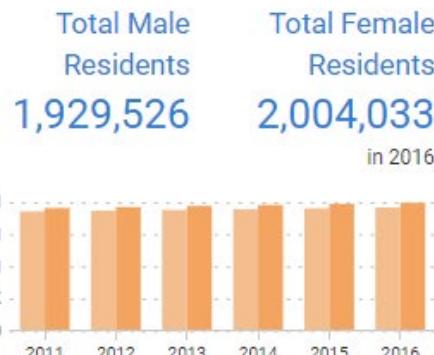
Search Singapore's Public Data

e.g. "rainfall", "gross domestic product", "transport"



Singapore at a glance

Singapore Residents By Gender, End June,
Annual - Data



Crude Birth Rate - Data

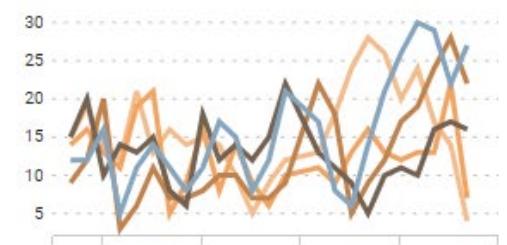
9.4

Per Thousand Population in 2016



1-hr PM2.5 Readings (Past 24 Hrs)

	west	east	central	south	north
µg/m ³ as of 6/9/2017 10AM	4	7	22	16	27





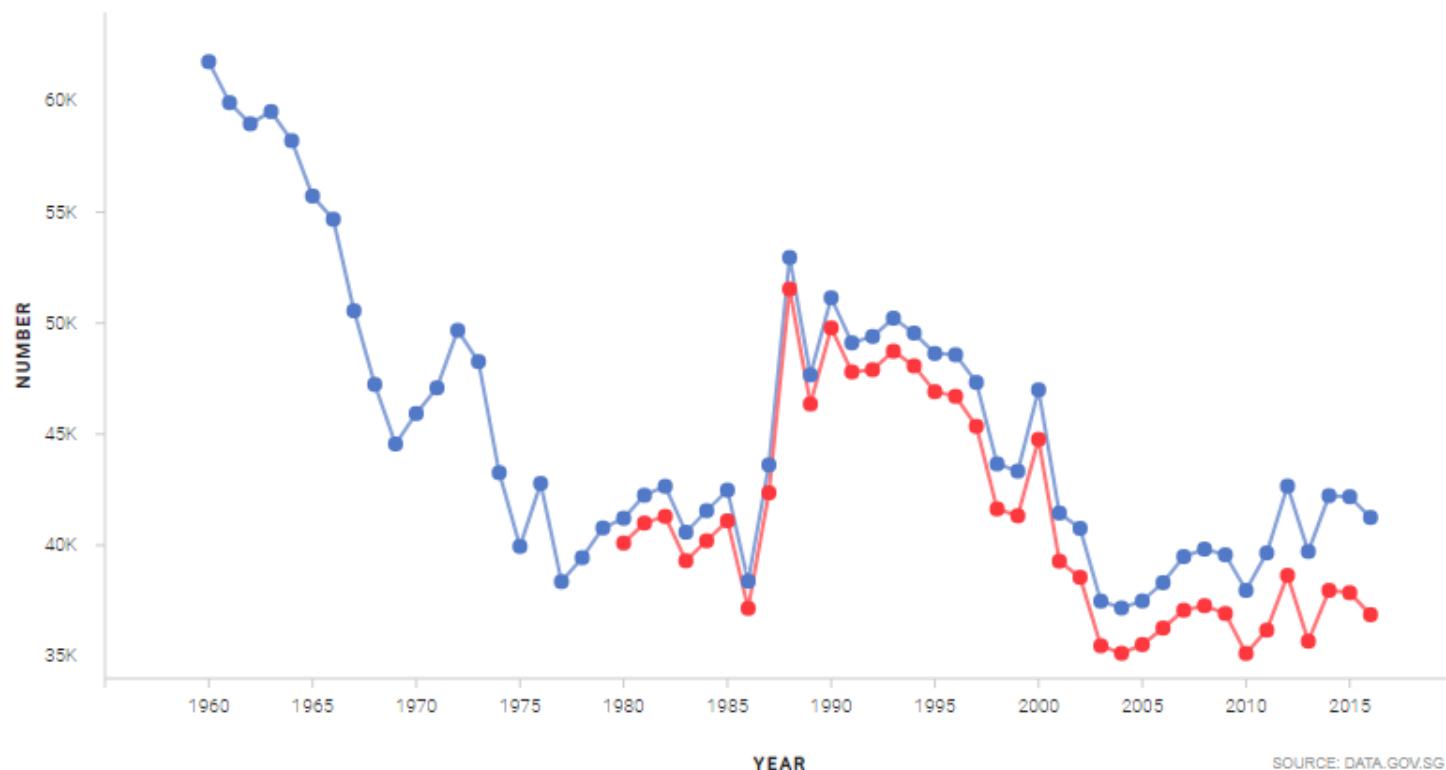
Live-Births

● Total Live-Births ● Resident Live-Births

Crude Birth Rate

Total Fertility Rate and
Reproduction Rate

Age-Specific Fertility Rate

Total Fertility Rate by
Ethnic Group

SOURCE: DATA.GOV.SG

Live-Births

Chart

Embed View

Births and Fertility, Annual

Managed by Ministry of Trade and Industry - Department of Statistics

A summary of key indicators measuring births and fertility in Singapore.

Download

You can
download any
data!

Open in

- Excel
- Notepad

	A	B	C	D	E
1	year	level_1	value		
2	1960	Crude Birth Rate	37.5		
3	1961	Crude Birth Rate	35.2		
4	1962	Crude Birth Rate	33.7		
5	1963	Crude Birth Rate	33.2		
6	1964	Crude Birth Rate	31.6		
7	1965	Crude Birth Rate	29.5		
8	1966	Crude Birth Rate	28.3		
9	1967	Crude Birth Rate	25.6		
10	1968	Crude Birth Rate	23.5		
11	1969	Crude Birth Rate	21.8		
12	1970	Crude Birth Rate	22.1		
13	1971	Crude Birth Rate	22.3		
14	1972	Crude Birth Rate	23.1		
15	1973	Crude Birth Rate	22		
16	1974	Crude Birth Rate	19.4		
17	1975	Crude Birth Rate	17.7		
18	1976	Crude Birth Rate	18.7		
19	1977	Crude Birth Rate	16.5		
20	1978	Crude Birth Rate	16.8		
21	1979	Crude Birth Rate	17.1		
22	1980	Crude Birth Rate	17.6		
23	1981	Crude Birth Rate	17.6		
24	1982	Crude Birth Rate	17.5		
25	1983	Crude Birth Rate	16.2		

```
year,level_1,value
1960,Crude Birth Rate,37.5
1961,Crude Birth Rate,35.2
1962,Crude Birth Rate,33.7
1963,Crude Birth Rate,33.2
1964,Crude Birth Rate,31.6
1965,Crude Birth Rate,29.5
1966,Crude Birth Rate,28.3
1967,Crude Birth Rate,25.6
1968,Crude Birth Rate,23.5
1969,Crude Birth Rate,21.8
1970,Crude Birth Rate,22.1
1971,Crude Birth Rate,22.3
1972,Crude Birth Rate,23.1
1973,Crude Birth Rate,22
1974,Crude Birth Rate,19.4
1975,Crude Birth Rate,17.7
1976,Crude Birth Rate,18.7
1977,Crude Birth Rate,16.5
1978,Crude Birth Rate,16.8
1979,Crude Birth Rate,17.1
1980,Crude Birth Rate,17.6
1981,Crude Birth Rate,17.6
1982,Crude Birth Rate,17.5
1983,Crude Birth Rate,16.3
1984,Crude Birth Rate,16.5
1985,Crude Birth Rate,16.6
1986,Crude Birth Rate,14.8
1987,Crude Birth Rate,16.6
1988,Crude Birth Rate,19.8
1989,Crude Birth Rate,17.5
1990,Crude Birth Rate,18.2
1991,Crude Birth Rate,17.1
1992,Crude Birth Rate,16.8
1993,Crude Birth Rate,16.8
1994,Crude Birth Rate,16.2
```

Let's Do it in Python

- Of course, you are **not** going to type the data into your Python code
 - one data one code?!
 - change in data = change in code?
 - Called “Hard Coding”
- Usually practice
 - Data file +
 - Python code that can read the file



Writing A File

Actually Easier

Writing A File

```
def write_something():
    with open('my_file.txt', 'w') as f:
        f.write('This is my first line')
        f.write('This is my second line')
```

```
write_something()
```

Indicate the file object f is for writing

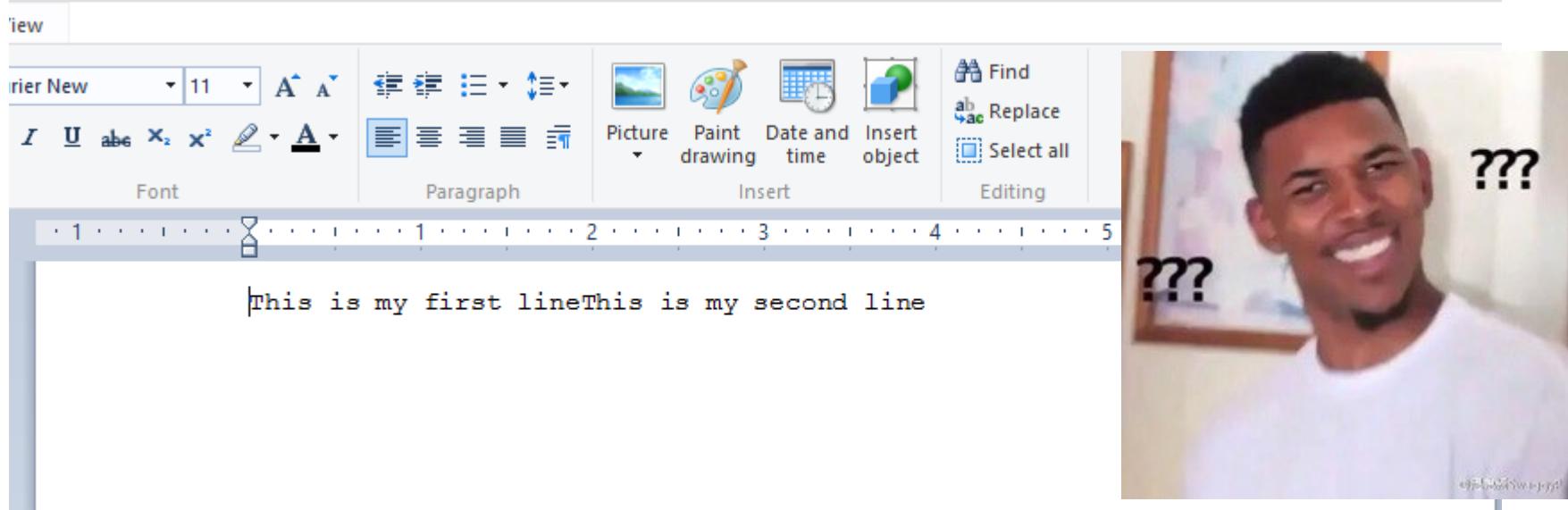
The file object that we called it “f” (can be any variable name)

Use the file “f” to write something in it

Writing A File

```
def write_something():
    with open('my_file.txt','w') as f:
        f.write('This is my first line')
        f.write('This is my second line')
```

```
write_something()
```

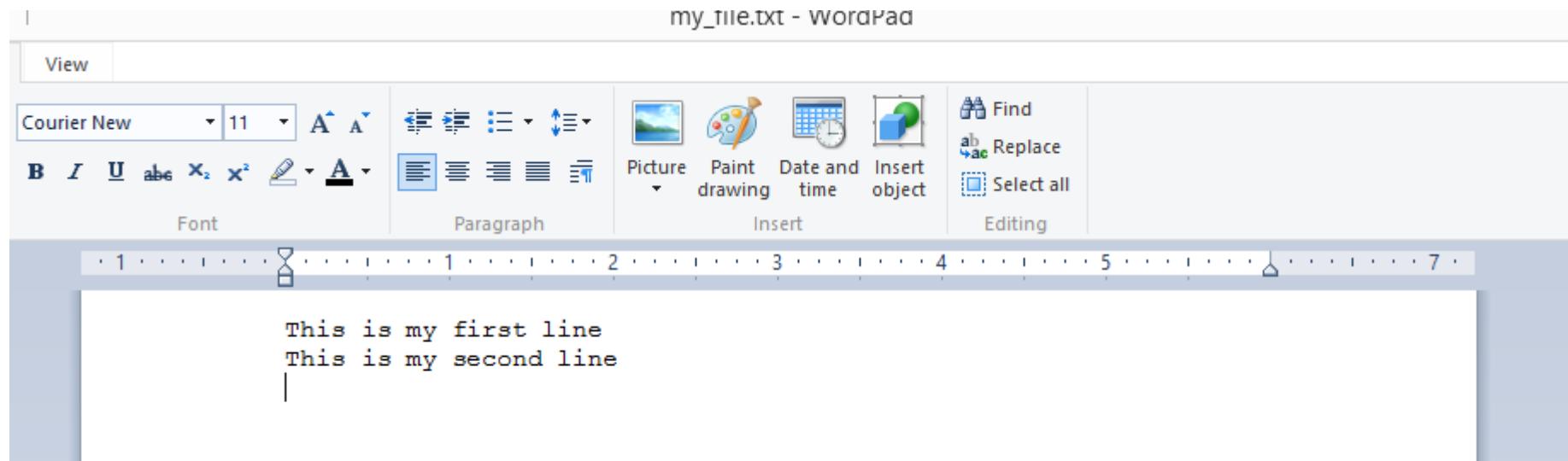


Writing a File

```
def write_something():
    with open('my_file.txt', 'w') as f:
        f.write('This is my first line'+'\n')
        f.write('This is my second line'+'\n')

write_something()
```

The newline character



Different File Opening Modes

```
def write_something():
    with open('my_file.txt', 'w') as f:
        f.write('This is my first line'+'\n')
        f.write('This is my second line'+'\n')
```

write_something()

Modes	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.

Different File Opening Modes

- Default is text format
- Storing in text mode is very space consuming
- E.g. storing the date '20180901'
 - Text (ASCII):
 - 50 48 49 56 48
57 48 49
 - Binary (Integer):
 - 01 33 EF A5

wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

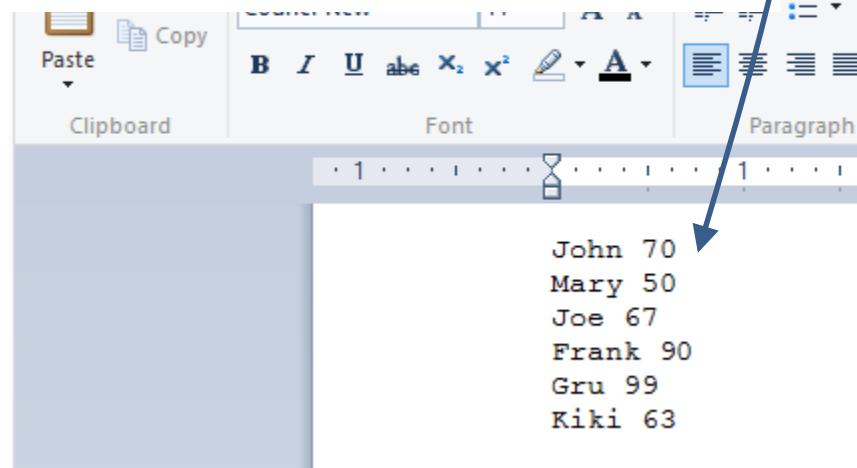
Reading a File

Try it out

- I have a text file called “student_marks.txt”

```
>>> with open('student_marks.txt') as f:  
     data = f.read()  
  
The file object  
read the whole file into  
“data” as a string
```

```
>>> data  
'John 70\nMary 50\nJoe 67\nFrank 90\nGru 99\nKiki 63'  
The new line character
```



String Operation Split

- Use the function split to separate the string into a list of strings by a separator

```
>>> data  
'John 70\nMary 50\nJoe 67\nFrank 90\nGru 99\nKiki 63'  
>>>  
>>> data.split()  
['John', '70', 'Mary', '50', 'Joe', '67', 'Frank', '90',  
'Gru', '99', 'Kiki', '63']
```

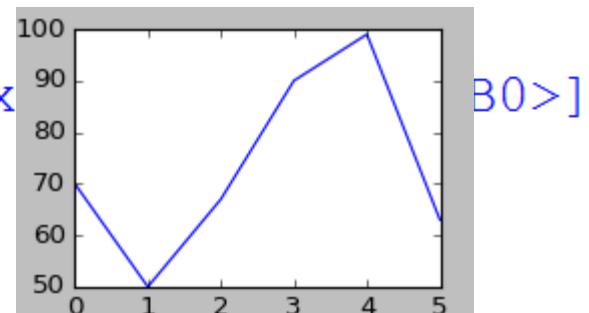
- If you do not put any argument for split(), the default separators are space and newline

Try it out

Starting from the second position and step two

- Extract all the scores

```
>>> data.split()  
['John', '70', 'Mary', '50', 'Joe', '67', 'Frank', '90',  
 'Gru', '99', 'Kiki', '63']  
>>> max(data.split())  
'Mary'  
>>> all_score = [int(i) for i in data.split()[1::2]]  
>>> all_score  
[70, 50, 67, 90, 99, 63]  
>>> max(all_score)  
99  
>>> plt.plot(all_score)  
[<matplotlib.lines.Line2D object at 0x...>]  
>>> plt.show()
```



Reading One Whole File into a String

- That' is not “healthy”
- Your file can be a few MB or even GB

```
>>> with open('student_marks.txt') as f:  
    data = f.read()
```

- Then this line of code will run in a very long time, may even end in crashing the whole program or even the system
- Better way to do is to read the file line-by-line

Reading the File Line-by-line

```
def read_line_by_line():
    with open('student_marks.txt', 'r') as f:
        for a_line in f:
            print(a_line)
```

John 70

Mary 50

Joe 67

Frank 90

Gru 99

Kiki 63

Wait a second...
Something's not right here.

The file type is also
“iterable”!!!



>>>

If you do it in Python Shell (bad)

```
>>> with open('student_marks.txt') as f:
```

```
    for a_line in f:  
        a_line
```

The file as an
“iterable”

```
'John 70\n'  
'Mary 50\n'  
'Joe 67\n'  
'Frank 90\n'  
'Gru 99\n'  
'Kiki 63'
```

console echo

Annoying newline
character '\n'

- How should we deal with these “\n”?

A More Complicated Example



Data.gov.sg

Search Singapore's Public Data

e.g. "rainfall", "gross domestic product", "transport"



Live-Births

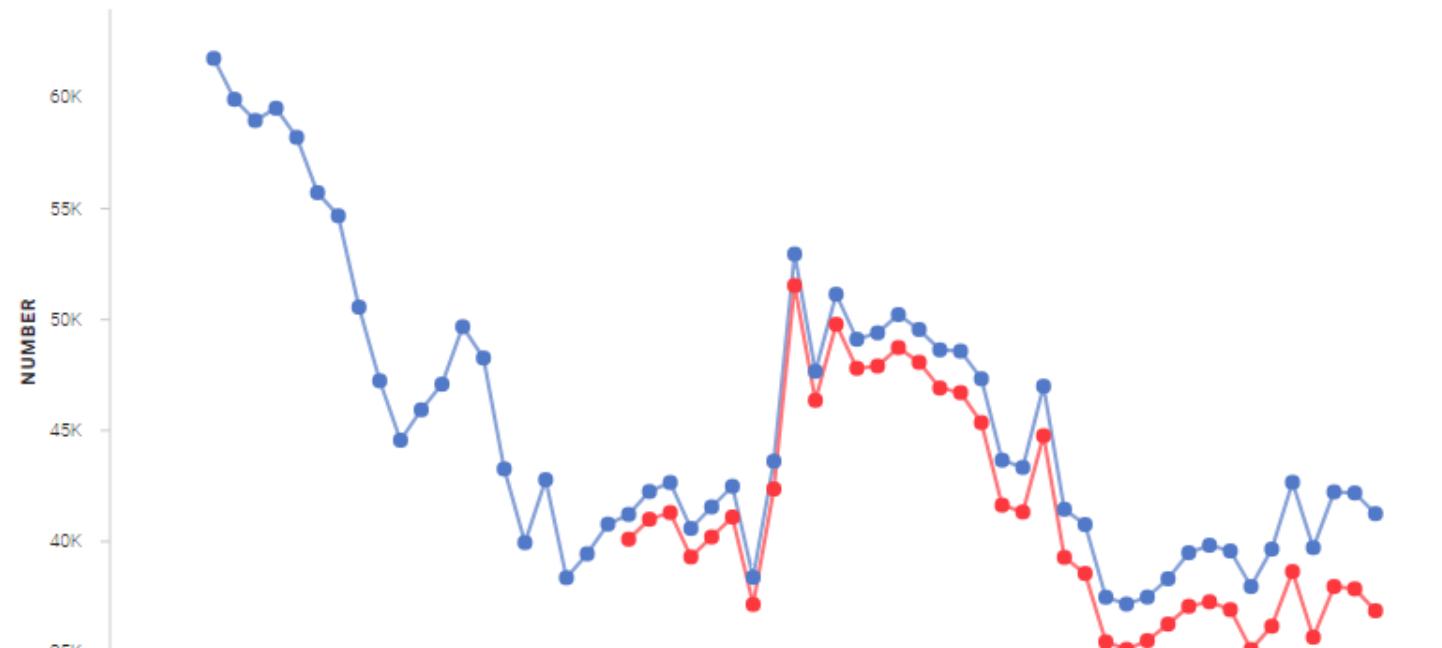
● Total Live-Births ● Resident Live-Births

Crude Birth Rate

Total Fertility Rate and
Reproduction Rate

Age-Specific Fertility Rate

Total Fertility Rate by
Ethnic Group



Open in

- Excel
- Notepad

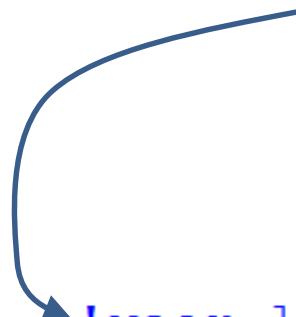
	A	B	C	D	E
1	year	level_1	value		
2	1960	Crude Birth Rate	37.5		
3	1961	Crude Birth Rate	35.2		
4	1962	Crude Birth Rate	33.7		
5	1963	Crude Birth Rate	33.2		
6	1964	Crude Birth Rate	31.6		
7	1965	Crude Birth Rate	29.5		
8	1966	Crude Birth Rate	28.3		
9	1967	Crude Birth Rate	25.6		
10	1968	Crude Birth Rate	23.5		
11	1969	Crude Birth Rate	21.8		
12	1970	Crude Birth Rate	22.1		
13	1971	Crude Birth Rate	22.3		
14	1972	Crude Birth Rate	23.1		
15	1973	Crude Birth Rate	22		
16	1974	Crude Birth Rate	19.4		
17	1975	Crude Birth Rate	17.7		
18	1976	Crude Birth Rate	18.7		
19	1977	Crude Birth Rate	16.5		
20	1978	Crude Birth Rate	16.8		
21	1979	Crude Birth Rate	17.1		
22	1980	Crude Birth Rate	17.6		
23	1981	Crude Birth Rate	17.6		
24	1982	Crude Birth Rate	17.5		
25	1983	Crude Birth Rate	16.2		

```
year,level_1,value
1960,Crude Birth Rate,37.5
1961,Crude Birth Rate,35.2
1962,Crude Birth Rate,33.7
1963,Crude Birth Rate,33.2
1964,Crude Birth Rate,31.6
1965,Crude Birth Rate,29.5
1966,Crude Birth Rate,28.3
1967,Crude Birth Rate,25.6
1968,Crude Birth Rate,23.5
1969,Crude Birth Rate,21.8
1970,Crude Birth Rate,22.1
1971,Crude Birth Rate,22.3
1972,Crude Birth Rate,23.1
1973,Crude Birth Rate,22
1974,Crude Birth Rate,19.4
1975,Crude Birth Rate,17.7
1976,Crude Birth Rate,18.7
1977,Crude Birth Rate,16.5
1978,Crude Birth Rate,16.8
1979,Crude Birth Rate,17.1
1980,Crude Birth Rate,17.6
1981,Crude Birth Rate,17.6
1982,Crude Birth Rate,17.5
1983,Crude Birth Rate,16.3
1984,Crude Birth Rate,16.5
1985,Crude Birth Rate,16.6
1986,Crude Birth Rate,14.8
1987,Crude Birth Rate,16.6
1988,Crude Birth Rate,19.8
1989,Crude Birth Rate,17.5
1990,Crude Birth Rate,18.2
1991,Crude Birth Rate,17.1
1992,Crude Birth Rate,16.8
1993,Crude Birth Rate,16.8
1994,Crude Birth Rate,16.2
```

Reading Data in Python

- You can start reading a file in Python by

```
>>> with open('crude-birth-rate.csv') as f:  
    f.readline()  
    f.readline()  
    f.readline()
```



```
'year,level_1,value\n'  
'1960,Crude Birth Rate,37.5\n'  
'1961,Crude Birth Rate,35.2\n'
```

- The line is read with a '\n' (newline)

Reading Data

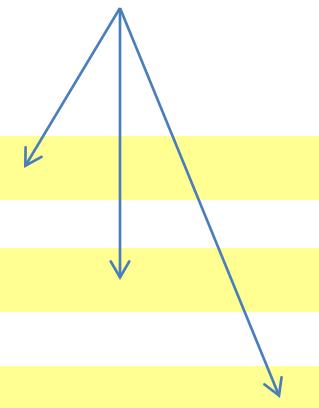
```
>>> with open('crude-birth-rate.csv') as f:  
    line1 = f.readline()  
    line2 = f.readline()  
    line3 = f.readline()  
    print(line1)  
    print(line2)  
    print(line3)
```

extra new line because
of '\n'

```
year,level_1,value
```

```
1960,Crude Birth Rate,37.5
```

```
1961,Crude Birth Rate,35.2
```



rstrip(): Strip Characters on the Right

```
>>> with open('crude-birth-rate.csv') as f:  
    line1 = f.readline().rstrip('\n')  
    line2 = f.readline().rstrip('\n')  
    line3 = f.readline().rstrip('\n')  
    print(line1)  
    print(line2)  
    print(line3)                                no more extra new line
```

```
year,level_1,value  
1960,Crude Birth Rate,37.5  
1961,Crude Birth Rate,35.2
```

```
>>>  
>>>  
>>>
```

String rstrip() and split()

```
>>> string = "555555 Hello Everybody!!! 55555"
>>> string.rstrip('5')
'555555 Hello Everybody!!! '
>>> string.lstrip('5')
' Hello Everybody!!! 55555'
>>> string.lstrip('5').rstrip('5')
' Hello Everybody!!! '

>>> string
'555555 Hello Everybody!!! 55555'
>>> string.split()
['555555', 'Hello', 'Everybody!!!!', '55555']
>>> string.split('o')
['555555 Hell', ' Everyb', 'dy!!! 55555']
```

Let's start writing the code

```
def plot_birth_rate():
    with open('crude-birth-rate.csv') as f:

        for line in f:
            print(line.rstrip('\n'))
```

- The file object 'f' is an iterable
- Every iteration you have a **hidden**
line = f.readline()

Let's start writing the code

```
def plot_birth_rate():
    with open('crude-birth-rate.csv') as f:

        for line in f:
            print(line.rstrip('\n'))
```

year	level_1	value
1960	Crude Birth Rate	37.5
1961	Crude Birth Rate	35.2
1962	Crude Birth Rate	33.7
1963	Crude Birth Rate	33.2
1964	Crude Birth Rate	31.6
1965	Crude Birth Rate	29.5
1966	Crude Birth Rate	28.3
1967	Crude Birth Rate	25.6
1968	Crude Birth Rate	23.5
1969	Crude Birth Rate	21.8
1970	Crude Birth Rate	22.1
1971	Crude Birth Rate	22.3

A string

Let's Split!

```
def plot_birth_rate():
    with open('crude-birth-rate.csv') as f:

        for line in f:
            print(line.rstrip('\n').split())
```

['year', 'level_1', 'value']
['1960, Crude', 'Birth', 'Rate, 37.5']
['1961, Crude', 'Birth', 'Rate, 35.2']
['1962, Crude', 'Birth', 'Rate, 33.7']
['1963, Crude', 'Birth', 'Rate, 33.2']
['1964, Crude', 'Birth', 'Rate, 31.6']
['1965, Crude', 'Birth', 'Rate, 29.5']
['1966, Crude', 'Birth', 'Rate, 28.3']

Split by space!!!



???

Let's Split Commas!

```
def plot_birth_rate():
    with open('crude-birth-rate.csv') as f:

        for line in f:
            print(line.rstrip('\n').split(','))
```

```
['year', 'level_1', 'value']
['1960', 'Crude Birth Rate', '37.5']
['1961', 'Crude Birth Rate', '35.2']
['1962', 'Crude Birth Rate', '33.7']
['1963', 'Crude Birth Rate', '33.2']
['1964', 'Crude Birth Rate', '31.6']
['1965', 'Crude Birth Rate', '29.5']
['1966', 'Crude Birth Rate', '28.3']
['1967', 'Crude Birth Rate', '25.6']
['1968', 'Crude Birth Rate', '23.5']
['1969', 'Crude Birth Rate', '21.8']
['1970', 'Crude Birth Rate', '22.1']
```



Let's manage our data

- We want to plot the year against the birthday
 - The value of the birth rate is x per thousand
 - So the actual no. of birth is x times 1000

```
['year', 'level_1', 'value']  
['1960', 'Crude Birth Rate', 37.5]  
['1961', 'Crude Birth Rate', 35.2]  
['1962', 'Crude Birth Rate', 33.7]  
['1963', 'Crude Birth Rate', 33.2]  
['1964', 'Crude Birth Rate', 31.6]  
['1965', 'Crude Birth Rate', 29.5]  
['1966', 'Crude Birth Rate', 28.3]  
['1967', 'Crude Birth Rate', 25.6]  
['1968', 'Crude Birth Rate', 23.5]  
['1969', 'Crude Birth Rate', 21.8]  
['1970', 'Crude Birth Rate', 22.1]
```

```
[37.5]  
[35.2]  
[33.7]  
[33.2]  
[31.6]  
[29.5]  
[28.3]  
[25.6]  
[23.5]  
[21.8]  
[22.1]
```

Convert
into integers

```
import matplotlib.pyplot as plt

def plot_birth_rate():
    with open('crude-birth-rate.csv') as f:
        f.readline() ← Discard the first line

    for line in f:
        list_form = line.rstrip('\n').split(',')
```

["line"] →

year	level 1	value
1960	Crude Birth Rate	37.5
1961	Crude Birth Rate	35.2
1962	Crude Birth Rate	33.7
1963	Crude Birth Rate	33.2

```
import matplotlib.pyplot as plt

def plot_birth_rate():
    with open('crude-birth-rate.csv') as f:
        f.readline() ← Discard the first line
        year = []
        num_birth = []
        for line in f:
            list_form = line.rstrip('\n').split(',')
            year.append(int(list_form[0]))
            num_birth.append(float(list_form[2])*1000)
```

year	level 1	value
1960	Crude Birth Rate	37.5
1961	Crude Birth Rate	35.2
1962	Crude Birth Rate	33.7
1963	Crude Birth Rate	33.2

“line”

```
import matplotlib.pyplot as plt

def plot_birth_rate():
    with open('crude-birth-rate.csv') as f:
        f.readline() ← Discard the first line
        year = []
        num_birth = []
        for line in f:
            list_form = line.rstrip('\n').split(',')
            year.append(int(list_form[0]))
            num_birth.append(float(list_form[2])*1000)

    plt.plot(year, num_birth, label="Birth Rate")
    plt.legend(loc="upper right")
    plt.title('Number of births.')
    plt.show()
```

```
plot_birth_rate()
```

year	level 1	value
1960	Crude Birth Rate	37.5
1961	Crude Birth Rate	35.2
1962	Crude Birth Rate	33.7
1963	Crude Birth Rate	33.2

"line"

Now You Know Why “Baby Bonus”



Reading CSV Files

Read in a
CSV file
into a list

Create a CSV File
Reader

```
>>> from pprint import pprint
>>> birth_file = open('crude-birth-rate.csv')
>>> birth_file_reader = csv.reader(birth_file)
>>> birth_data = list(birth_file_reader)
```

```
>>> pprint(birth_data)
[['year', 'level_1', 'value'],
 ['1960', 'Crude Birth Rate', '37.5'],
 ['1961', 'Crude Birth Rate', '35.2'],
 ['1962', 'Crude Birth Rate', '33.7'],
 ['1963', 'Crude Birth Rate', '33.2'],
 ['1964', 'Crude Birth Rate', '31.6'],
 ['1965', 'Crude Birth Rate', '29.5'],
 ['1966', 'Crude Birth Rate', '28.3'],
 ['1967', 'Crude Birth Rate', '25.6'],
 ['1968', 'Crude Birth Rate', '23.5'],
 ['1969', 'Crude Birth Rate', '21.8']]
```

Remember these
four lines of code

No need for all
those string
strip(), split() etc.

Today

- You have learned how to read and write a file
 - Or more precisely, reading or writing a general file
 - In fact, we got an easier way to read a CSV file
 - Wait until we learn multi-dimensional arrays
- You can say that you “finished” the (most of the) “core” Python Language
- The rest is extra packages, features

Sequences and Higher Order Functions

Scaling a Sequence

- Given a sequence of numbers, how to scale every element?
- Let's say, scale by 2

[5,1,4,9,11,22,12,55]



[10,2,8,18,22,44,24,110]

Scaling a Sequence

- Given a sequence of numbers, how to scale every element?

```
def seqScaleI(seq,n):  
    output = []  
    for i in seq:  
        output.append(i*n)  
    return output
```

```
def seqScaleR(seq,n):  
    if not seq:  
        return seq  
    return [seq[0]*n]+seqScaleR(seq[1:],n)
```

Squaring a Sequence

- Given a sequence of numbers, how to square every element?

[5,1,4,9,11,22,12,55]



[25, 1, 16, 81, 121, 484, 144, 3025]

```
def seqSquareI(seq):
    output = []
    for i in seq:
        output.append(i*i)
    return output
```

Squaring/Scaling a Sequence

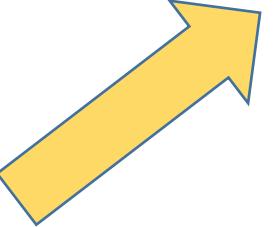
- Other than the function name (that can change to anything), what is the difference?

```
def seqScaleI(seq,n):  
    output = []  
    for i in seq:  
        output.append(i*n)  
    return output
```

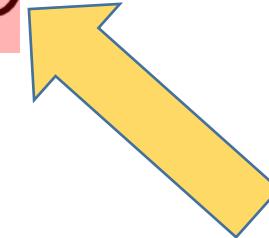
```
def seqSquareI(seq):  
    output = []  
    for i in seq:  
        output.append(i*i)  
    return output
```

- What should we do with other operations to a sequence?
 - E.g. cube, abs, etc.?

```
def map(f,seq):  
    output = []  
    for i in seq:  
        output.append(f(i))  
    return output
```



```
def seqScaleI(seq,n):  
    output = []  
    for i in seq:  
        output.append(i*n)  
    return output
```



```
def seqSquareI(seq):  
    output = []  
    for i in seq:  
        output.append(i*i)  
    return output
```

Difference Operations on a Sequence

```
>>> lst = [5,1,4,9,11,22,12,55]
>>> map(square,lst)
[25, 1, 16, 81, 121, 484, 144, 3025]
>>> map(scale2,lst)
[10, 2, 8, 18, 22, 44, 24, 110]
>>> map(lambda x:x*x,lst)
[25, 1, 16, 81, 121, 484, 144, 3025]
>>> map(lambda x:2*x,lst)
[10, 2, 8, 18, 22, 44, 24, 110]
>>> map(lambda x:-x,lst)
[-5, -1, -4, -9, -11, -22, -12, -55]
```

Our map()

- However, our map() can only process list
 - Cannot work on other sequences like tuples, strings, etc.
- However, Python does have its original version of map()
 - But it will return a type “map” object
 - You can convert that object into other sequences like list or tuples

```
>>> tup = (1,-2,3)
>>> map1 = map(abs,tup)
>>> map1
<map object at 0x112e61438>
>>> type(map1)
<class 'map'>
>>> map1Tuple = tuple(map1)
>>> map1Tuple
(1, 2, 3)
>>> map1List = list(map1)
>>> map1List
[]
```

Python's map()

```
>>> tup = (1,-2,3)
>>> map1 = map(abs,tup)
>>> map1
<map object at 0x112e61358>
>>> type(map1)
<class 'map'>
>>> map1List = list(map1)
>>> map1List
[1, 2, 3]
>>> map1Tuple = tuple(map1)
>>> map1Tuple
()
```

```
>>> tup = (1,-2,3)
>>> map1 = map(abs,tup)
>>> map1
<map object at 0x112e61438>
>>> type(map1)
<class 'map'>
>>> map1Tuple = tuple(map1)
>>> map1Tuple
(1, 2, 3)
>>> map1List = list(map1)
>>> map1List
[]
```

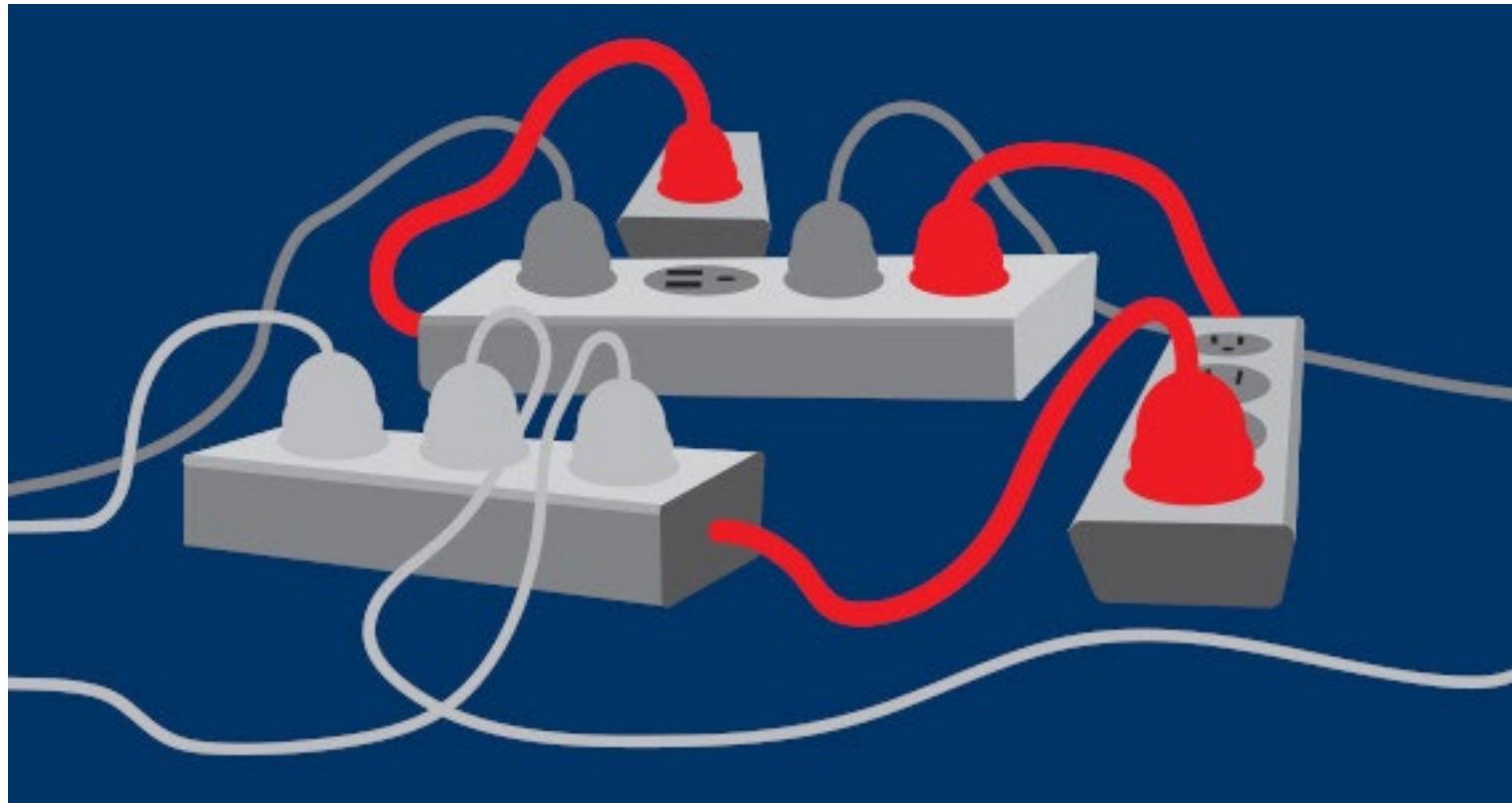
Python's map()

- The map object is actually an “iterable”
 - After you “took out” items from the map object, the items will be “gone”
- Conclusion
 - Conversion from a map object to a tuple or list only once

```
>>> tup = (1,-2,3)
>>> map1 = map(abs,tup)
>>> map1
<map object at 0x112e61438>
>>> type(map1)
<class 'map'>
>>> map1Tuple = tuple(map1)
>>> map1Tuple
(1, 2, 3)
>>> map1List = list(map1)
>>> map1List
[]
```

Kattis Demo

- <https://open.kattis.com/problems/electricaloutlets>



Python's Filter

Python's Filter

- Python's map()
 - Apply a function f to every item x in the sequence
- Python's filter()
 - Apply a **predicate** function f to every item x in the sequence
 - A predicate is a function that return True or False
 - Return an iterable that
 - Keep the item if $f(x)$ returns True
 - Remove the item otherwise

Python's filter()

```
>>> l = [1,2,3,'a',(1,2),('b',3)]
>>> filter(lambda x:type(x)==int,l)
<filter object at 0x112e618d0>
>>> list(filter(lambda x:type(x)==int,l))
[1, 2, 3]
>>> l = [1,2,'a',(1,2),6,('b',3),999]
>>> list(filter(lambda x:type(x)==int,l))
[1, 2, 6, 999]
>>> list(filter(lambda x:type(x)==str,l))
['a']
>>> l2 = [1,4,5,-4,9,-99,0,32,-9]
>>> list(filter(lambda x: x < 0 , l2))
[-4, -99, -9]
```

Counting a Sequence
Shallowly or Deeply

How to Count the Number of Element in a Sequence?

```
>>> lst = [5,1,4,9,11,22,12,55]  
>>> seqCountI(lst)  
8
```

- Of course we can use `len()`
- But what if we want to implement it ourselves?

```
def seqCountR(seq):  
    if not seq:  
        return 0  
    return 1 + seqCountR(seq[1:])
```

However, it's Shallow

```
>>> lst2 = [1,2,3,[4,5,6,7]]  
>>> seqCountR(lst2)  
4
```

- How to count "deeply"?

```
>>> deepcount(lst2)  
7
```

- And what about a list like this

```
>>> lst3 = [1, 4, 9, [1, 4], [4, 9, 16, [1, 4, 9]], [9, 16, 25]]  
>>> deepcount(lst3)  
14
```

Counting Logic? Shallow Count

- Total count = count of the **first** item + count of the rest
 - But the count of the first item is always 1

```
def seqCountR(seq):  
    if not seq:  
        return 0  
    return 1 + seqCountR(seq[1:])
```

- Can we do the same thing for deep count?
- What is the difference between deep and shallow count?
 - In deep count, the “length” of the first item may not be 1

Counting Logic? Deep Count

- Total count = count of the **first** item + count of the rest
 - But the count of the first item is only 1 if it's not a sequence
- **[1,2,3,4,[2,3,4],[1]]**
 - The first item has a count 1
- **[[1,2],3,4,5]**
 - The first item does not have a count 1
- Two questions:
 - How to tell the first item is a sequence or not?
 - What to do if the first item is a sequence?

First Question

- How to tell the first item is a list or not a list
 - Assuming we only have list
 - Not difficult to extend to tuples
- Check

`type(seq[0]) == list`

- E.g.

```
>>> l1 = [1,2,3,4,[2,3,4],[1]]
```

```
>>> type(l1[0]) == list
```

False

```
>>> l2 = [[1,2],3,4,5]
```

```
>>> type(l2[0]) == list
```

True

Second Question

- If the first item is NOT a list, e.g. `[1,2,3,4,[2,3,4],[1]]`
 - count of the first item is 1
- If the first item IS a list, e.g. `[[1,2],3,4,5]`
 - recursively compute `deepcount()` of the first item!
- And this can handle if the first item is a list of a list of a list of
- e.g.
 - `[[[[1,2],2],4],2],3,4,5]`

Second Question

- If the first item is NOT a list, e.g. `[1,2,3,4,[2,3,4],[1]]`
 - count of the first item is 1
- If the first item IS a list, e.g. `[[1,2],3,4,5]`
 - recursively compute `deepcount()` of the first item!

```
def deepcount(seq):
    if seq == []:
        return 0
    elif type(seq) != list:
        return 1
    else:
        return deepcount(seq[0])+ deepcount(seq[1:])
```

Simple Test

```
deepcount([1])
```



1 deepcount(1) + deepcount([])

```
def deepcount(seq):
    if seq == []:
        return 0
    elif type(seq) != list:
        return 1
    else:
        return deepcount(seq[0])+ deepcount(seq[1:])
```

Whenever we
reached this line
Count + 1

Simple Test

```
deepcount([1,2,3])
```



```
deepcount(1) + deepcount([2,3])
```

1



```
1 deepcount(2) + deepcount([3])
```

1



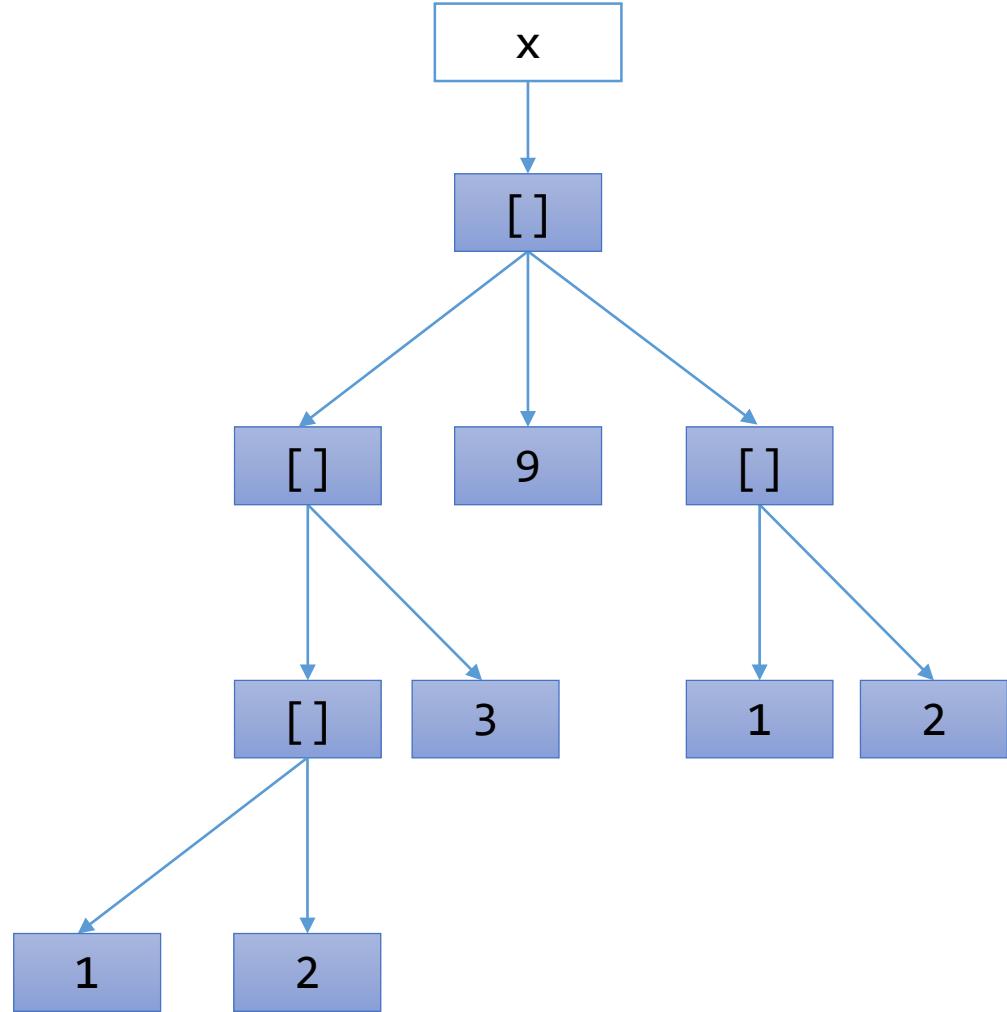
```
1 deepcount(3) + deepcount([])
```

1

Tracing the Code

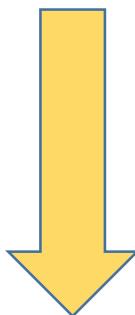
- $x = [[[1, 2], 3], 9, [1, 2]]$

```
def deepcount(seq):
    if seq == []:
        return 0
    elif type(seq) != list:
        return 1
    else:
        return deepcount(seq[0])+ deepcount(seq[1:])
```

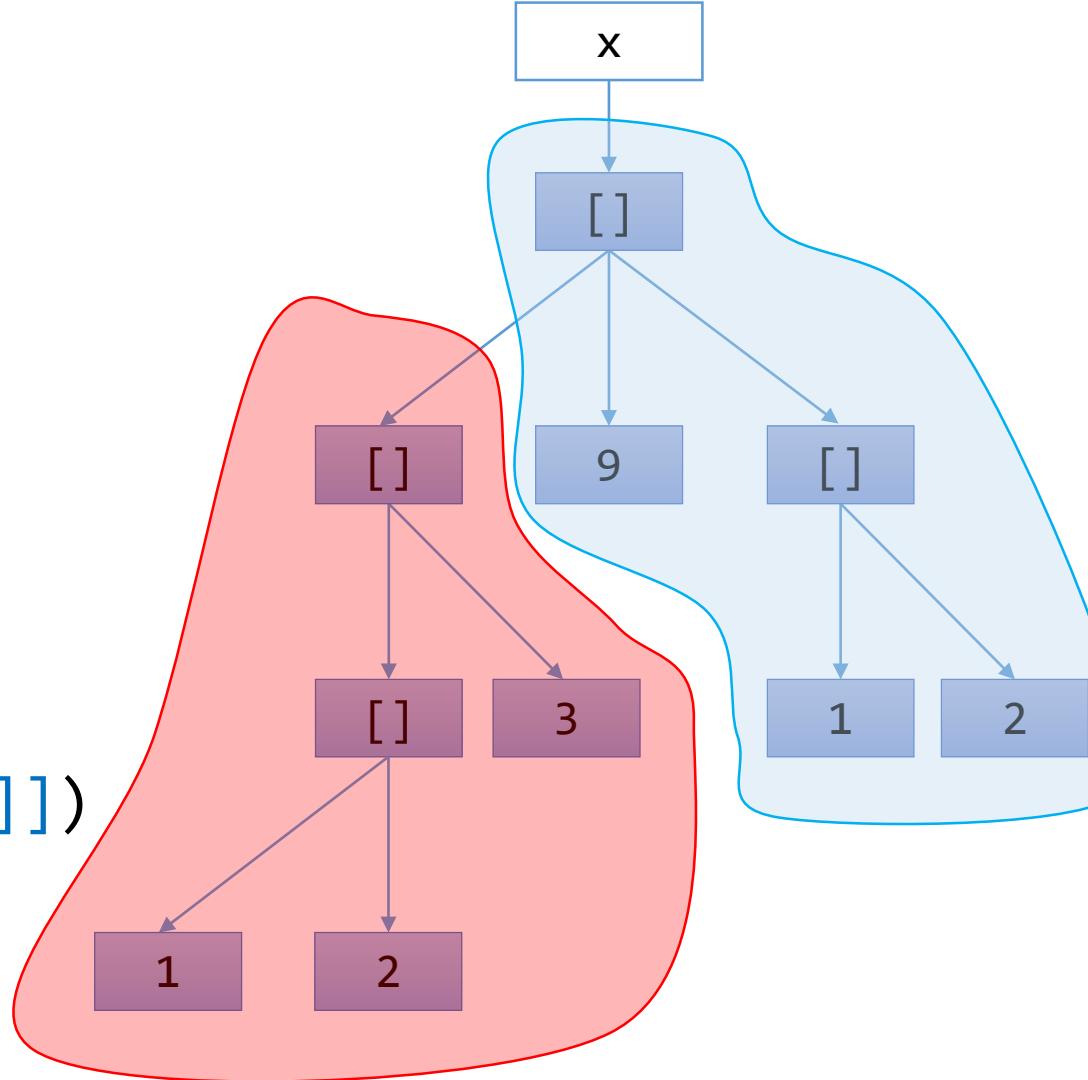


Tracing the Code

deepcount([[1,2],3],9,[1,2]])



deepcount([[1,2],3])+deepcount([9,[1,2]])



Let's Consider the Left Term

deepcount([[1,2],3])



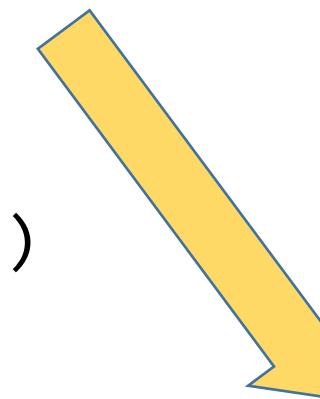
deepcount([1,2]) + deepcount([3])



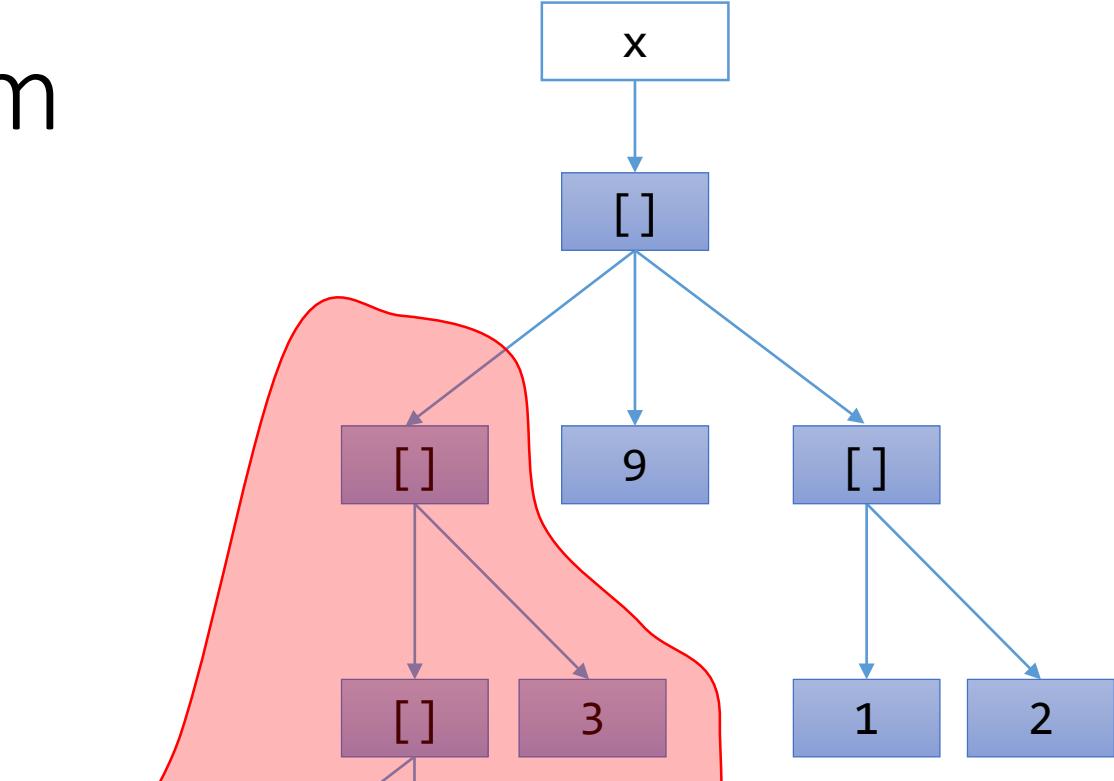
deepcount(1) + deepcount([2])



deepcount(2)+deepcount([])

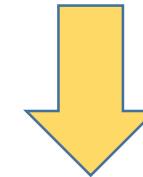


deepcount(3)+deepcount([])

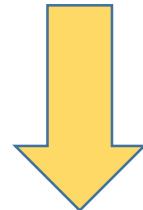


Let's Consider the Left Term

deepcount([[1,2],3])



deepcount([1,2]) + deepcount([3])

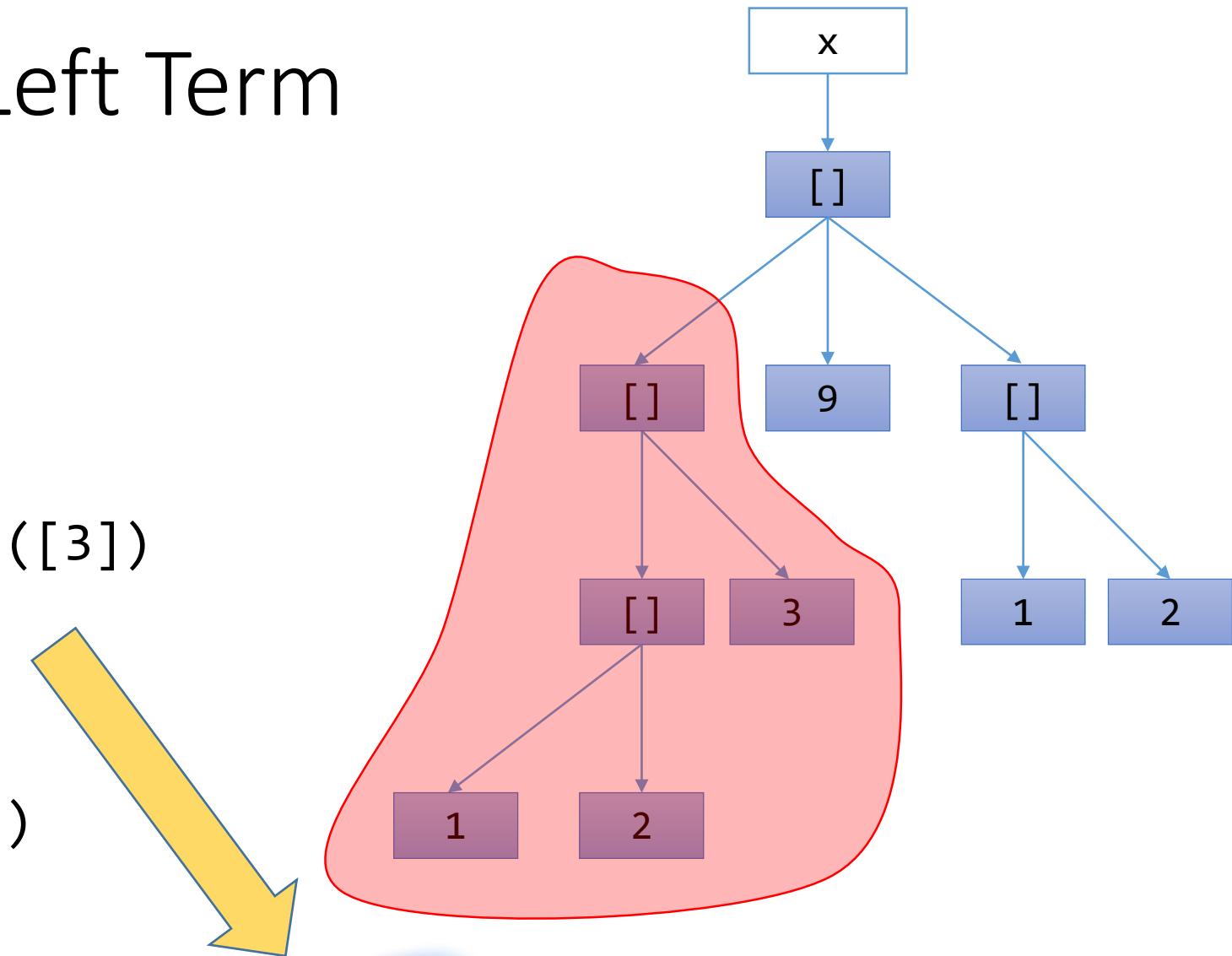


1 deepcount(1) + deepcount([2])



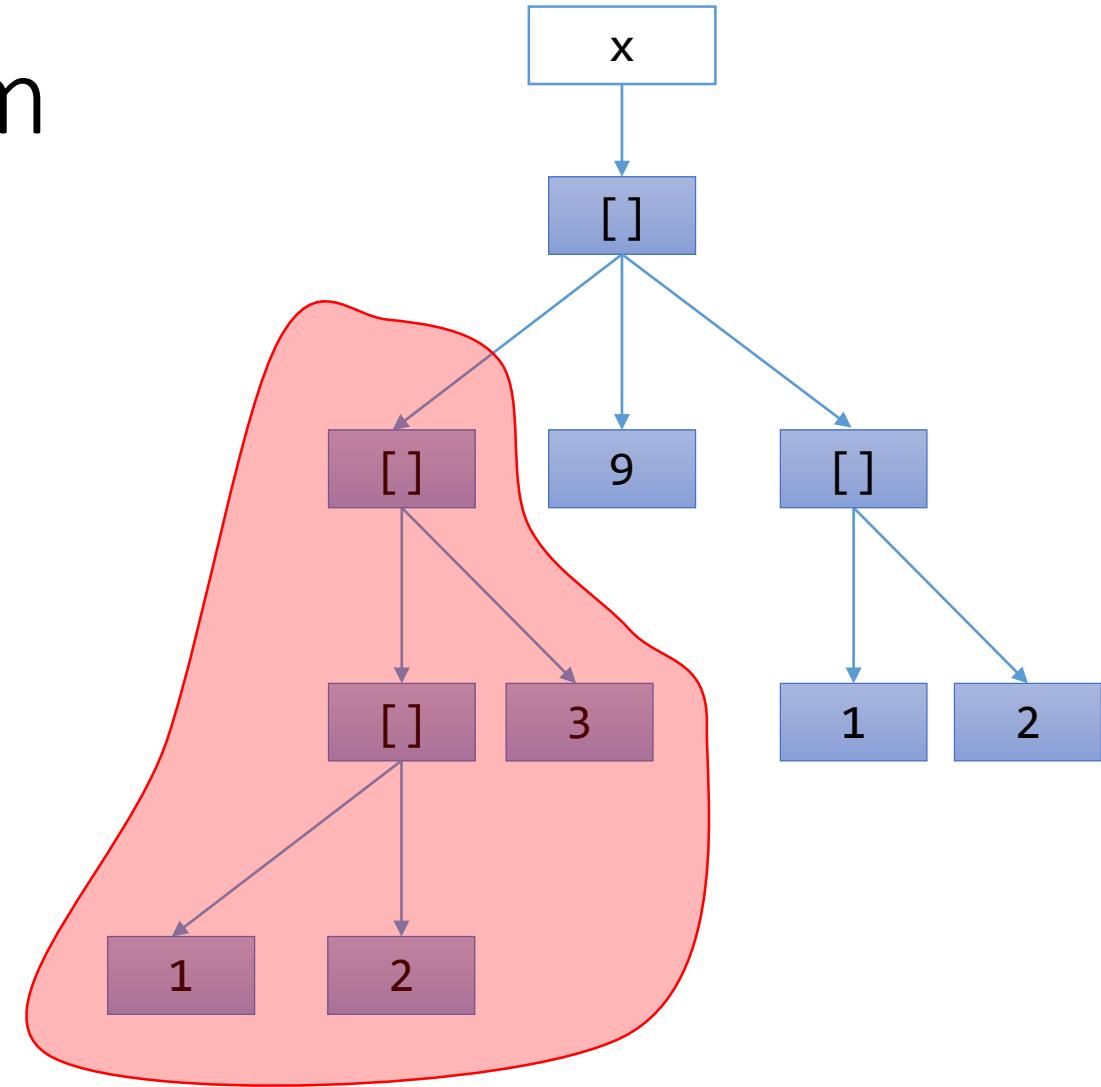
deepcount(2)+deepcount([])

deepcount(3)+deepcount([])



Let's Consider the Left Term

deepcount([[1,2],3])) ↪ 3



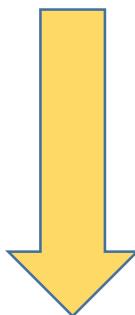
1

1

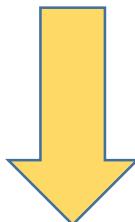
1

Tracing the Code

deepcount([[1,2],3],9,[1,2]])

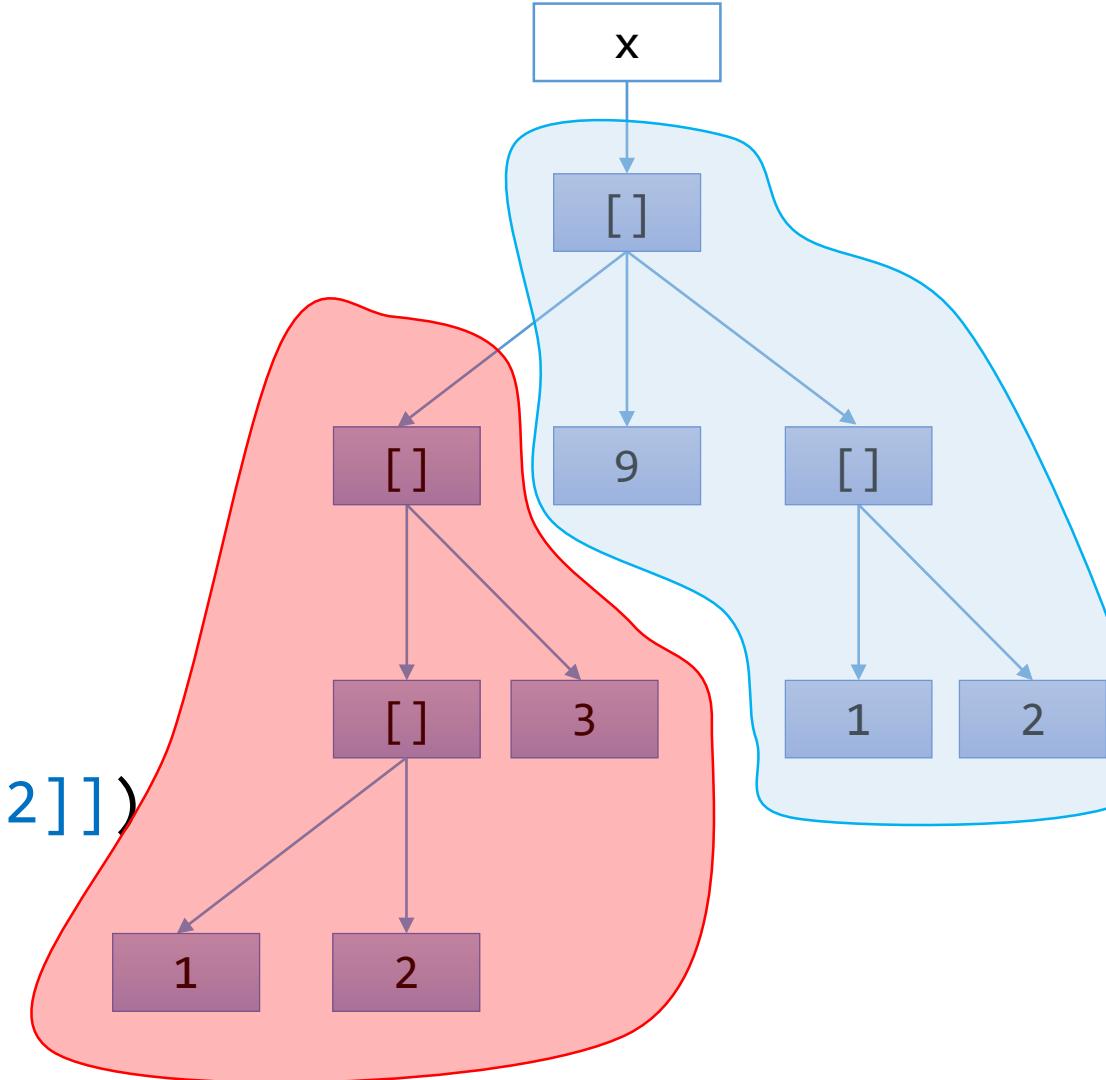


deepcount([[1,2],3])+deepcount([9,[1,2]])



3

+deepcount([9,[1,2]])

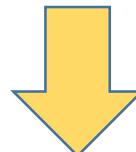


Tracing the Code

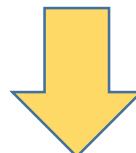
deepcount([9,[1,2]])



deepcount(**9**) + deepcount(**[[1,2]]**)



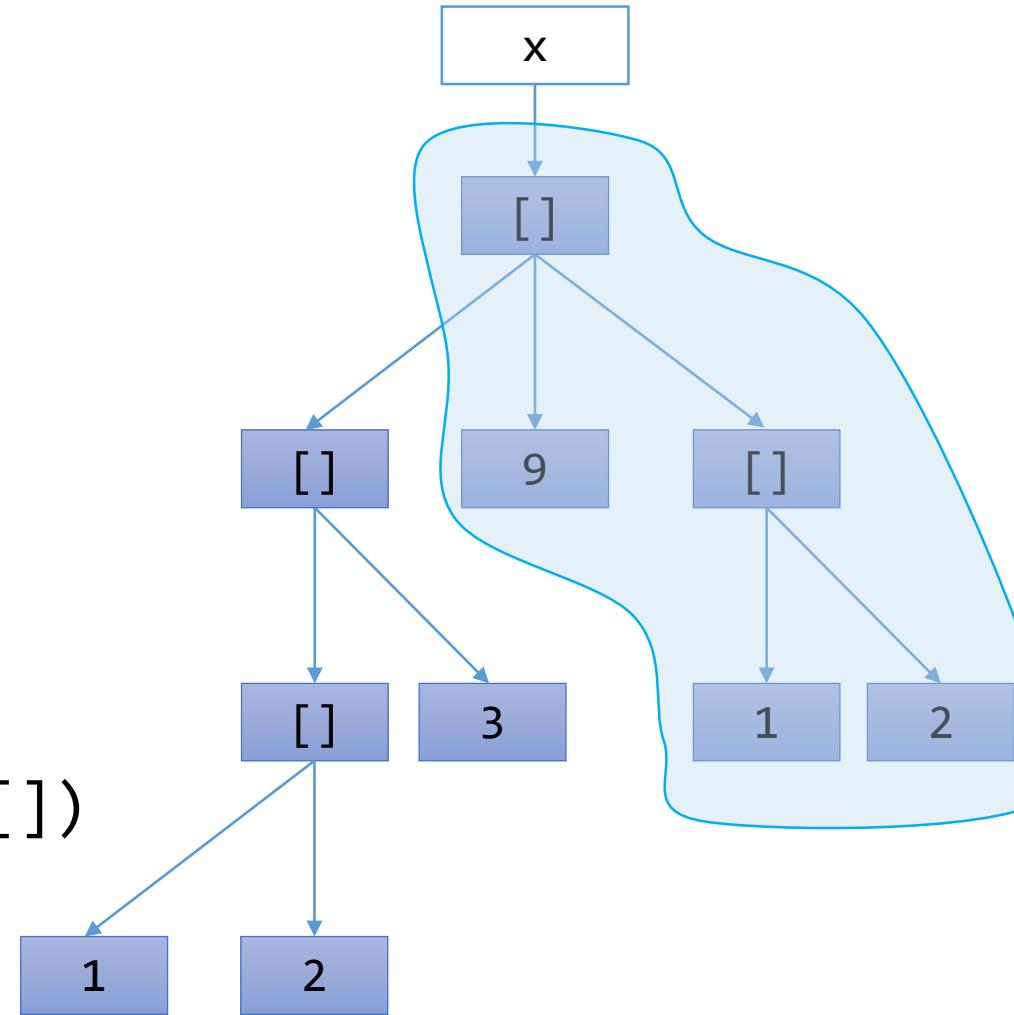
deepcount(**[1,2]**) + deepcount(**[]**)



deepcount(**1**) + deepcount(**[2]**)



deepcount(**2**) + deepcount(**[]**)



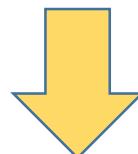
Tracing the Code

deepcount([9,[1,2]])

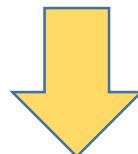


deepcount(**9**) + deepcount(**[[1,2]]**)

1



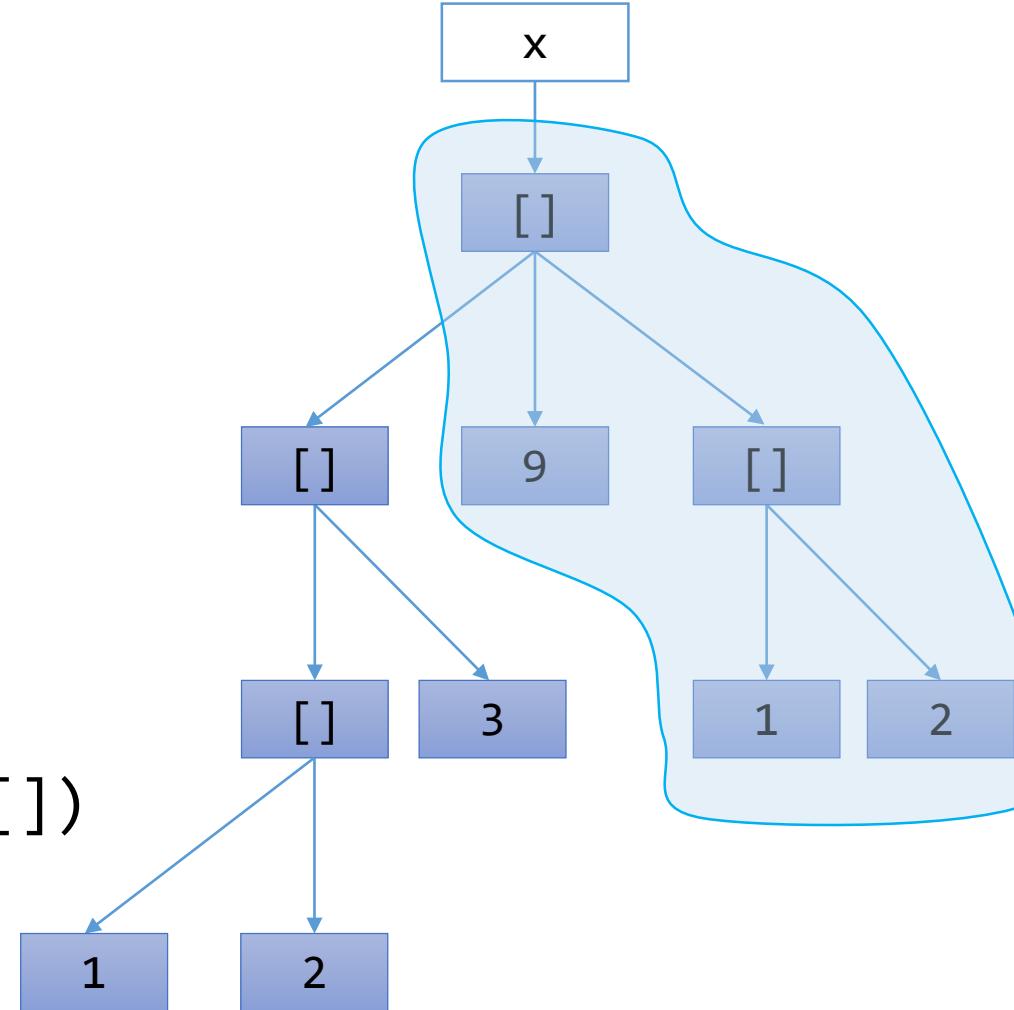
deepcount([1,2]) + deepcount([])



deepcount(1) + deepcount([2])

1

deepcount(**1**) + deepcount([])



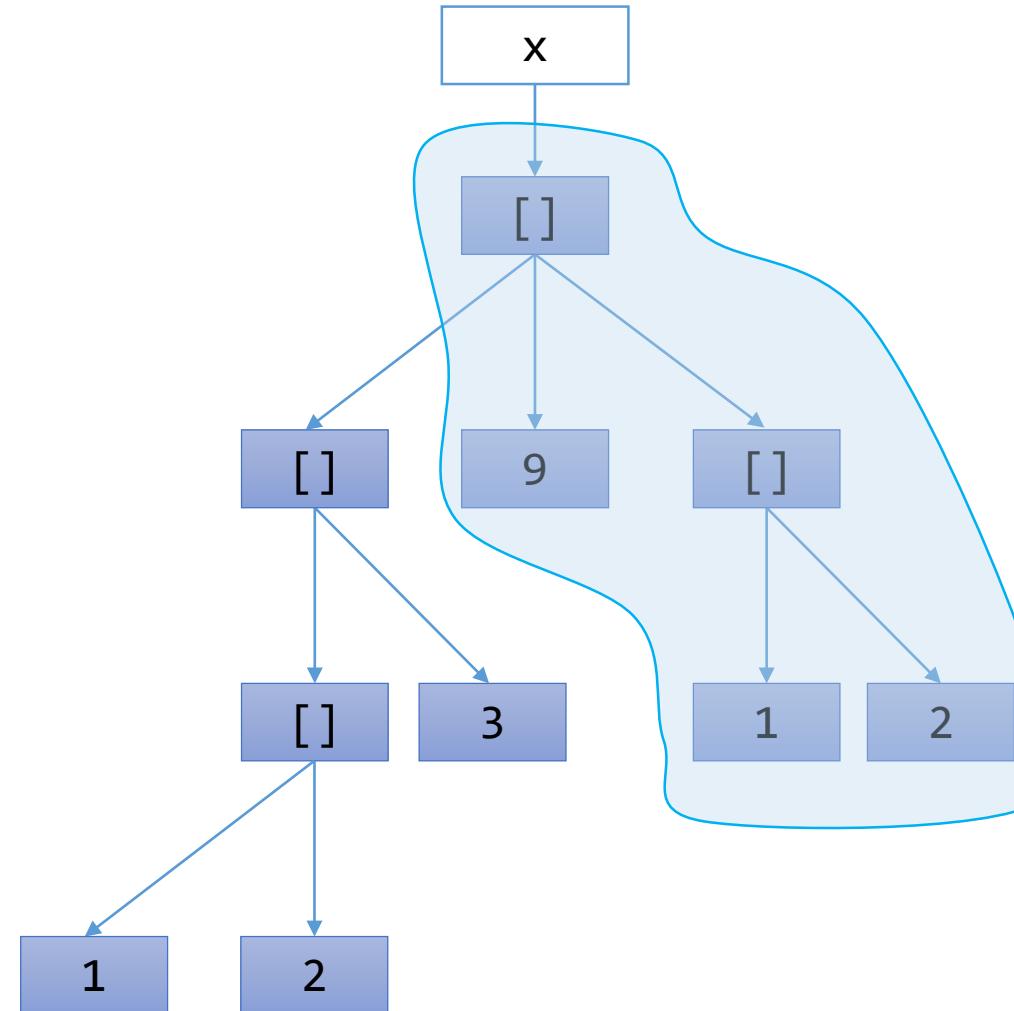
Tracing the Code

deepcount([9,[1,2]]) ↪ 3

1

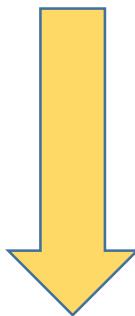
1

1

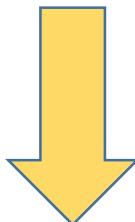


Tracing the Code

deepcount([[1,2],3],9,[1,2]])



deepcount([[1,2],3])+deepcount([9,[1,2]])

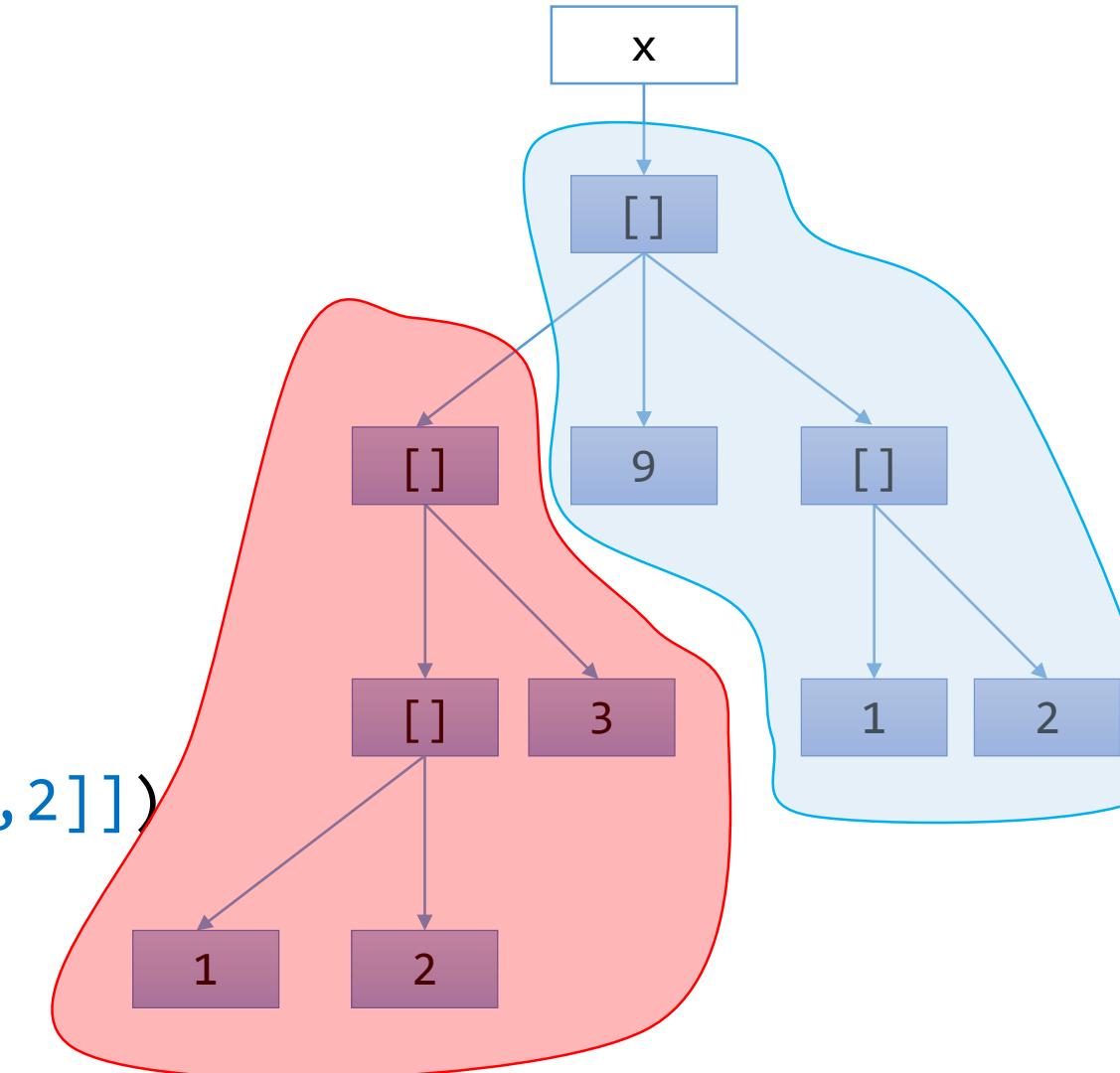


3

+

3

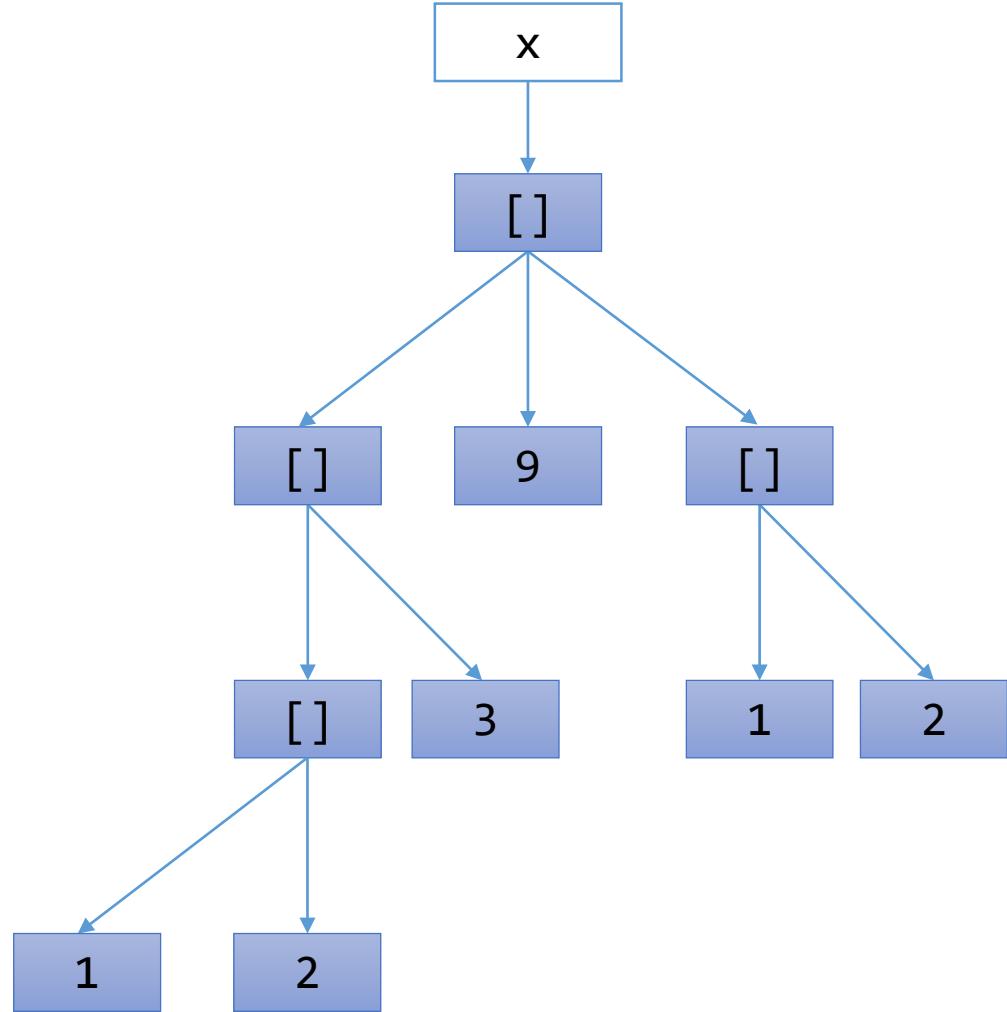
= 6



Tracing the Code

- $x = [[[1, 2], 3], 9, [1, 2]]$

```
def deepcount(seq):
    if seq == []:
        return 0
    elif type(seq) != list:
        return 1
    else:
        return deepcount(seq[0])+ deepcount(seq[1:])
```



How about

- DeepSquare?

```
>>> l = [1, 4, 9, [1, 4], [4, 9, 16, [1, 4, 9]], [9, 16, 25]]  
>>> deepSquare(l)  
[1, 16, 81, [1, 16], [16, 81, 256, [1, 16, 81]], [81, 256, 625]]
```

- Deep Increment (by 1)?

```
>>> deepInc(l)  
[2, 3, 4, [2, 3], [3, 4, 5, [2, 3, 4]], [4, 5, 6]]  
>>> deepInc(deepInc(deepInc(l)))  
[4, 5, 6, [4, 5], [5, 6, 7, [4, 5, 6]], [6, 7, 8]]
```

Get Some Insight from DeepCount?

```
def deepcount(seq):
    if seq == []:
        return 0
    elif type(seq) != list:
        return 1
    else:
        return deepcount(seq[0])+ deepcount(seq[1:])
```

DeepSquare

- Spot the difference?

```
def deepSquare(seq):
    if seq == []:
        return seq
    elif type(seq) != list:
        return seq*seq
    else:
        return [deepSquare(seq[0])] + deepSquare(seq[1:])
```

DeepSquare

- Base case is different
 - If seq is reduced to an empty list, return it as it is

```
def deepSquare(seq):  
    if seq == []:  
        return seq  
    elif type(seq) != list:  
        return seq*seq  
    else:  
        return [deepSquare(seq[0])] + deepSquare(seq[1:])
```

DeepSquare

- The leaf case
 - If the item is not a list, return its square instead of 1

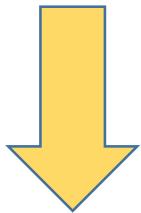
```
def deepSquare(seq):  
    if seq == []:  
        return seq  
    elif type(seq) != list:  
        return seq*seq  
    else:  
        return [deepSquare(seq[0])] + deepSquare(seq[1:])
```

DeepSquare

- Otherwise
 - Return the list of recursive call of the first item
 - Huh? Why not return the “recursive call of the first item”? Why an extra “layer” of list?
 - And concatenate with the recursive call of the “rest” of the list

```
def deepSquare(seq):
    if seq == []:
        return seq
    elif type(seq) != list:
        return seq*seq
    else:
        return [deepSquare(seq[0])] + deepSquare(seq[1:])
```

deepSquare([2])



[deepSquare(2)] + deepSquare([])



[2*2]

+



[]



[4]

```
def deepSquare(seq):
    if seq == []:
        return seq
    elif type(seq) != list:
        return seq*seq
    else:
        return [deepSquare(seq[0])] + deepSquare(seq[1:])
```

DeepInc

- How different from DeepSquare?

```
def deepInc(seq):
    if seq == () or seq == []:
        return seq
    elif type(seq) != list:
        return seq+1
    else:
        return [deepInc(seq[0])] + deepInc(seq[1:])
```



ARE YOU THINKING...

WHAT I'M THINKING???

```
def deepInc(seq):
    if seq == () or seq == []:
        return seq
    elif type(seq) != list:
        return seq+1
    else:
        return [deepInc(seq[0])] + deepInc(seq[1:])

def deepSquare(seq):
    if seq == []:
        return seq
    elif type(seq) != list:
        return seq*seq
    else:
        return [deepSquare(seq[0])] + deepSquare(seq[1:])
```

deepMap !!!!

```
def deepMap(func,seq):
    if seq == []:
        return seq
    elif type(seq) != list:
        return func(seq)
    else:
        return [deepMap(func,seq[0])] + deepMap(func,seq[1:])
```

deepMap!!!

```
>>> l = [1, 2, 3, [1, 2], [2, 3, 4, [1, 2, 3]], [3, 4, 5]]  
>>> deepMap(square,l)  
[1, 4, 9, [1, 4], [4, 9, 16, [1, 4, 9]], [9, 16, 25]]  
>>> deepMap(str,l)  
['1', '2', '3', ['1', '2'], ['2', '3', '4', ['1', '2', '3']],  
 ['3', '4', '5']]  
>>> deepMap(lambda x:x/2,l)  
[0.5, 1.0, 1.5, [0.5, 1.0], [1.0, 1.5, 2.0, [0.5, 1.0, 1.5]],  
 [1.5, 2.0, 2.5]]
```

Remember List Copy by copy()?

```
>>> l2 = l.copy()  
>>> l2  
[1, 2, 3, [1, 2], [2, 3, 4, [1, 2, 3]], [3, 4, 5]]  
>>> l[3][0] = 999  
>>> l2  
[1, 2, 3, [999, 2], [2, 3, 4, [1, 2, 3]], [3, 4, 5]]  
>>> l  
[1, 2, 3, [999, 2], [2, 3, 4, [1, 2, 3]], [3, 4, 5]]
```

- Shallow copy!!!!!! (Please refer to tutorials)

deepCopy()

```
>>> l2 = deepMap(lambda x: x.copy() if type(x)==list else x,l)
>>> l2
[1, 2, 3, [1, 2], [2, 3, 4, [1, 2, 3]], [3, 4, 5]]
```

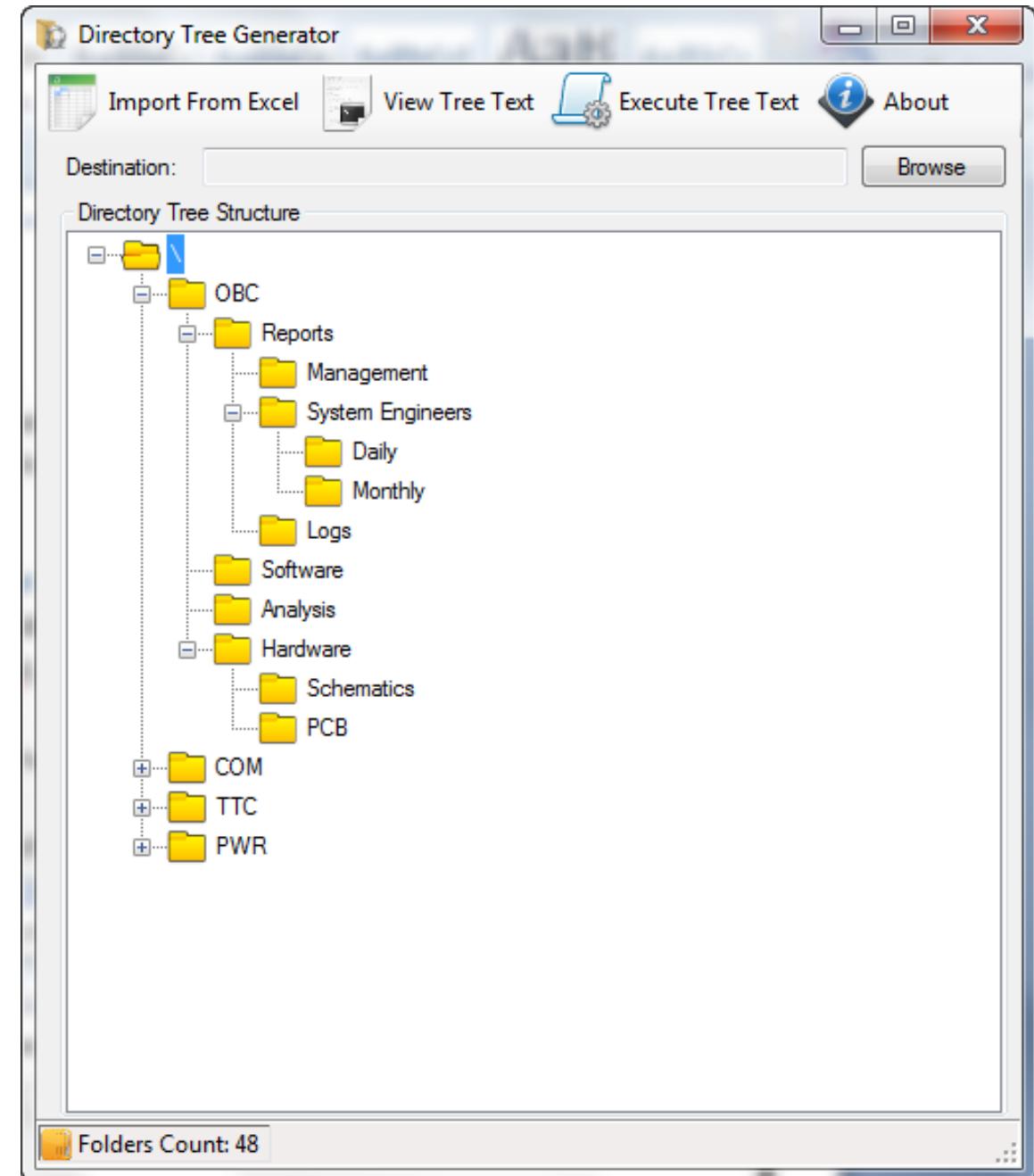
- And it works!

```
>>> l[3][0] = 999
>>> l
[1, 2, 3, [999, 2], [2, 3, 4, [1, 2, 3]], [3, 4, 5]]
>>> l2
[1, 2, 3, [1, 2], [2, 3, 4, [1, 2, 3]], [3, 4, 5]]
```

Why Do I Want to Go
“Deep”?

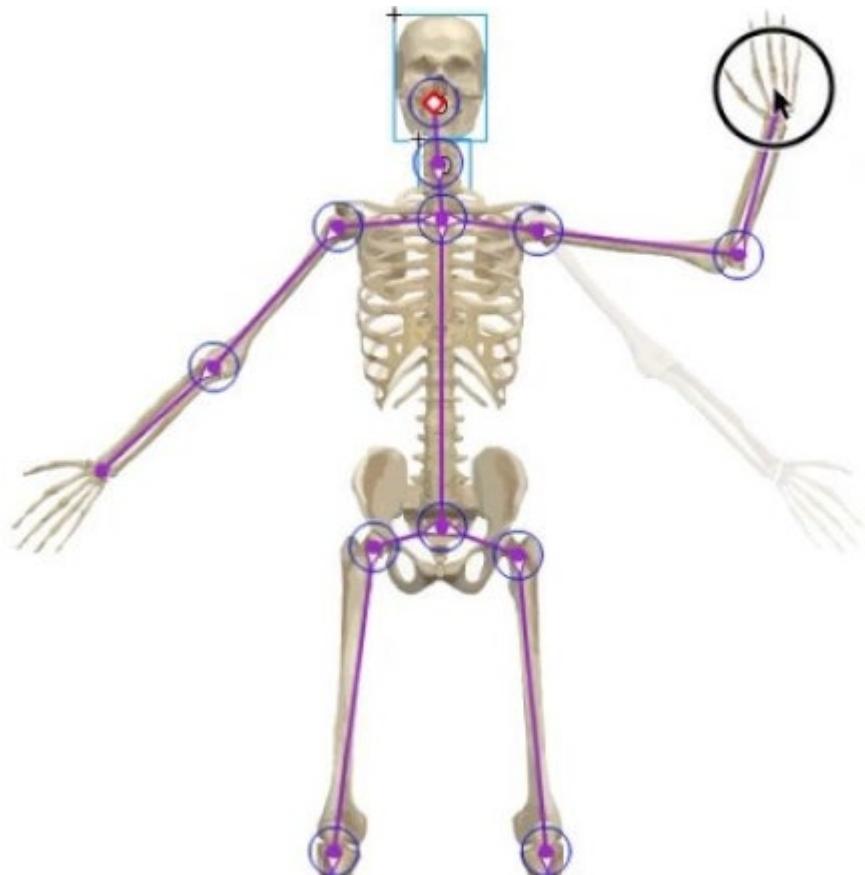
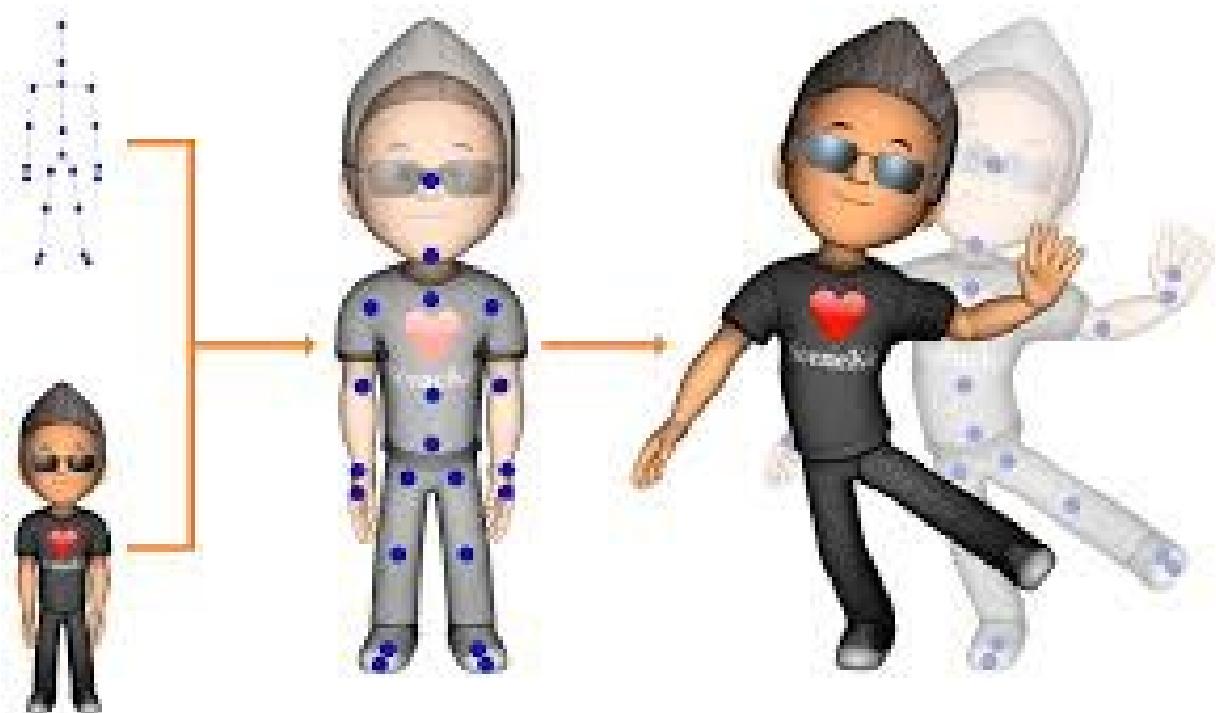
Copying a Directory

- When the directory contains a lot of files in many subdirectories



Computer Animation

- Skeleton animation



Shortest Path Tree

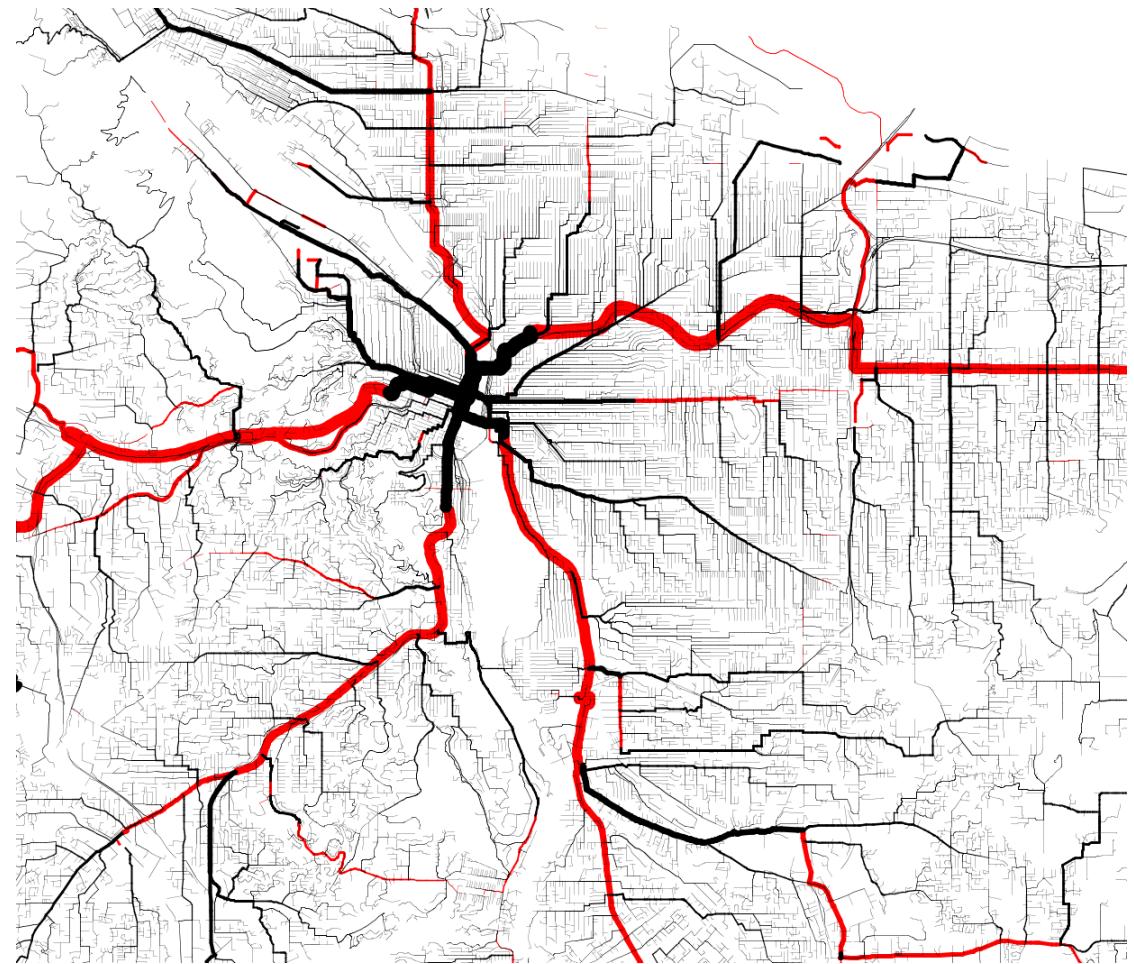
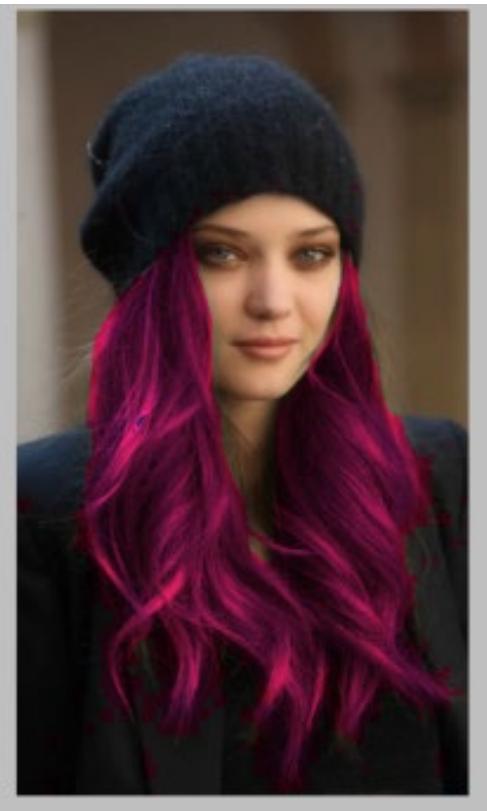


Image Processing

- An image is a list of lists of lists
 - The first level list: rows
 - The second level of lists: column
 - The third level of lists: RGB values
- Map a function to change certain values
 - E.g. change colors



How about if I just want to
be shallow?



How about if I just want to be shallow?

- Given a nested list, output a list with all the elements but without any

```
>>> l = [1, 2, 3, [1, 2], [2, 3, 4, [1, 2, 3]], [3, 4, 5]]  
>>> flatten(l)  
[1, 2, 3, 1, 2, 2, 3, 4, 1, 2, 3, 3, 4, 5]
```

Flatten()

```
def flatten(seq):
    if seq == []:
        return seq
    elif type(seq) != list:
        return [seq]
    else:
        return flatten(seq[0])+ flatten(seq[1:])
```

```
flatten([[1]])
```

```
flatten([[1]]) + flatten([])
```

```
flatten([1]) + flatten([]) + flatten([])
```

```
flatten(1) + flatten([]) + flatten([]) + flatten([])
```

```
[1]           + flatten([]) + flatten([]) + flatten([])
```

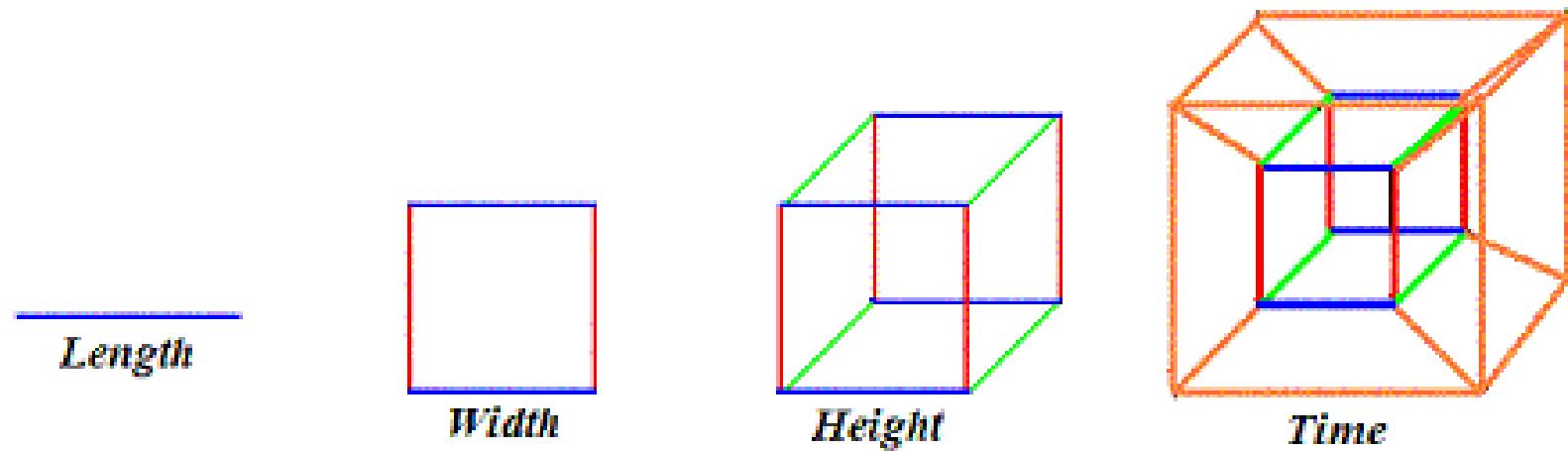
Conclusions

- map() is a powerful tool in Python
 - Allows you to perform a lot of operations with less redundant code
- Deep operations are useful to solve problems with non-linear data
 - E.g. Trees, n-dim arrays, graphs
- Using recursive functions wisely is the key for algorithms
 - Higher level of coding

Dimensions

- Are we living in a three dimensional space?

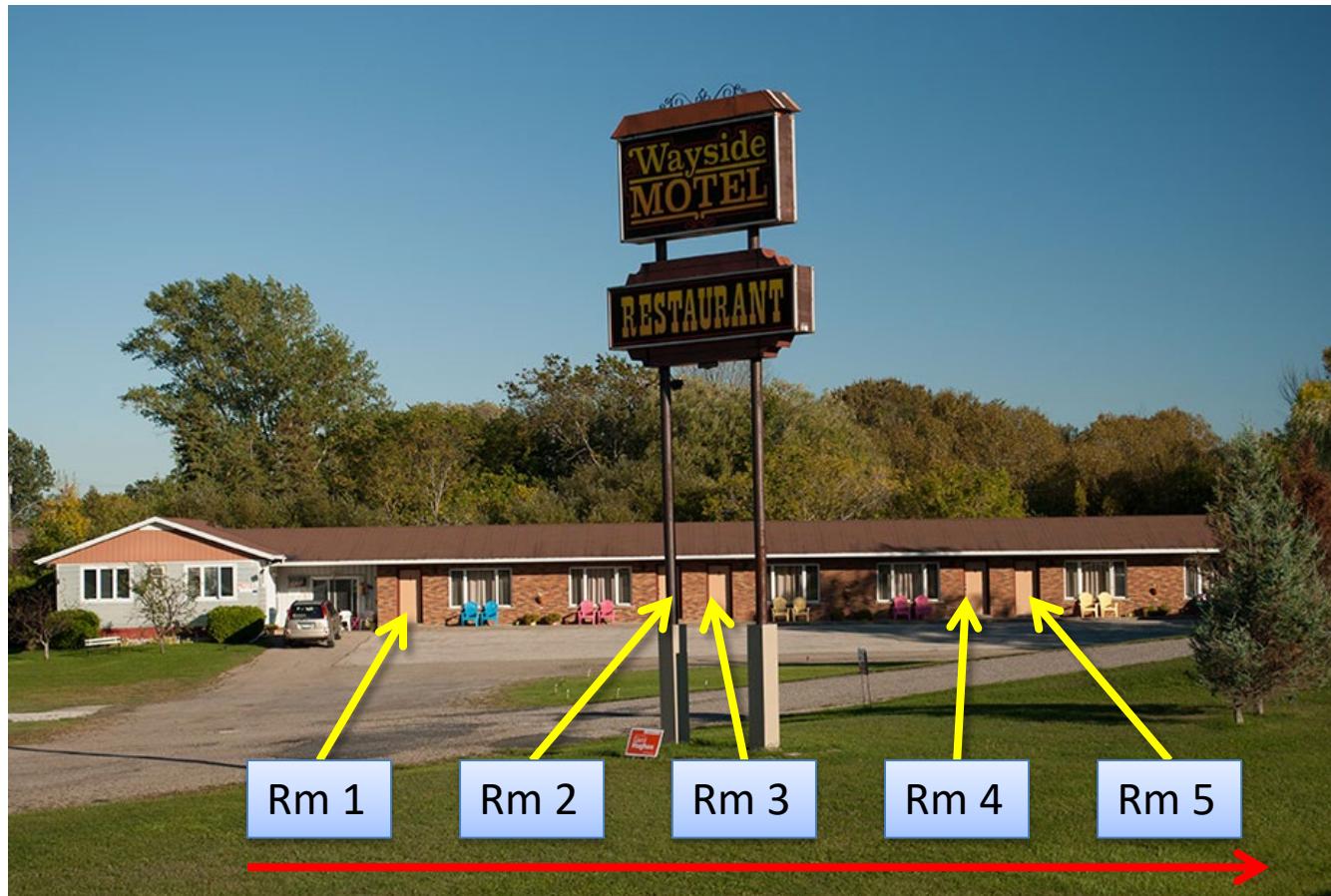
The Four Dimensions



[The xkcd guide to the fourth dimension](#)

Two-dimensional Array

Motel in US (1D)

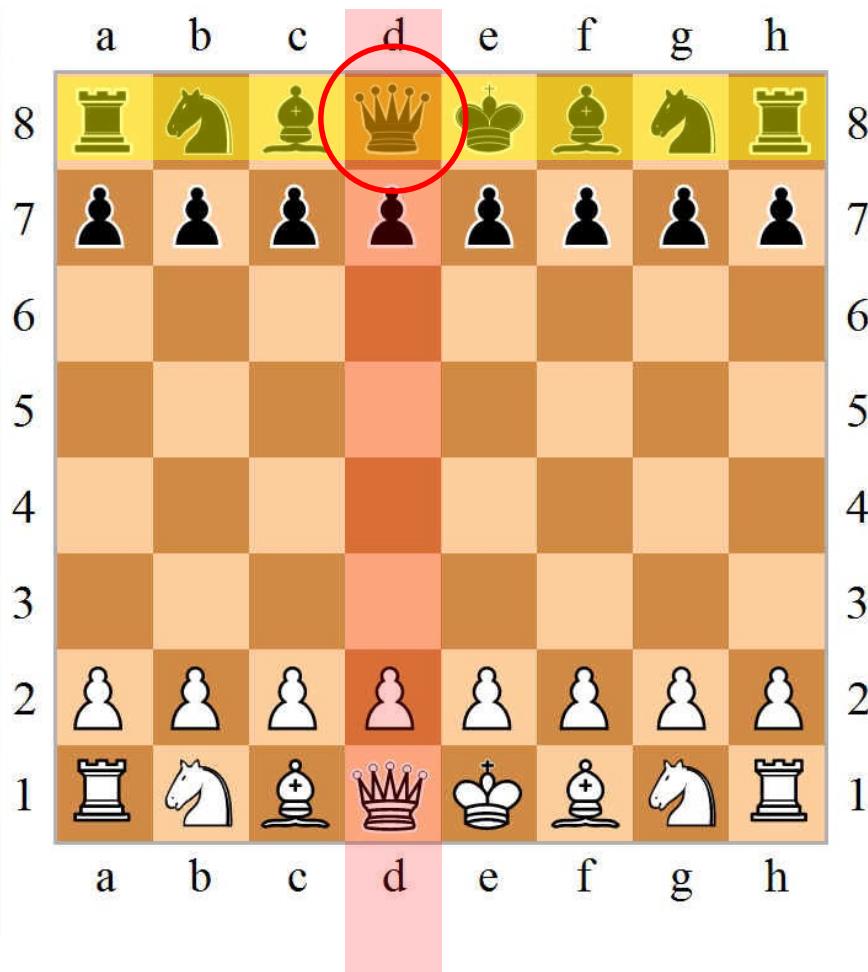


Hotel (2D)

My room is 02-05



Chess Position



a8	b8	c8	d8	e8	f8	g8	h8
a7	b7	c7	d7	e7	f7	g7	h7
a6	b6	c6	d6	e6	f6	g6	h6
a5	b5	c5	d5	e5	f5	g5	h5
a4	b4	c4	d4	e4	f4	g4	h4
a3	b3	c3	d3	e3	f3	g3	h3
a2	b2	c2	d2	e2	f2	g2	h2
a1	b1	c1	d1	e1	f1	g1	h1

crude-... Search Sheet

Home Insert Page Layout Formulas Data Review View

Normal Page Layout Custom Views Show Zoom 150% Zoom to 100% Freeze Panes View Macros Record Macro

B8 Crude Birth Rate

	A	B	C	D
1	year	level_1	value	
2	1960	Crude Birth Rate	37.5	
3	1961	Crude Birth Rate	35.2	
4	1962	Crude Birth Rate	33.7	
5	1963	Crude Birth Rate	33.2	
6	1964	Crude Birth Rate	31.6	
7	1965	Crude Birth Rate	29.5	
8	1966	Crude Birth Rate	28.3	
9	1967	Crude Birth Rate	25.6	
10	1968	Crude Birth Rate	23.5	
11	1969	Crude Birth Rate	21.8	
12	1970	Crude Birth Rate	22.1	
13	1971	Crude Birth Rate	22.3	
14	1972	Crude Birth Rate	23.1	

crude-birth-rate +

Ready

- Row 8
- Col B
- So, B8

Dimensions

One Dimensional Array, A

Index	Contents
0	'Apple'
1	'John'
2	'Eve'
3	'Mary'
4	'Ian'
5	'Smith'
6	'Kelvin'

A[5] = 'Smith'

Two Dimensional Array, M

	0	1	2	3
0	'Apple'	'Lah'	'Cat'	'Eve'
1	'Hello'	'Pay'	'TV'	'Carl'
2	'What'	'Bank'	'Radio'	'Ada'
3	'Frog'	'Peter'	'Sea'	'Eat'
4	'Job'	'Fry'	'Gym'	'Wow'
5	'Walk'	'Fly'	'Cook'	'Look'

M[4][1] = 'Fry'

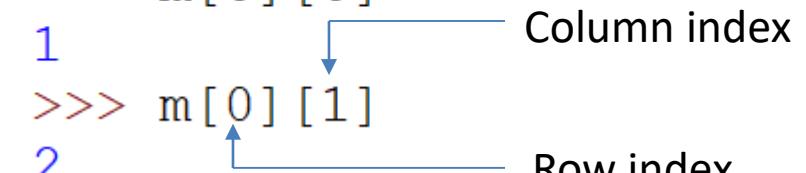
2-Dimensional Array in Python

- 1-d array

```
>>> l = [1,2,3,4,5]  
>>> l[3]  
4
```

- 2-d array

```
>>> m = [[1,2], [3,4]]  
>>> m[0][0]  
1  
>>> m[0][1]  
2  
>>> m[1][0]  
3  
>>> m[1][1]  
4
```

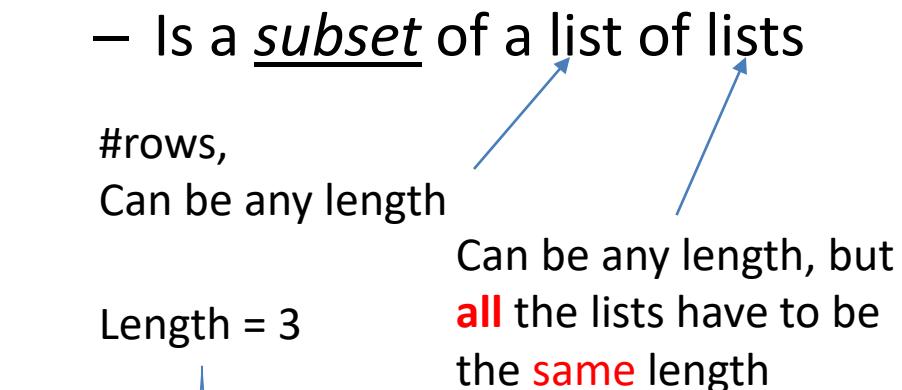
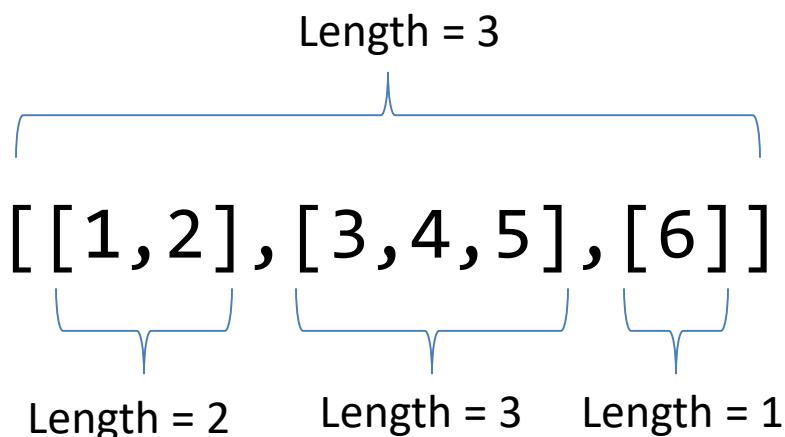


- Equivalent to

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

What is the Difference Between ...

- A list of lists
 - More specifically, all the lists do not have to be the same length
- 2D Array
 - Is a subset of a list of lists



↔

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

A Larger 2-d Array

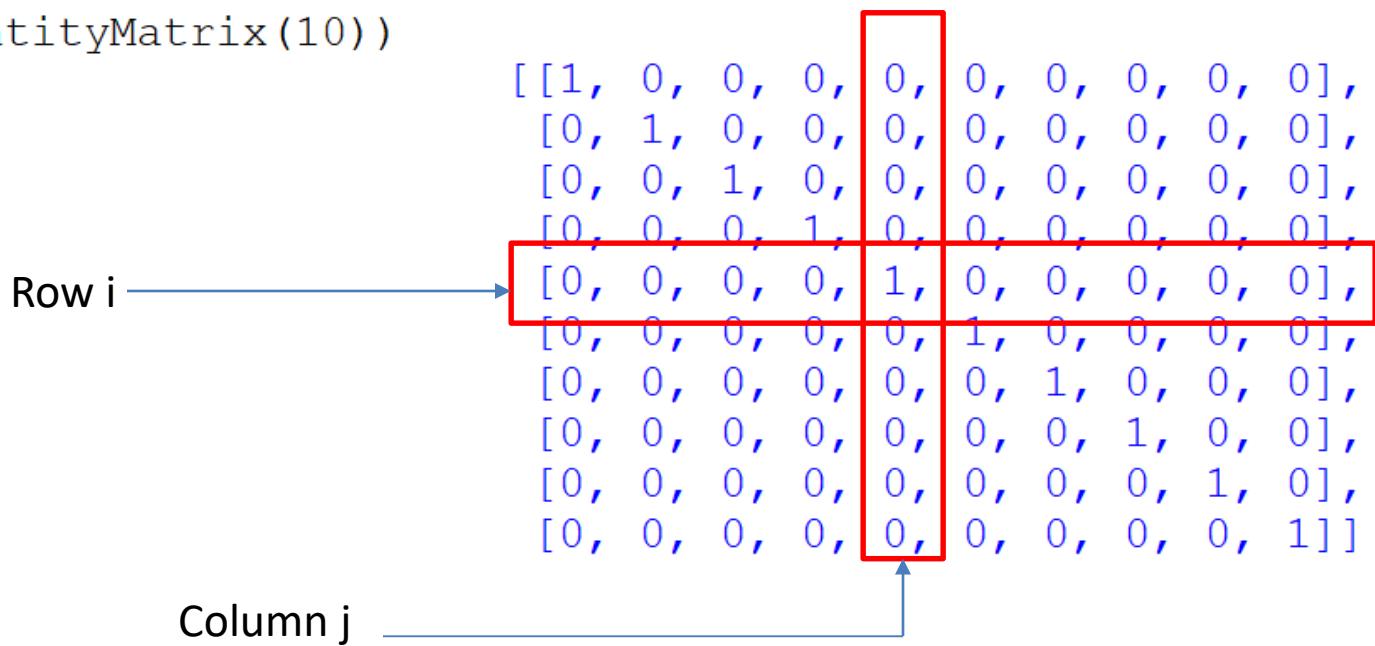
- Creating a 4×10 matrix

```
>>> m2 = [[11,12,13,14,15,16,17,18,19,20], [21,22,23,24,25,26,27,28,29,30],  
           [31,32,33,34,35,36,37,38,39,40], [41,42,43,44,45,46,47,48,49,50]]  
>>> m2  
[[11, 12, 13, 14, 15, 16, 17, 18, 19, 20], [21, 22, 23, 24, 25, 26, 27, 28,  
     29, 30], [31, 32, 33, 34, 35, 36, 37, 38, 39, 40], [41, 42, 43, 44, 45,  
     46, 47, 48, 49, 50]]  
>>> pprint(m2) ← pprint() from  
                           the package  
                           pprint makes  
                           the format nicer  
[[11, 12, 13, 14, 15, 16, 17, 18, 19, 20],  
 [21, 22, 23, 24, 25, 26, 27, 28, 29, 30],  
 [31, 32, 33, 34, 35, 36, 37, 38, 39, 40],  
 [41, 42, 43, 44, 45, 46, 47, 48, 49, 50]]  
>>> len(m2) ← Number of rows  
4  
>>> len(m2[0]) ← Number of columns  
10
```

To Create an $N \times N$ Identical Matrix

```
def identityMatrix(N):
    output = []
    for i in range(N):
        row = []
        for j in range(N):
            row.append(1 if i == j else 0)
        output.append(row)
    return output

pprint(identityMatrix(10))
```



2D Looping

- Let's say we just want to print 0 and 1

```
[[1, 0, 0, 0, 0, 0, 0, 0, 0],  
 [0, 1, 0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 1, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 1, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 1, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 1, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 1, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0, 1, 0],  
 [0, 0, 0, 0, 0, 0, 0, 0, 1]]
```



```
1000000000  
0100000000  
0010000000  
0001000000  
0000100000  
0000010000  
0000001000  
0000000100  
0000000010  
0000000001
```

2D Looping

- Let's say we just want to print 0 and 1

```
def mTightPrint(m) :  
    for i in range(len(m)) :  
        line = ''  
        for j in range(len(m[0])) :  
            line += str(m[i][j])  
        print(line)  
  
mTightPrint(m1)
```

↑
Row i
↑
Column j

j = 0 1 2 3 4 5 6 7 8 9	↓↓↓↓↓↓↓↓↓↓
i = 0 →	1000000000
i = 1 →	0100000000
i = 2 →	0010000000
i = 3 →	0001000000
i = 4 →	0000100000
i = 5 →	0000010000
i = 6 →	0000001000
i = 7 →	0000000100
i = 8 →	0000000010
i = 9 →	0000000001

To Create an R x C Zero Matrix

```
def createZeroMatrix(r,c):  
    output = []  
    for i in range(r):  
        row = []  
        for j in range(c):  
            row.append(0)  
        output.append(row)  
    return output
```

```
>>> m = createZeroMatrix(4,9)  
>>> m  
[[0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0]]  
>>> pprint(m)  
[[0, 0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0, 0]]
```

Accessing Entries

- Remember
 - First row index
 - Then column index

i = 1 →

```
>>> pprint(m)
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
>>> m[1][3] = 9
>>> m[3][7] = 6
>>> pprint(m)
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 9, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 6, 0, 0]]
```

↑
j = 3

2D Array Applications

- Matrix operations
- Recoding 2D data
 - Tables
 - Spatial Data

Matrix Addition

```
def sumMatrices(m1,m2): # Assuming m1 and m2 have the same #r,c
    r = len(m1)
    c = len(m1[0])
    output = createZeroMatrix(r,c)
    for i in range(r):
        for j in range(c):
            output[i][j] = m1[i][j] + m2[i][j]
    return output
```

- How do to Matrix Multiplication or other matrix operations, e.g. inverse, transpose?

```
>>> pprint(m1)
[[4, 7, 6, 1, 1, 2, 2, 1, 8],
 [7, 0, 3, 5, 7, 3, 3, 9, 2],
 [6, 3, 1, 3, 2, 2, 5, 8, 9],
 [4, 0, 3, 2, 9, 3, 6, 8, 0]]
>>> pprint(m2)
[[2, 0, 2, 5, 3, 9, 3, 2, 4],
 [0, 4, 7, 2, 6, 5, 1, 7, 5],
 [6, 0, 6, 7, 5, 9, 0, 7, 7],
 [0, 8, 1, 5, 2, 5, 9, 1, 3]]
>>> pprint(sumMatrices(m1,m2))
[[6, 7, 8, 6, 4, 11, 5, 3, 12],
 [7, 4, 10, 7, 13, 8, 4, 16, 7],
 [12, 3, 7, 10, 7, 11, 5, 15, 16],
 [4, 8, 4, 7, 11, 8, 15, 9, 3]]
```

Tables

- Just like other Spreadsheet applications

Name	Stu. No.	English	Math	Science	Social Studies
John	A1000000A	90	80	100	70
Peter	A1000009D	60	100	60	90
Paul	A1000003C	80	80	70	90
Mary	A1000001B	100	70	80	80

```
>>> records = [['John', 'A1000000A', 90, 80, 100, 70],  
                 ['Peter', 'A1000009D', 60, 100, 60, 90],  
                 ['Paul', 'A1000003C', 80, 80, 70, 90],  
                 ['Mary', 'A1000001B', 100, 70, 80, 80]]  
  
>>> records[2][5] = 100  
  
>>> pprint(records)  
[['John', 'A1000000A', 90, 80, 100, 70],  
 ['Peter', 'A1000009D', 60, 100, 60, 90],  
 ['Paul', 'A1000003C', 80, 80, 70, 100],  
 ['Mary', 'A1000001B', 100, 70, 80, 80]]
```

Tables

- Like in Spreadsheet applications

Name	Stu. No.	English	Math	Science	Social Studies
John	A1000000A	90	80	100	70
Peter	A1000009D	60	100	60	90
Paul	A1000003C	80	80	70	90
Mary	A1000001B	100	70	80	80

- Or, you can setup a more “comprehensible” column index dictionary

```
>>> colIndex = {'name':0, 'SN':1, 'eng':2, 'math':3, 'sci':4, 'sos':5}  
>>> records[3][colIndex['eng']] = 0  
>>> pprint(records)  
[['John', 'A1000000A', 90, 80, 100, 70],  
 ['Peter', 'A1000009D', 60, 100, 60, 90],  
 ['Paul', 'A1000003C', 80, 80, 70, 100],  
 ['Mary', 'A1000001B', 0, 70, 80, 80]]
```

Tables

- You can even append a new record for a new student

```
>>> records.append(['Seon', 'A1000004Z', 70, 80, 70, 80])
>>> pprint(records)
[['John', 'A1000000A', 90, 80, 100, 70],
 ['Peter', 'A1000009D', 60, 100, 60, 90],
 ['Paul', 'A1000003C', 80, 80, 70, 100],
 ['Mary', 'A1000001B', 0, 70, 80, 80],
 ['Seon', 'A1000004Z', 70, 80, 70, 80]]
```

Tables

- Or do any computations on the table
 - E.g. The average of the English scores

```
>>> pprint(records)
[['John', 'A1000000A', 90, 80, 100, 70],
 ['Peter', 'A1000009D', 60, 100, 60, 90],
 ['Paul', 'A1000003C', 80, 80, 70, 100],
 ['Mary', 'A1000001B', 0, 70, 80, 80],
 ['Seon', 'A1000004Z', 70, 80, 70, 80]]
>>> scoreSumEng = 0
>>> for i in range(len(records)):
    scoreSumEng += records[i][colIndex['eng']]
>>> print(scoreSumEng/len(records))
60.0
```

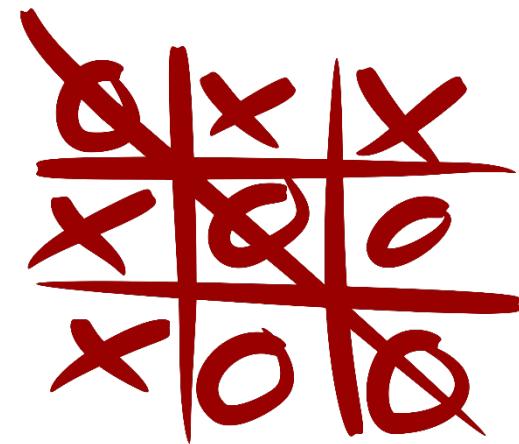
Spatial Data

Tic-tac-toe, Chess, SRPG

Tic-tac-toe

- Tic-tac-toe
 - A game of 3×3 grid
 - How to represent the game?
 - A 3×3 Matrix:

```
game = createZeroMatrix(3,3)
```
- How to "make a move"?
 - You can directly ask the user to input a pair of coordinates, e.g. (1,1)
 - However, sometime it's confusing for users who cannot tell row indices from column indices



Tic-tac-toe

- Because the grid size is very small, we can make it easier for the user by labeling each box with a number from 1 to 9
- For the position pos from 1 to 9, we can map to the coordinates (i, j) as

$$i = (pos-1)//3$$

$$j = (pos-1)\%3$$

1|2|3

4|5|6

7|8|9

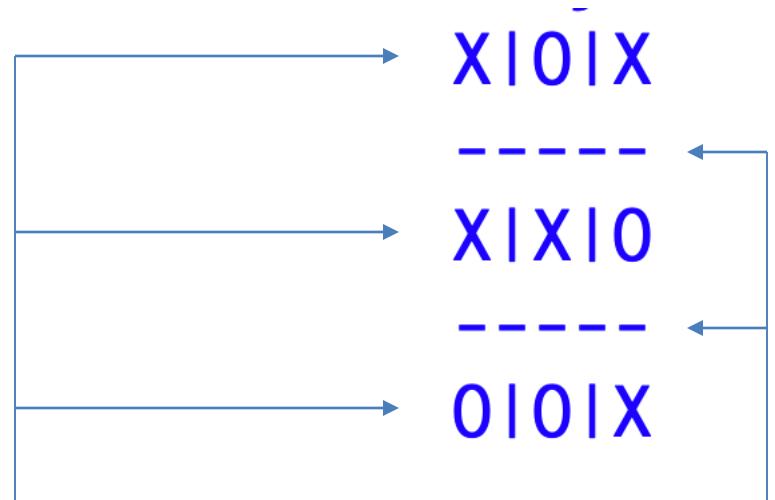
- Challenge

- Given the row and column indices i and j, how to compute the position pos?

A Simple TTT Game

- First, for a matrix **game** that represent the current state of the game, let's make a function to print it nicely

```
def printTTT(game):  
    for i in range(3):  
        print(f'{game[i][0]}|{game[i][1]}|{game[i][2]}')  
        if i !=2:  
            print( '-----')
```



A Simple TTT Game

```
def tttGamePlay():
    game = createZeroMatrix(3,3)
    for i in range(3):
        for j in range(3):
            game[i][j] = i*3+j+1
    player = 0
    piece = ['X', 'O']
    printTTT(game)
    for i in range(9): # Anyhow play 9 times
        pos = int(input(f'Player {piece[player]} move:')) - 1
        game[pos//3][pos%3] = piece[player]
        player = 1 - player
    printTTT(game)
```

Create the game board

A Simple TTT Game

```
def tttGamePlay():
    game = createZeroMatrix(3,3)
    for i in range(3):
        for j in range(3):
            game[i][j] = i*3+j+1
    player = 0
    piece = ['X','0']
    printTTT(game)
    for i in range(9): # Anyhow play 9 times
        pos = int(input(f'Player {piece[player]} move:')) - 1
        game[pos//3][pos%3] = piece[player]
        player = 1 - player
    printTTT(game)
```

Put the number 1 to 9 into the board for display purposes

1|2|3

4|5|6

7|8|9

A Simple TTT Game

```
def tttGamePlay():
    game = createZeroMatrix(3,3)
    for i in range(3):
        for j in range(3):
            game[i][j] = i*3+j+1
    player = 0
    piece = ['X','O']
    printTTT(game)
    for i in range(9): # Anyhow play 9 times
        pos = int(input(f'Player {piece[player]} move:')) - 1
        game[pos//3][pos%3] = piece[player]
        player = 1 - player
    printTTT(game)
```

There are two players,
Player 0 and Player 1
Player 0 uses 'X'
Player 1 uses 'O'

The first player is Player 0

And they will take turns

A Simple TTT Game

```
def tttGamePlay():
    game = createZeroMatrix(3,3)
    for i in range(3):
        for j in range(3):
            game[i][j] = i*3+j+1
    player = 0
    piece = ['X','0']
    printTTT(game)
    for i in range(9): # Anyhow play 9 times
        pos = int(input(f'Player {piece[player]} move:')) - 1
        game[pos//3][pos%3] = piece[player]
        player = 1 - player
    printTTT(game)
```

Anyhow play 9 times
(We didn't implement any error or winning checking yet)

Gameplay

1|2|3

4|5|6

7|8|9

Player X move:5

1|2|3

4|X|6

7|8|9

Player 0 move:3

1|2|0

4|X|6

7|8|9

Player X move:6

1|2|0

4|X|X

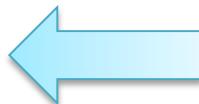
7|8|9

Chess

```
def createChessGame():
    game = createZeroMatrix(8, 8)
    for j in range(8):
        game[1][j] = 'p'
        game[6][j] = 'P'
    game[7] = ['R', 'K', 'B', 'Q', 'E', 'B', 'K', 'R']
    game[0] = list(map(lambda x:x.lower(), game[7]))
    return game
```

```
game = createChessGame()
mTightPrint(game)
```

```
rkbqebkr
pppppppp
00000000
00000000
00000000
00000000
PPPPPPPP
RKBQEBKR
```



Other Text Based Chess

	A	B	C	D	E	F	G	H
8	r	n	b	q	k	b	n	r
7	p	p	p	p	p	p	p	p
6
5
4
3
2	P	P	P	P	P	P	P	P
1	R	N	B	Q	K	B	N	R

	A	B	C	D	E	F	G	H
8	#	#	#	#	#	#	#	#
7	R	N	B	Q	K	B	N	R
6	#	#	#	#	#	#	#	#
5	P	P	P	P	P	P	P	P
4	#	#	#	#	#	#	#	#
3	#	#	#	#	#	#	#	#
2	#	#	#	#	#	#	#	#
1	#	P	#	P	#	P	#	P
0	R	N	B	Q	K	B	N	R

```
worldwindow@Blackice: /hom
Blackice:/home/worldwindow/c++_projekte/cchess# ./cc
### ConChess - A console based chess game #####
Author: Christian Guckelsberger Version 0.01

---A-----B-----C-----D-----E-----F-----G-----H---
1 |Castle1| Horse1 | Bishop1 | Queen  | King   | Bishop2 | Horse2 | Castle2 |
2 | Pawn1 | Pawn2  | Pawn3  | Pawn4  | Pawn5 | Pawn6  | Pawn7 | Pawn8  |
3 |-----|-----|-----|-----|-----|-----|-----|-----|
4 |-----|-----|-----|-----|-----|-----|-----|-----|
5 |-----|-----|-----|-----|-----|-----|-----|-----|
6 |-----|-----|-----|-----|-----|-----|-----|-----|
7 | Pawn1 | Pawn2 | Pawn3 | Pawn4 | Pawn5 | Pawn6 | Pawn7 | Pawn8 |
8 |Castle1| Horse1| Bishop1| King   | Queen  | Bishop2| Horse2| Castle2 |

++Player1++
Please enter the name of the figure you want to move: [
```

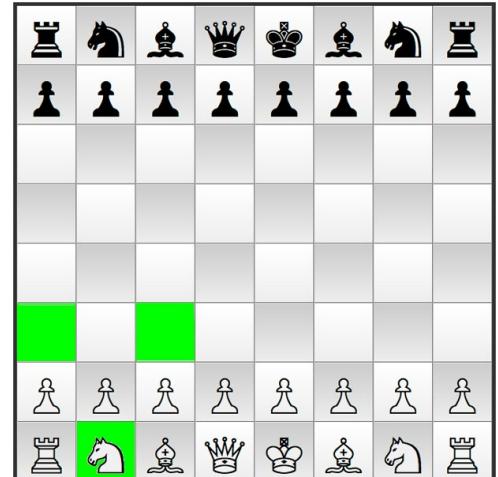
Chess

- How to make a move?
- E.g. moving a knight
 - Maybe

game[7][1] = 0

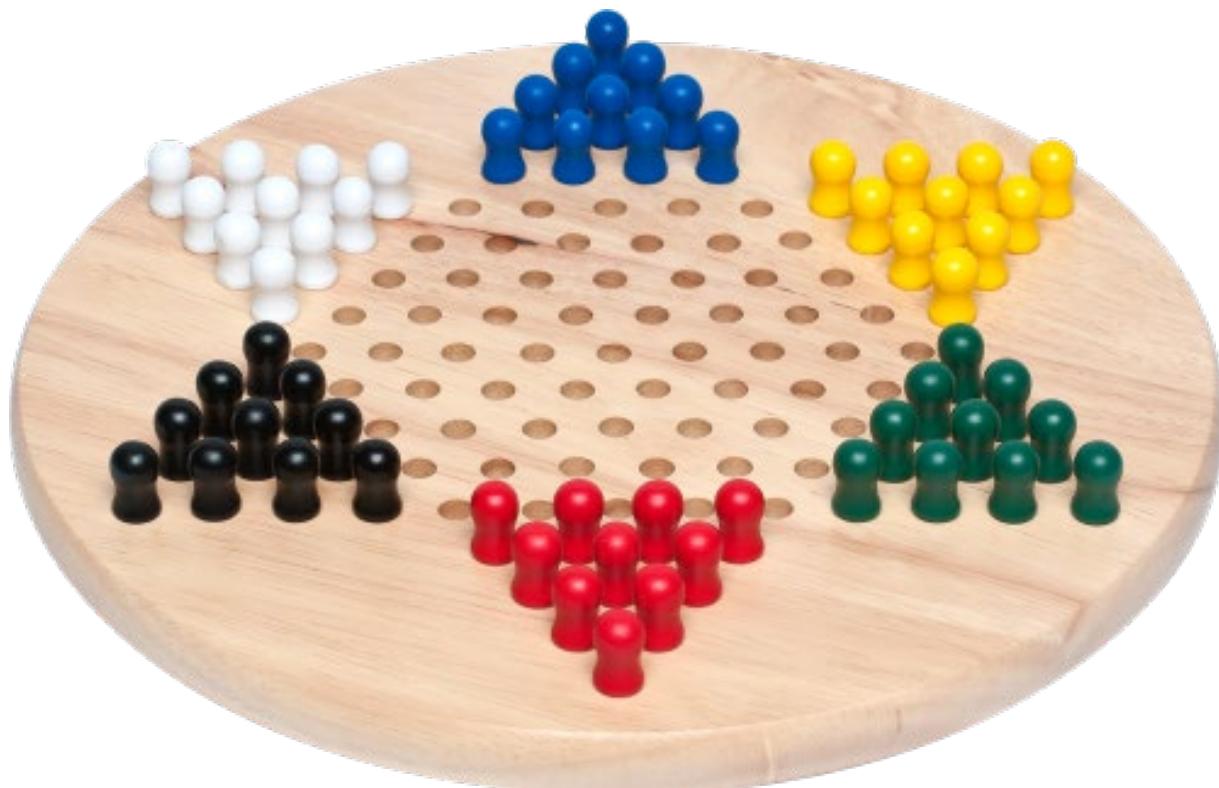
game[5][2] = 'K'

rkbqebkr
pppppppp
00000000
00000000
00000000
00000000
PPPPPPPP
RKBQEBKR



Chinese Checkers?

- How to represent the board?

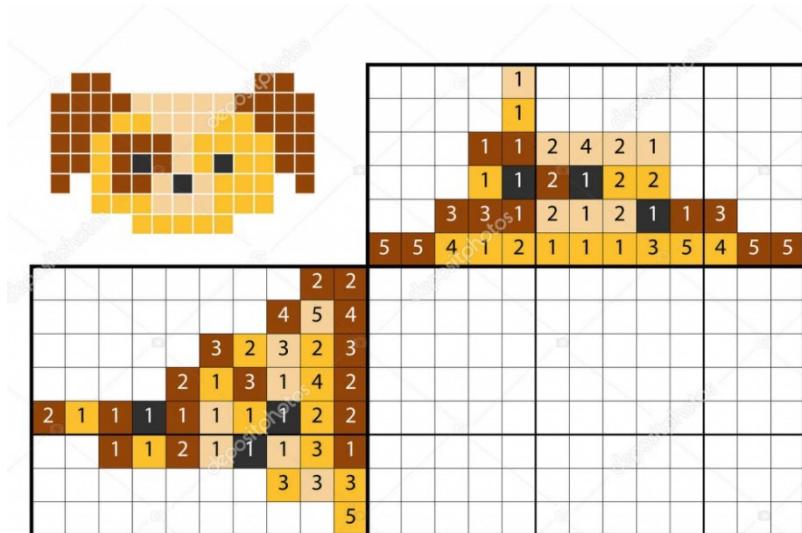


3D Chess

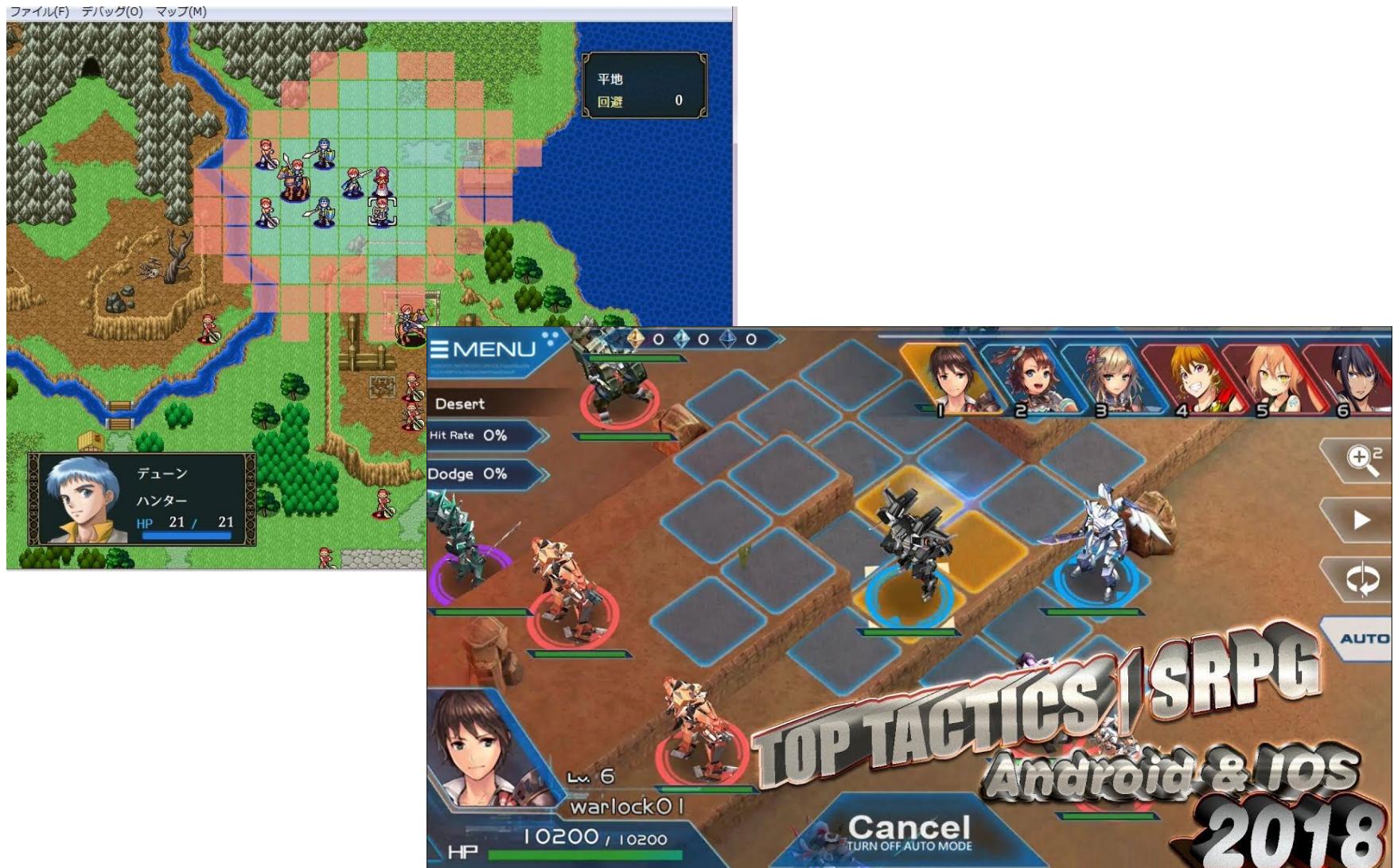


Other 2D Array Games

5	3		7			
6		1	9	5		
	9	8			6	
8			6			3
4		8	3			1
7		2			6	
6			2	8		
	4	1	9			5
		8		7	9	

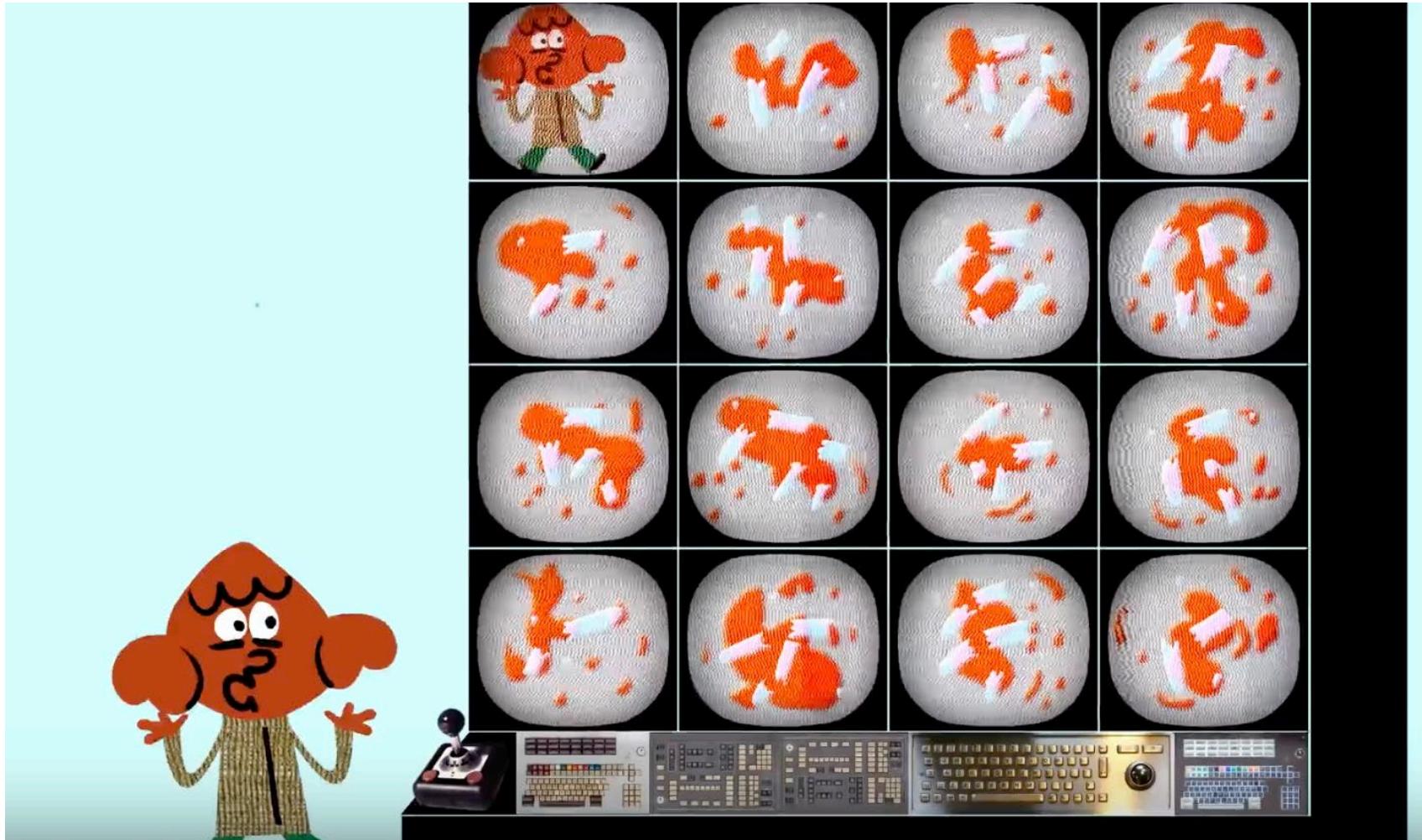


RPG, SRPG



Game of Life

- <https://bitstorm.org/gameoflife/>



- Virus Riddle

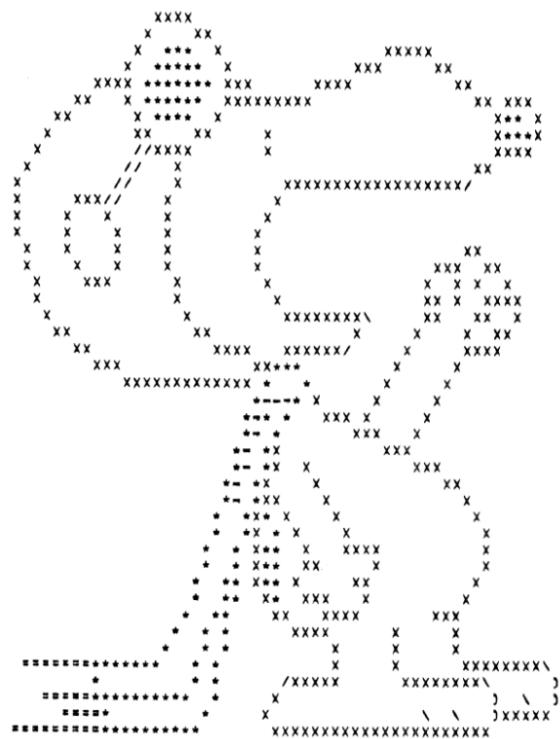
circlePic()

```
def circlePic():
    l = 30
    center = 1/2
    pic = createZeroMatrix(l, l)
    for i in range(l):
        for j in range(l):
            if (l/3)**2 < (i-center)**2 + (j-center)**2 < (l/2)**2:
                pic[i][j] = "#"
            else:
                pic[i][j] = "."
    mTightPrint(pic)
```



The image shows a decorative border made of a repeating pattern of black dots and diagonal lines. The pattern forms a diamond shape, with each side composed of a series of diagonal lines pointing towards the center. The spaces between these lines are filled with black dots. This pattern is repeated around the entire border of the page.

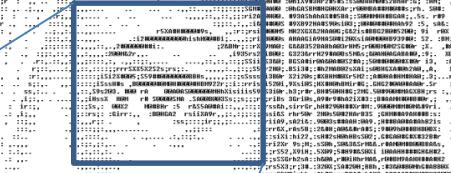
ASCII Arts

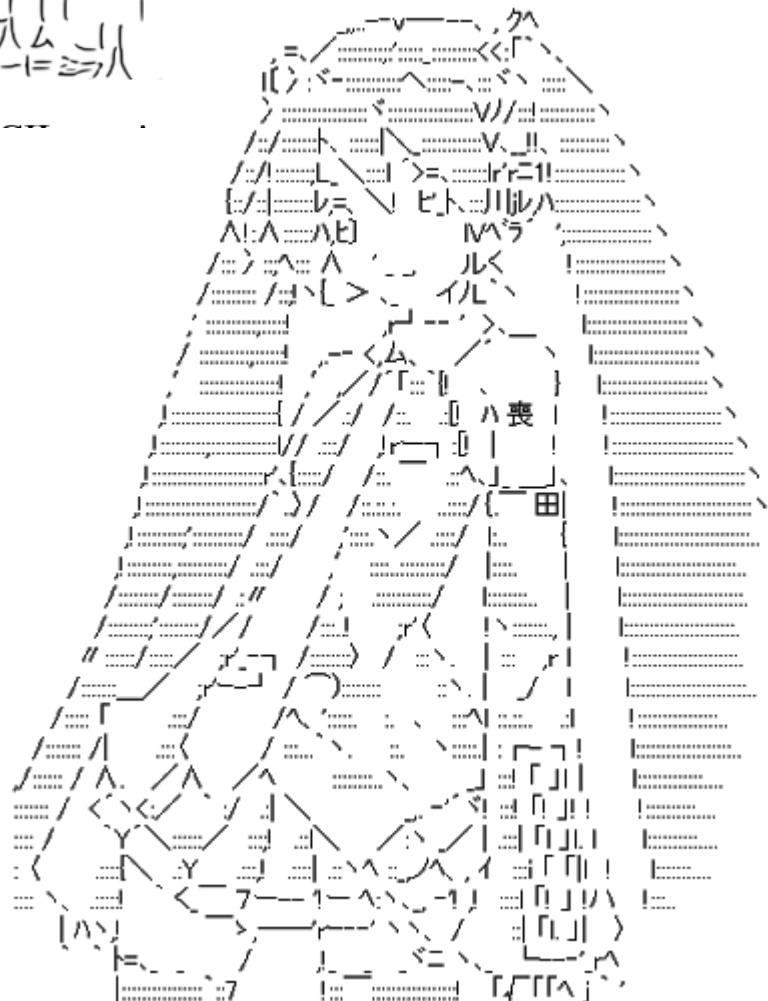
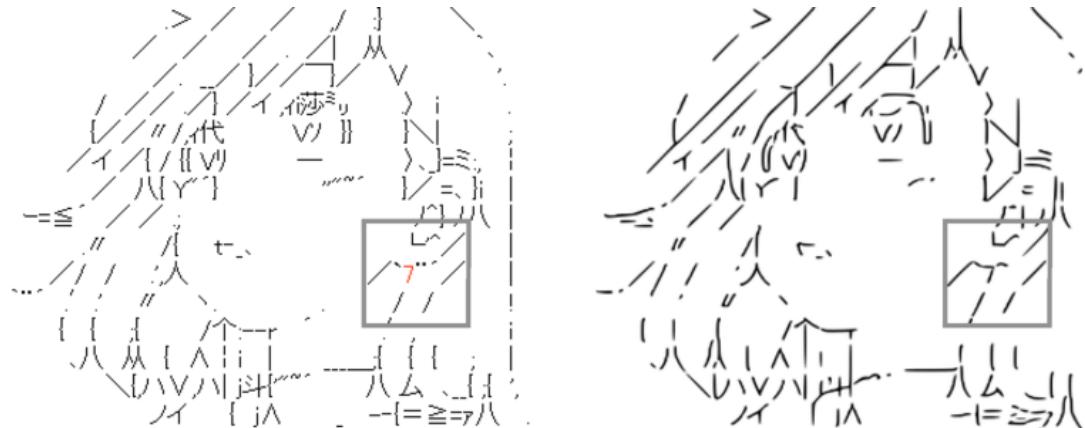


```

      _.-.----BBBBBBBBB--_
      ..-BBB"''' ooOOOoop"BBB\_
      .cBBB.:BBBBBBBBBBBp,.....BBpBB\_
      .pBB/BBBBBBBBBBBBP/" " " .-B\Bl\
      .PB/BBBBBBBBBBBBB/ /M\.m\ }!!}BII
      .Pd/BBBBBBBBBBBBBBB| /M'\../|m\ }!!}B
      __p.../Bq/BBBBBBBBBBBBBBBBB|---"---"--->,)BpBb-.
      .-BB"''"/BBBBBBBBBBB..---" bBBBBBBBBBBBBBBBBBBBb,";BBb\.
      .Pb"BBBBBBBBBBBBP." oBBBBBBBBBBBBBBBBBBBPPPPPPPBP\BB:\_
      [B{BBBBBBBBBBB.P""P"-----BBBBBBBBBBBB-BB
      {B"}o*****--*BBBIBBBBBBBBBB/BP\BBBB/bp/|BBBBBBBBBBB.*
      {B/bBBB..PBBB\|BBB\|BBB|BB|"-*BB"BB| BU
      {B[BBBBB"\|B\|BBBb}}BBBBB | |B|}BBBB _---"---"BBP\
      {B{BBBBb}*|B\|O}|"/"BB\...\\BBBBBBB*BBBBBBBBB\B\
      \B\|BBB,,,*=BBB\|BBB/0/*|\|BBBBBBBBBBBBBBB|B}
      *B\..BBBBBBBBBBBBBBB|{BBB*-...BBBBBBBBBBBBBBB|B}
      *B*****\|BBBBBBBBB\B\BBBBBb\,BBBBBBBBBBB|B/
      *>BB.****\|BBBBBBBBB|{BBBBBBb\P..PBBB|BP/
      /BP/bB*/*\|\\PPBBBB\|PPPPPPPPPBo>BBBB"
      /B|//"/BBBB\|/\^BboBbo-.,,Bb,,B,,,BP"'''bbBB***b_
      /B|//"/BBBBP/-*||ooo.BBB\||||* *BBP* \B\ }B}
      /B}\*,B|BBB\|****\* *||PP\||||/_B/||| 10\ /B\
      /B'\|BBP// * ...B\|..OB\|****\| \ \| /0\ \B\
      |B|B|B|BBP// H\|**\|V\| * "-ooP/ \B\
      |B\Bb\|B0/| )|||B\|B\|V\| : \| /B}
      \B\B\B\|V\|V\| 1|||)B\|B\|V\| \| /B\
      \B\B\B\|B\|V\| -.-. /B\B\|V\|B/PP\| - -lo_ _b\|/
      ,B"---oo_-++,- *|||B/BBB/BB,
      oBB(BBBBBoo\|,...b__bIII*dBP//B***Bboo
      .bB"---o\|...BBB))bboooooB**oo*boOo-oB-
      /B/BBBBBBBBB,---BPPPPP***Booooooooooooo
      |B|BBBBBBBBBBB)B) (|BBB BBBB BBBB B
      \B|BBBBBBBBBBB)B) (0BBBBBB BBBB B
      \B|BBBBBBBBBBB)B/ \B|BBBBBBBBBBB
      \B|PPPPPPPPP/BII/ (B(*****
      \B\|*****\|,oB) "----BBBBBB+++++B
      "BBBBBBBBBBP**

```





Wait...

- What if I put three colors as each of the item in my 2D array?

```
[[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],  
 [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],  
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]]
```

[100,20
0,100]
Red, green, blue
= [100,200,100]

2D and 3D Arrays

2D Arrays

- A list of lists
- A vehicle loading with vehicles
 - And each vehicle contains some passengers



3D Arrays

- A list of lists of lists
- A vehicle loading with vehicles loading with vehicles
 - And each vehicle contains some passengers



$2+1 = 3D$ Array as a Picture

```
pic = [[[0,0,255], [255,0,0], [0,0,255]],  
        [[255,0,0], [255,0,0], [255,0,0]],  
        [[0,0,255], [255,0,0], [0,0,255]]]
```

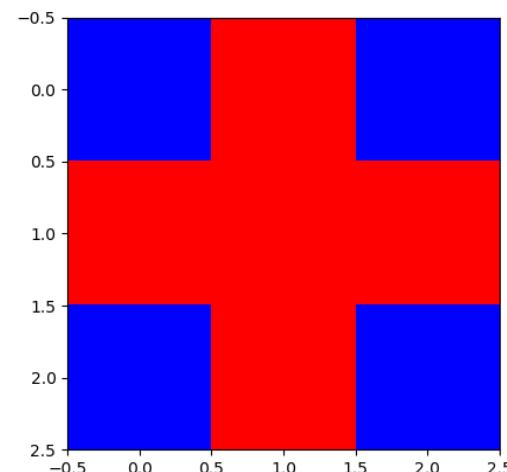
```
import matplotlib.pyplot as plt  
plt.imshow(pic)  
plt.show()
```

One pixel is a list of three values of red, blue and green.

Usually ranging from 0 to 255

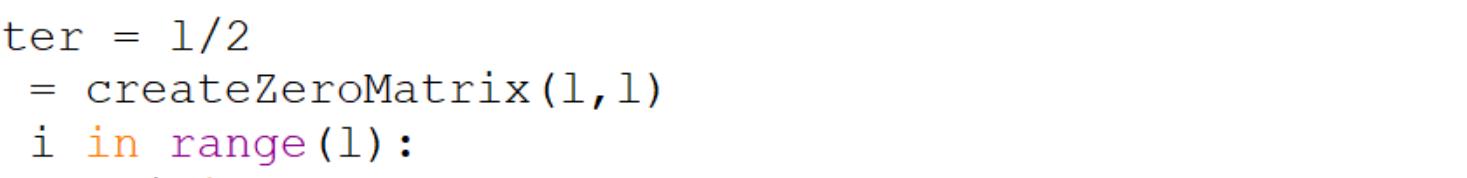
`imshow()` is to draw a picture in bitmap

- 3×3 ($\times 3$ for colors) pixels (zoomed)



Recap: circlePic()

```
def circlePic():
    l = 30
    center = l/2
    pic = createZeroMatrix(l,l)
    for i in range(l):
        for j in range(l):
            if (l/3)**2 < (i-center)**2 + (j-center)**2 < (l/2)**2:
                pic[i][j] = "#"
            else:
                pic[i][j] = "."
    mTightPrint(pic)
```



A large grid of blue '#' symbols on a white background, arranged in a pattern of horizontal and vertical lines. The grid consists of approximately 100 columns and 100 rows of '#' characters, creating a dense, dotted texture.

```
def circlePic2():
    l = 300
    center = l/2
    pic = createZeroMatrix(l,l)
    for i in range(l):
        for j in range(l):
            if (l/3)**2 < (i-center)**2 + (j-center)**2 < (l/2)**2:
                pic[i][j] = [0,0,255]
            else:
                pic[i][j] = [255,0,0]
    plt.imshow(pic)
    plt.show()
```

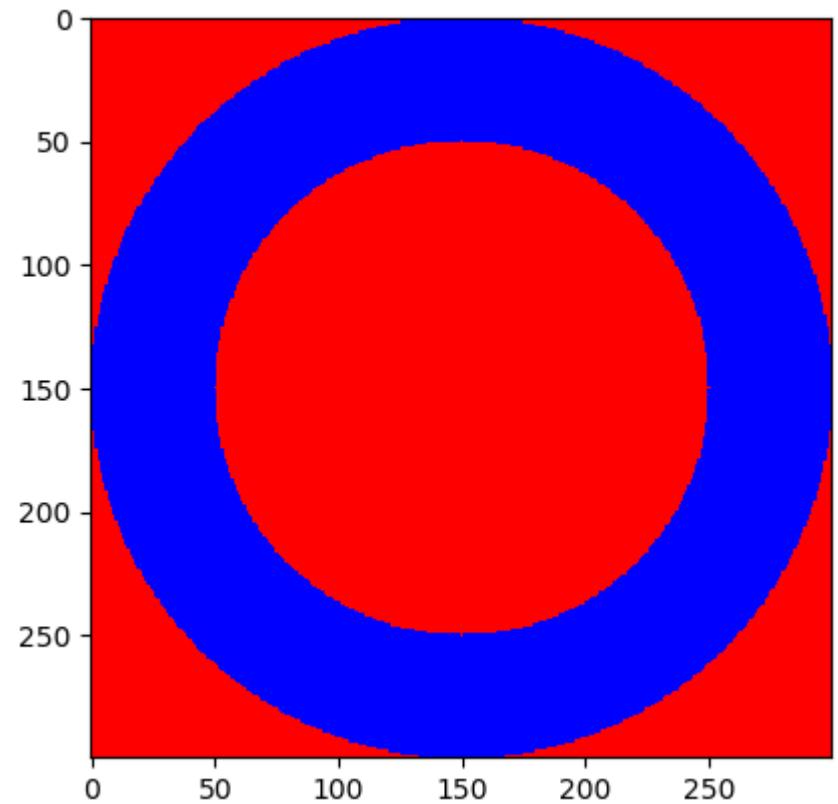


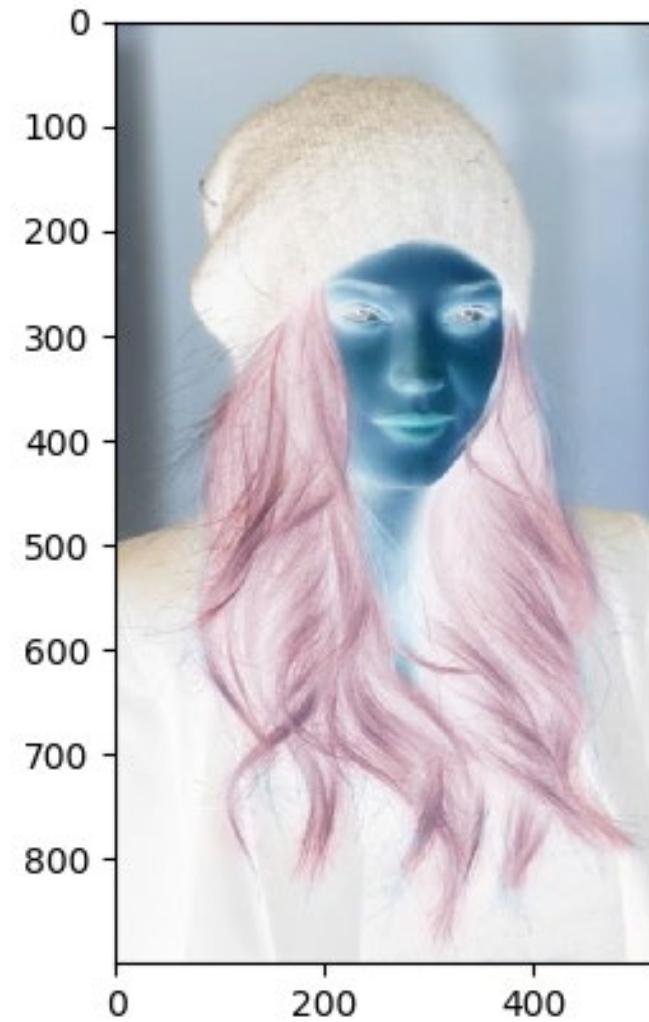
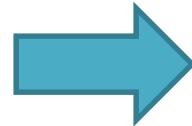
Image Processing

Using the `imageio` package to read in a photo as a 3D

```
>>> import imageio  
>>> import matplotlib.pyplot as plt  
>>> pic = imageio.imread('green hair girl.jpg')  
>>> len(pic)  
900  
>>> len(pic[0])  
517  
>>> len(pic[0][0])  
3  
>>> plt.imshow(pic)  
<matplotlib.image.AxesImage object at  
>>> plt.show()
```



Negative Image

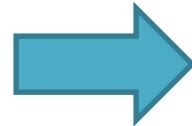


Negative Image

```
import imageio
def negImage(imgFile):
    pic = imageio.imread(imgFile)
    plt.imshow(pic)
    plt.show()          The picture has h x w numbers of pixels
                        (h = #rows, w = #columns)
    h = len(pic)       ← Loop through each pixel with indices (i, j)
    w = len(pic[0])
    for i in range(h):
        for j in range(w):
            for k in range(3):
                pic[i][j][k] = 255 - pic[i][j][k]

    plt.imshow(pic)
    plt.show()
```

Image Processing



Dyeing Hair

```
import imageio

def dye_hair():
    pic = imageio.imread('green hair girl.jpg')
    plt.imshow(pic)
    plt.show()

    h = len(pic)
    w = len(pic[0])
    for i in range(h):
        for j in range(w):
            R = pic[i][j][0]
            G = pic[i][j][1]
            B = pic[i][j][2]
            if G > R and G > B:
                pic[i][j] = [R*2, G*0.2, B*0.8]
    plt.imshow(pic)
    plt.show()
```

Loop through each pixel with indices (i, j)

If `pic[i][j]` has more green, change it to purple

Multi-dimensional Array

$2+1 = 3D$ Array as a Picture

```
pic = [[[0,0,255], [255,0,0], [0,0,255]],  
        [[255,0,0], [255,0,0], [255,0,0]],  
        [[0,0,255], [255,0,0], [0,0,255]]]
```

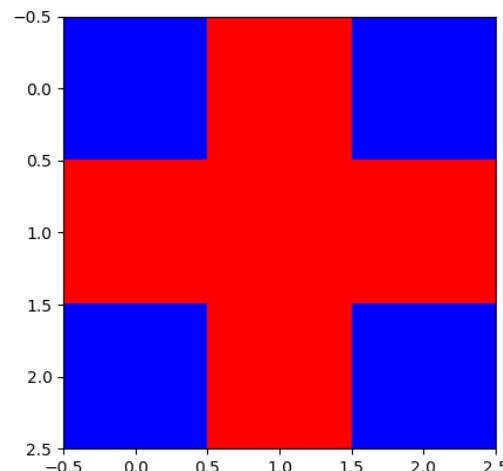
```
import matplotlib.pyplot as plt  
plt.imshow(pic)  
plt.show()
```

One pixel is a list of three values of red, blue and green.

Usually ranging from 0 to 255

`imshow()` is to draw a picture in bitmap

- 3×3 pixels (zoomed)



Dimensions

One Dimensional Array, A

Index	Contents
0	'Apple'
1	'John'
2	'Eve'
3	'Mary'
4	'Ian'
5	'Smith'
6	'Kelvin'

$A[5] = \text{'Smith'}$

Two Dimensional Array, M

	0	1	2	3
0	'Apple'	'Lah'	'Cat'	'Eve'
1	'Hello'	'Pay'	'TV'	'Carl'
2	'What'	'Bank'	'Radio'	'Ada'
3	'Frog'	'Peter'	'Sea'	'Eat'
4	'Job'	'Fry'	'Gym'	'Wow'
5	'Walk'	'Fly'	'Cook'	'Look'

$M[4][1] = \text{'Fry'}$

Multi-Dimensional Array

- A **three** dimensional array

The diagram illustrates a three-dimensional array structure. It consists of a main 10x5 grid of numbers, with each cell containing a value like 65,340 or 418,482. A red arrow points diagonally upwards from the bottom-left corner of the grid towards the top-right corner, indicating the depth dimension. Another red arrow points horizontally across the top row of the grid, indicating the width dimension. A vertical red arrow points downwards from the bottom-left corner, indicating the height dimension.

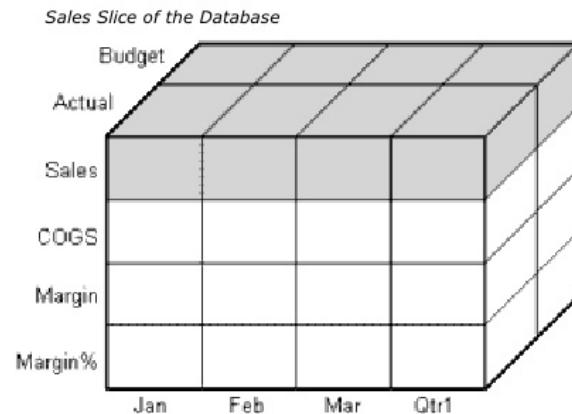
Index	0	1	2	3	4
0	65,340	12,483	138,189	902,960	633,877
1	5,246	424,642	650,380	821,254	866,122
2	89,678	236,781	601,691	329,274	913,534
3	103,902	4,567	733,611	263,010	85,550
4	2,778	658,305	128,788	978,155	620,702
5	45,024	55,058	705,586	89,672	384,605
6	780	47,538	523,784	556,801	617,107
7	32,667	350,890	834,753	638,108	85,188
8	56,083	145,582	775,040	548,322	756,587
9	41,123	543,542	537,738	513,048	418,482

Fancy Terminology in Business

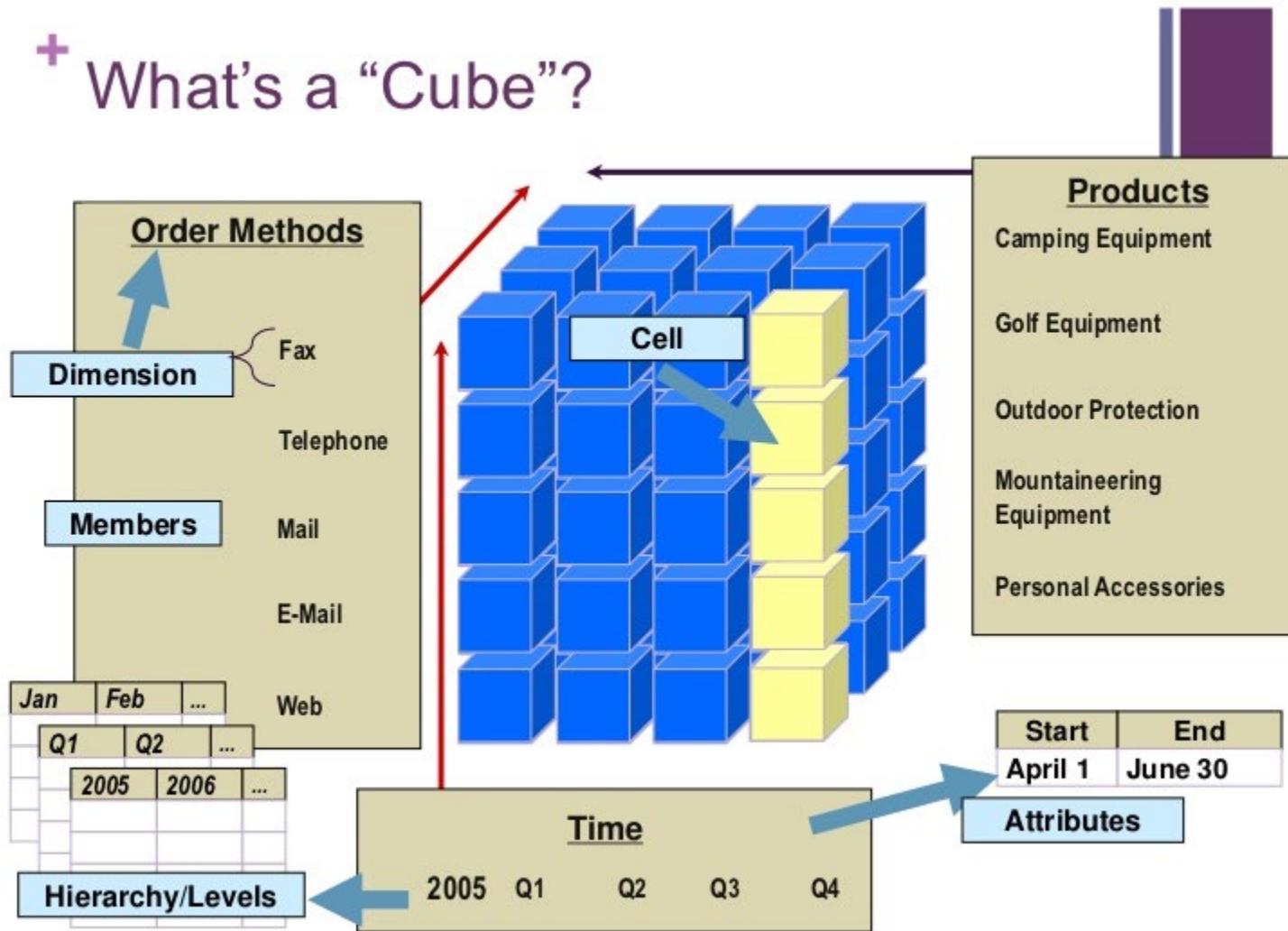
- Multidimensional Data Model
- Multidimensional Analysis



The shaded cells is called a slice illustrate that, when you refer to Sales, you are referring to the portion of the database containing eight Sales values.

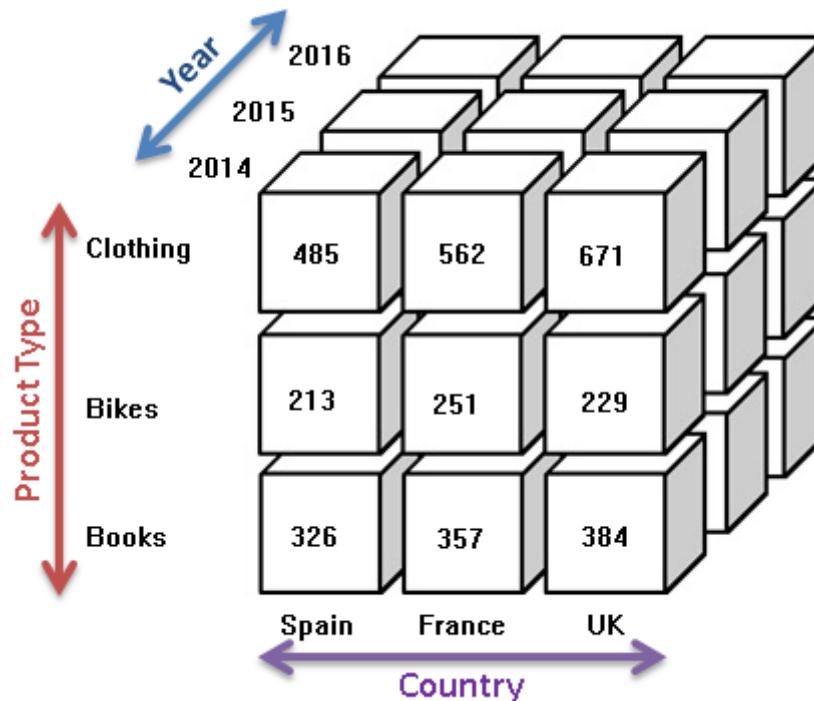


+ What's a “Cube”?



Sales Cube

- Product Type x Year x Country



- 4th Dimensional Sales Cube:
 - Product Type x Year x Country x {Predicted vs actual}

Multi-Dimensional Array

- Year x Ethnic Group x {death/birth}

The screenshot shows a Microsoft Excel spreadsheet titled "crude-birth-death-natural-increase-rates-by-ethnic-group-from-1971-2015.csv". The data is presented in a table with columns labeled A through I. The first few rows of data are as follows:

	A	B	C	D	F	G	H	I
1	year	ethnic_group	crude_death_rate	crude_birth_rate				
2	1971	Chinese	5.5	22.1				
3	1971	Malays	4.6	22.9				
4	1971	Indians	5.9	21.1				
5	1971	Others	5.6	29.2				
6	1972	Chinese	5.4	23				
7	1972	Malays	4.6	23.6				
8	1972	Indians	6.1	21.1				
9	1972	Others	6	29.1				
10	1973	Chinese	5.5	22.3				
11	1973	Malays	4.6	21.2				
12	1973	Indians	6.6	19				
13	1973	Others	5.7	29.2				

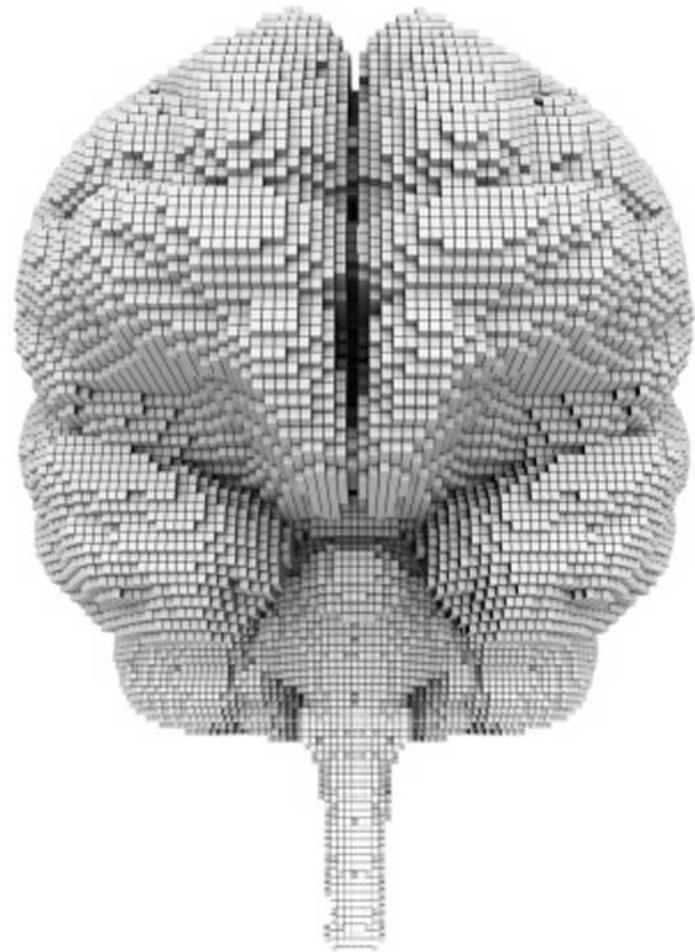
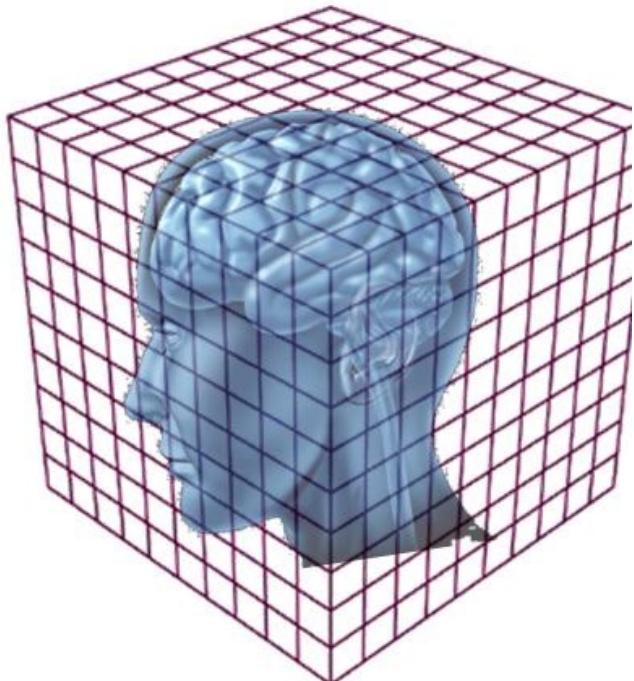
And.... how can we forget?!





Voxels

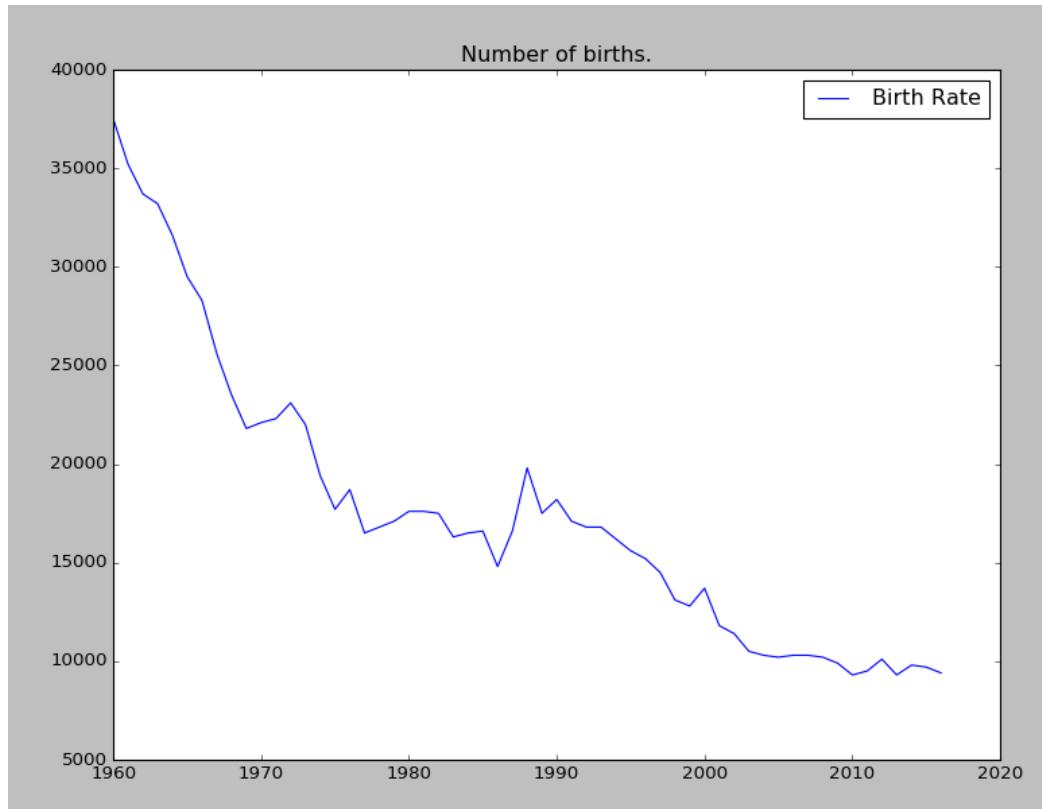
- 2D element: Pixels
- 3D element: Voxels



Reading a CSV File

2D Array x CSV

- Remember last time, we plotted the Singapore birth rate with the data in a CSV file



```
import matplotlib.pyplot as plt

def plot_birth_rate():
    with open('crude-birth-rate.csv') as f:
        f.readline()
        year = []
        num_birth = []
        for line in f:
            list_form = line.rstrip('\n').split(',')
            year.append(int(list_form[0]))
            num_birth.append(float(list_form[2])*1000)

    plt.plot(year, num_birth, label="Birth Rate")
    plt.legend(loc="upper right")
    plt.title('Number of births.')
    plt.show()

plot_birth_rate()
```

Reading CSV Files

Read in a
CSV file
into a list

Create a CSV File
Reader

```
>>> from pprint import pprint
>>> birth_file = open('crude-birth-rate.csv')
>>> birth_file_reader = csv.reader(birth_file)
>>> birth_data = list(birth_file_reader)
```

```
>>> pprint(birth_data)
[['year', 'level_1', 'value'],
 ['1960', 'Crude Birth Rate', '37.5'],
 ['1961', 'Crude Birth Rate', '35.2'],
 ['1962', 'Crude Birth Rate', '33.7'],
 ['1963', 'Crude Birth Rate', '33.2'],
 ['1964', 'Crude Birth Rate', '31.6'],
 ['1965', 'Crude Birth Rate', '29.5'],
 ['1966', 'Crude Birth Rate', '28.3'],
 ['1967', 'Crude Birth Rate', '25.6'],
 ['1968', 'Crude Birth Rate', '23.5'],
 ['1969', 'Crude Birth Rate', '21.8']]
```

Remember these
four lines of code

No need for all
those string
strip(), split() etc.



Are you
pulling
my leg?

Caution

- Again, using lists or tuples?
- Again, the old list referencing problem

```
>>> row = [1, 2, 3]
>>> m = []
>>> m.append(row)
>>> m.append(row)
>>> m.append(row)
>>> m
[[1, 2, 3], [1, 2, 3], [1, 2, 3]]
>>> m[0][0] = 999
>>> m
[[999, 2, 3], [999, 2, 3], [999, 2, 3]] 
>>> m2 = m
>>> m2
[[999, 2, 3], [999, 2, 3], [999, 2, 3]]
>>> m2[2][2] = 'X'
>>> m2
[[999, 2, 'X'], [999, 2, 'X'], [999, 2, 'X']] 
```

Conclusion

- Dimensionality is not that difficult
 - If you can move from 1D to 2D, moving to 3D or ND is not that difficult
 - *In bosonic string theory, spacetime is 26-dimensional, while in superstring theory it is 10-dimensional, and in M-theory it is 11-dimensional.*
- MD-arrays solve a lot of problems
 - Math, images, engineering, robotics, AI,

Object-Oriented Programming

OOP

Recap: Tables

- Just like other Spreadsheet applications

Name	Stu. No.	English	Math	Science	Social Studies
John	A1000000A	90	80	100	70
Peter	A1000009D	60	100	60	90
Paul	A1000003C	80	80	70	90
Mary	A1000001B	100	70	80	80

```
>>> records = [['John', 'A1000000A', 90, 80, 100, 70],  
               ['Peter', 'A1000009D', 60, 100, 60, 90],  
               ['Paul', 'A1000003C', 80, 80, 70, 90],  
               ['Mary', 'A1000001B', 100, 70, 80, 80]]
```

- How about more “attributes”

Name	Stu. No.	Gender	Year	English	Math	Science	Social Studies
John	A1000000A	M	2018	90	80	100	70
Peter	A1000009D	M	2018	60	100	60	90
Paul	A1000003C	M	2017	80	80	70	90
Mary	A1000001B	F	2019	100	70	80	80

NUS Application

A. Personal Particulars		
Name (Please underline Family Name):	NRIC/Passport No.:	Gender: Male / Female #
Faculty / School and Department:	Matriculation No.:	Nationality:
Residential Address:	Home Tel: Mobile:	NUS e-mail address: Other e-mail address:
Mailing Address (if different from above):	Date of Birth: Ethnicity:	Please list the countries which you have visited for: Holidays: Others:
Names of Parent(s)/ Guardian(s)/ Next of Kin#:	Relationship:	Total Annual Household Income (of family members who are financially supporting the applicant in S\$): Number of family members supported by this income:
Number of family members:	Number of Siblings:	
Residential Address of Parent/ Guardian/ Next of Kin#:	Office Tel: Home Tel:	Occupation: Designation:
Mailing Address (if different from above):	Mobile:	Employer:

Stored as a 2D Array

- Like in Spreadsheet applications

Name	Stu. No.	English	Math	Science	Social Studies
John	A1000000A	90	80	100	70
Peter	A1000009D	60	100	60	90
Paul	A1000003C	80	80	70	90
Mary	A1000001B	100	70	80	80

- Or, you can setup a more “comprehensible” column index dictionary

```
>>> colIndex = {'name':0, 'SN':1, 'eng':2, 'math':3, 'sci':4,  
'sos':5}
```

```
>>> records[3][colIndex['eng']] = 0
```

```
>>> pprint(records)
```

```
[['John', 'A1000000A', 90, 80, 100, 70],  
 ['Peter', 'A1000009D', 60, 100, 60, 90],  
 ['Paul', 'A1000003C', 80, 80, 70, 100],  
 ['Mary', 'A1000001B', 0, 70, 80, 80]]
```

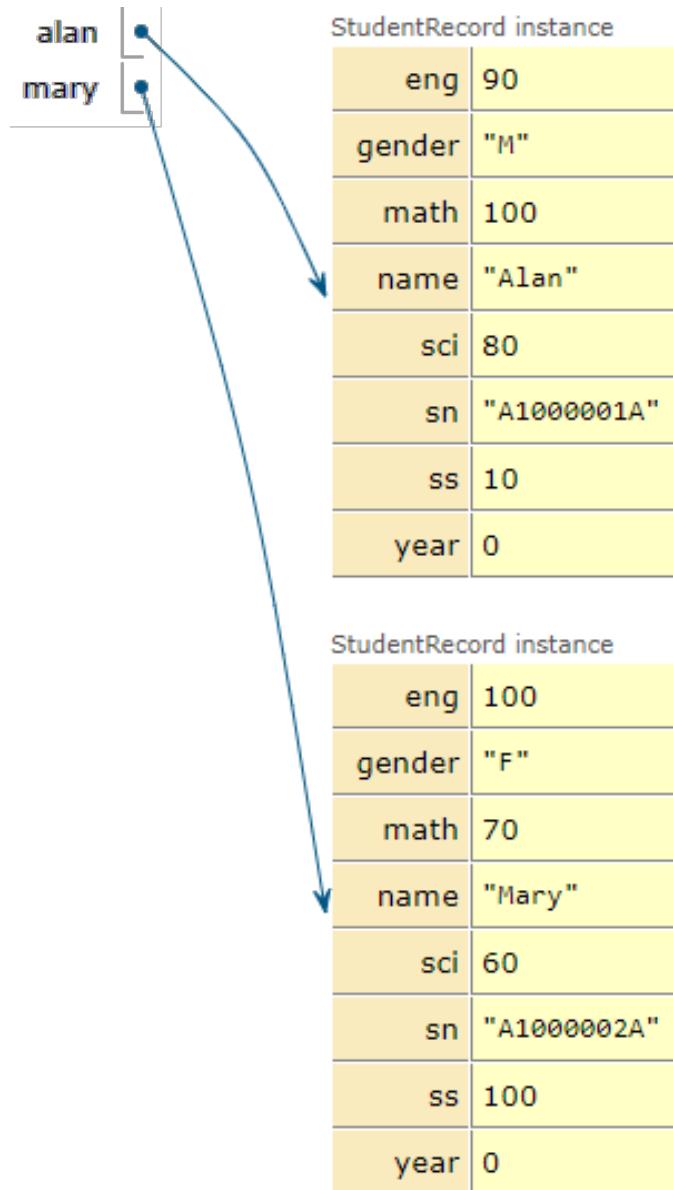
MR. CLUMSY

by Roger Hargreaves



Isn't it nicer?

```
>>> alan.name  
'Alan'  
>>> alan.gender  
'M'  
>>> alan.sci  
80  
>>> alan.math  
100  
>>> mary.name  
'Mary'  
>>> mary.sn  
'A1000002A'  
>>> allStudents = [alan,mary]  
>>> allStudents[0].name  
'Alan'  
>>> allStudents[1].sn  
'A1000002A'
```



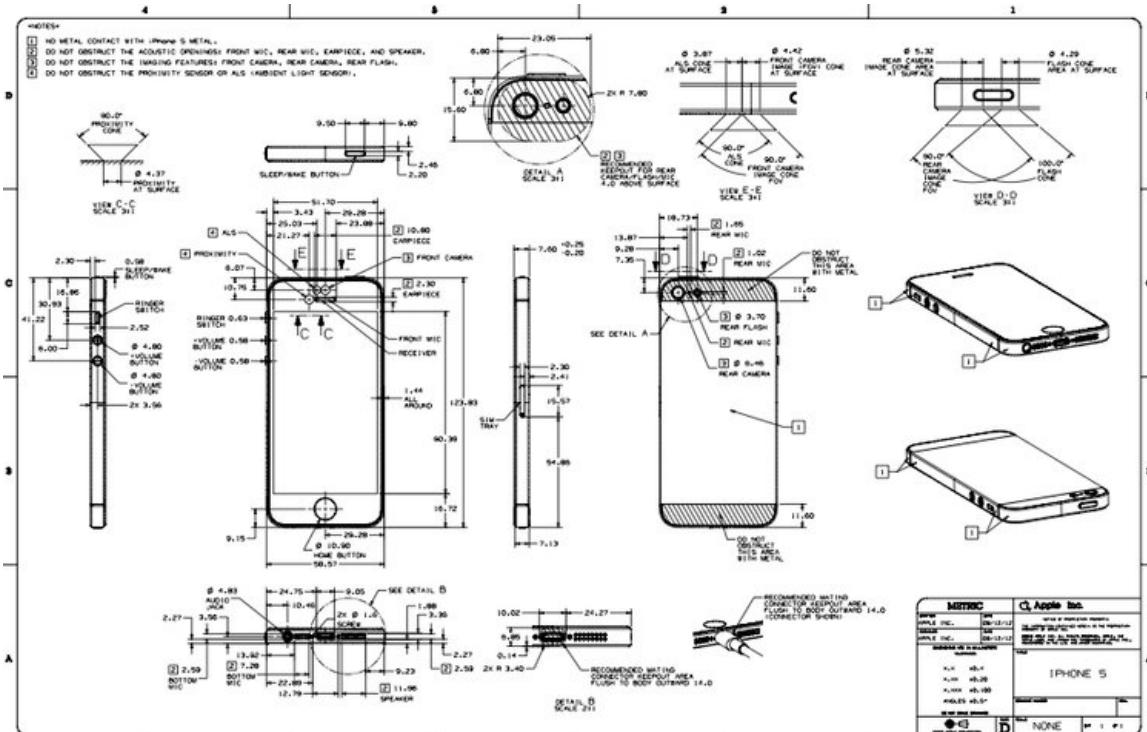
Class and Instance

Definitions

- **Class:**
 - specifies the common behavior of entities.
 - a *blueprint* that defines properties and behavior of an object.
- **Instance:**
 - A particular object or entity of a given class.
 - A concrete, usable object created from the blueprint.

Classes vs Instances

- Class
 - Blueprints, designs



- Instance
 - Actual copies you use



One **blueprint** can produce a lot of copies of **iPhone**

One **class** can produce a lot of copies of **instances**

Example

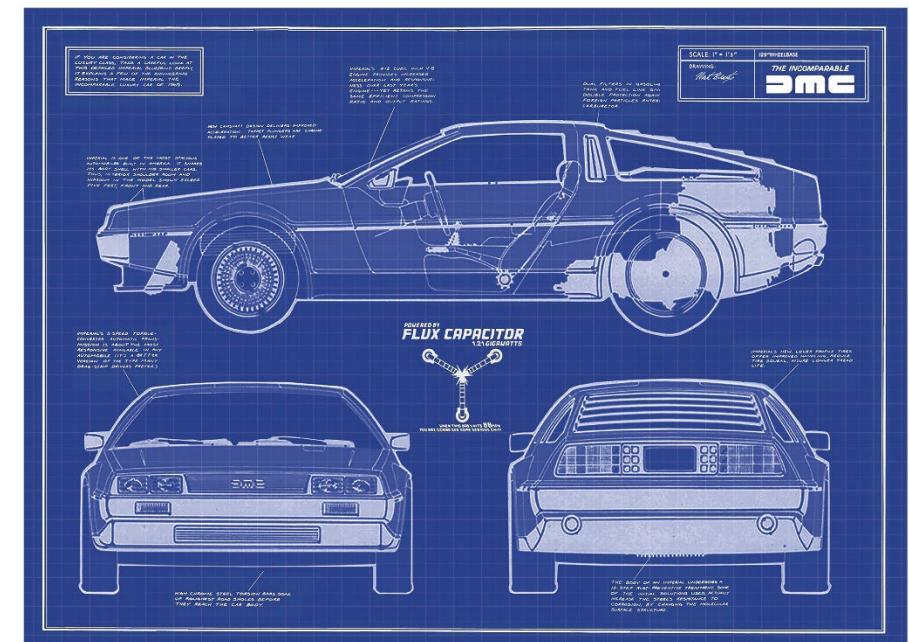
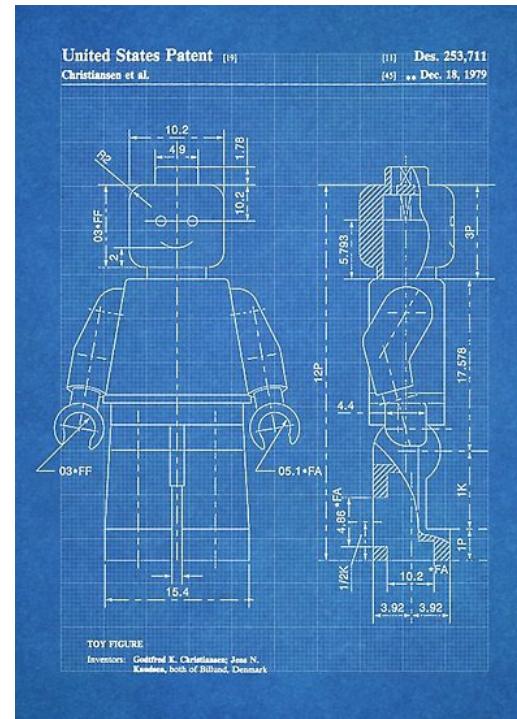
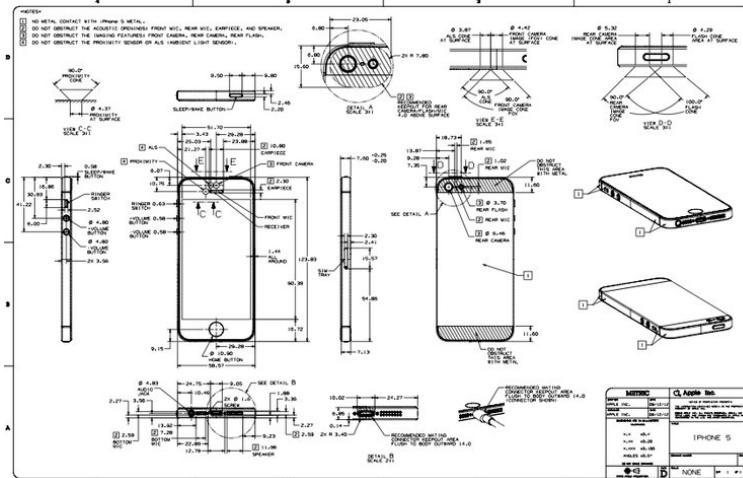
- String is a **class** in Python

```
>>> s = 'abc'  
>>> type(s)  
<class 'str'>
```

- The variable **s** above, is an **instance** of the class **str** (String)
 - And you can create other instances, e.g. **s1**, **str1**, **s2**, etc. of ONE class **str**
 - And *each* instance will store different values

Designing our own class

- Python OOP means we can design our own class and methods!



Back to the Future™ and © is a trademark and copyright of Universal Studios and its joint venture licensee by Universal Studios. All rights reserved.

- Let's try to create a class called "StudentRecord"

Class StudentRecord

- Design
 - In a student record, we want to store
 - Name, student number, gender, year, and marks for English, math, science and social studies
 - Or maybe more?

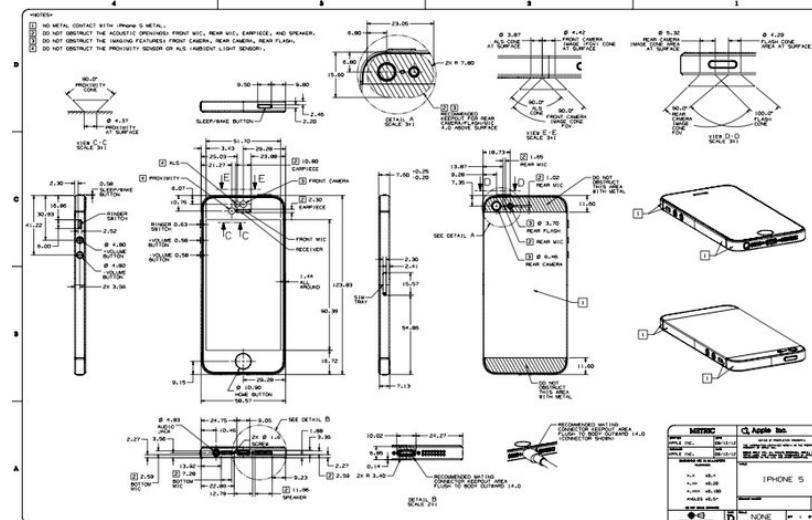
```
class StudentRecord():  
    def __init__(self):  
        self.name = ''  
        self.sn = ''  
        self.gender = ''  
        self.year = 0  
        self.eng = 0  
        self.math = 0  
        self.sci = 0  
        self.ss = 0
```

In OOP, these are called **attributes**

Class StudentRecord

- Design
 - In a student record, we want to store
 - Name, student number, gender, year, and marks for English, math, science and social studies
 - Or maybe more?
- But this is **ONLY** the **class**
 - Namely the blueprint
 - Can we use this phone blueprint to call someone?
 - You have to **MAKE** a phone by it

```
class StudentRecord():
    def __init__(self):
        self.name = ''
        self.sn = ''
        self.gender = ''
        self.year = 0
        self.eng = 0
        self.math = 0
        self.sci = 0
        self.ss = 0
```



Create an Instance

- When you create a new **instance/variable**:
 >>> alan = StudentRecord()
- It's like you create a new variable x for integer
 >>> x = 1
- A new instance/variable is born/created
- Important:

When you create an instance, the constructor function is called

```
class StudentRecord():  
    def __init__(self):  
        self.name = ''  
        self.sn = ''  
        self.gender = ''  
        self.year = 0  
        self.eng = 0  
        self.math = 0  
        self.sci = 0  
        self.ss = 0
```

A “self” variable?

- Every class definition has access to a `self` variable
- `self` is a reference to the entire instance

What is `__init__()`?

- `def __init__(self):`
 - called when the object is first initialized
 - `self` argument is a reference to the object calling the method.
 - It allows the method to reference properties and other methods of the class.
- Are there other special methods?
 - Yes! Special methods have `__` in front and behind the name

Create an Instance

- When you create a new instance/variable:

```
>>> alan = StudentRecord()
```

- When you create an instance, the **constructor** function is called

- What? What constructor?!

- In every class, you have to define a function called “__init__”

- “self” means your own record

- To distinguish from a local variable in a function (as we had learned so far) that will be destroyed after the function ended

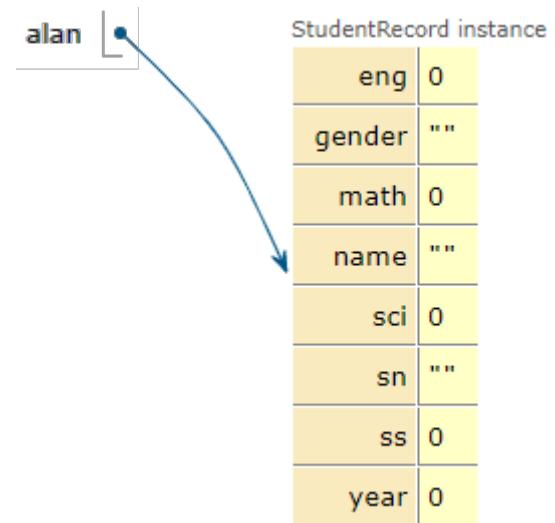
```
class StudentRecord():  
    def __init__(self):  
        self.name = ''  
        self.sn = ''  
        self.gender = ''  
        self.year = 0  
        self.eng = 0  
        self.math = 0  
        self.sci = 0  
        self.ss = 0
```

With two underscores “__”

Create an Instance

- When you create a new instance/variable:
`>>> alan = StudentRecord()`
- When you create an instance, the constructor function is called
- So after the line above, the instance **alan** will contain
 - So, the values in the constructor can be considered the default values for initialization

```
class StudentRecord():  
    def __init__(self):  
        self.name = ''  
        self.sn = ''  
        self.gender = ''  
        self.year = 0  
        self.eng = 0  
        self.math = 0  
        self.sci = 0  
        self.ss = 0
```



You Store Values into an Instance

```
14 alan = StudentRecord()  
15 alan.name = 'Alan'  
16 alan.sn = 'A1000001A'  
17 alan.gender = 'M'  
18 alan.eng = 90  
→ 19 alan.math = 100  
→ 20 alan.sci = 80  
21 alan.ss = 10
```

alan

StudentRecord instance

eng	90
gender	"M"
math	100
name	"Alan"
sci	0
sn	"A1000001A"
ss	0
year	0

Before the red arrow

StudentRecord instance

eng	90
gender	"M"
math	100
name	"Alan"
sci	80
sn	"A1000001A"
ss	10
year	0

Finally. (Oops, I forgot to assign "year")

Create as Many Instances as You Want

```
alan = StudentRecord()  
alan.name = 'Alan'  
alan.sn = 'A1000001A'  
alan.gender = 'M'  
alan.eng = 90  
alan.math = 100  
alan.sci = 80  
alan.ss = 10
```

```
mary = StudentRecord()  
mary.name = 'Mary'  
mary.sn = 'A1000002A'  
mary.gender = 'F'  
mary.eng = 100  
mary.math = 70  
mary.sci = 60  
mary.ss = 100
```



StudentRecord instance

eng	90
gender	"M"
math	100
name	"Alan"
sci	80
sn	"A1000001A"
ss	10
year	0

StudentRecord instance

eng	100
gender	"F"
math	70
name	"Mary"
sci	60
sn	"A1000002A"
ss	100
year	0

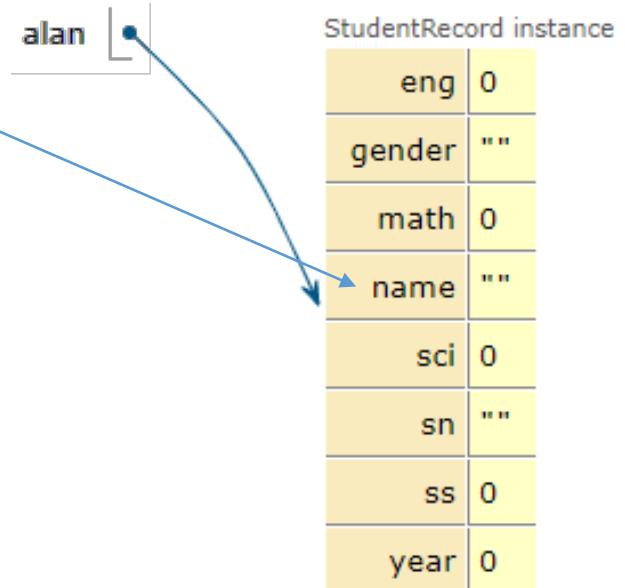
A “self” variable?

- Every class definition has access to a `self` variable
- `self` is a reference to the entire instance
- A self variable will NOT **disappear** when the function exits

With or without “self.”

- The variable “name” will disappear after the completion of the function `__init__()`
- The variable “`self.name`” will remain after the completion of the function `__init__()`

```
class StudentRecord():
    def __init__(self):
        name = 'whatever'
        self.name = ''
        self.sn = ''
        self.gender = ''
        self.year = 0
        self.eng = 0
        self.math = 0
        self.sci = 0
        self.ss = 0
```



Other than Student Records

	Remius Lv 1	Holy Knight HP 558/ 558 MP 98/ 98
	Naia Lv 1	Monk HP 682/ 682 MP 46/ 46
	Elicia Lv 1	Knight HP 645/ 645 MP 95/ 95
	Erik Lv 1	Warrior HP 562/ 562 MP 41/ 41

	13-inch MacBook Air	13-inch MacBook Pro	15-inch MacBook Pro
Base Processor	8th Gen Intel dual-core 1.6GHz i5, turbo boost up to 3.6Ghz	8th Gen Intel quad-core 1.4GHz i5, turbo boost up to 3.9GHz	9th Gen Intel 6-core 2.6GHz i7, turbo boost up to 4.5GHz
Storage	128GB, 256GB, 512GB, or 1TB SSD	128GB, 256GB, 512GB, 1TB, or 2TB SSD	256GB, 512GB, 1TB, 2TB, or 4TB SSD
RAM	8GB or 16GB	8GB or 16GB	16GB or 32GB
Graphics	Integrated Intel UHD 617	Integrated Intel Plus Iris 645	Base w/ Radeon Pro 555X 4GB of GDDR5 memory + Intel UHD Graphics 630
Wi-Fi	802.11ac	802.11ac	802.11ac
Touch ID	✓	✓	✓
Touch Bar	✗	✓	✓
T2 Security Chip	✓	✓	✓

Or Bank Accounts

- What attributes should we store in a bank account record?
 - Name
 - Balance
- Isn't it a bit clumsy to set the values?
 - We know that some attributes must be initialized before used

```
class BankAccount():
    def __init__(self):
        self.name = ''
        self.balance = 0
```

```
myAcc = BankAccount()
myAcc.name = 'Alan'
myAcc.balance = 1000
```

```
johnAcc = BankAccount()
johnAcc.name = 'John Wick'
johnAcc.balance = 1000000000
```

Initialization through Constructors

- We know that some attributes must be initialized before used
- When we create an instance, we initialize through the constructor

```
class BankAccount():
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance

myAcc = BankAccount('Alan', 1000)
johnAcc = BankAccount('John Wick', 100000000)
```

- It is a good way to “**force**” the instance to be initialized

```
>>> myAcc.balance
1000
>>> myAcc.name
'Alan'
```

Modifying Attributes

- Of course, we can change the attributes of any instance

```
>>> myAcc.balance += 999  
>>> myAcc.balance  
1999
```

- However, is it always good to be changed like that?

```
>>> myAcc.balance -= 10000  
>>> myAcc.balance  
-8001
```

- There are always some rules to control how to modify the attributes
 - In real life, how do you withdraw money from your account?

“Rules”

- Can you walk in the bank and get any **amount** even **more** than your balance? Or any other bank transactions?
- Must through some “mechanism”, e.g.
 - Bank tellers, ATM, phone/internet banking, etc.
- And these mechanisms have some rules,
 - E.g. you cannot withdraw more than your balance

Bank Accounts with “Methods”

```
class BankAccount():
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
    def withdraw(self, amount):
        if self.balance < amount:
            print(f"Money not enough! You do not have ${amount}")
            return 0
        else:
            self.balance -= amount
            return amount
    def showBalance(self):
        print(f'Your balance is ${self.balance}')
```

Attributes

Methods

Bank Accounts with “Methods”

```
>>> myAcc = BankAccount('Alan',1000)
>>> myAcc.showBalance()
Your balance is $1000
>>> myAcc.withdraw(123)
123
>>> myAcc.showBalance()
Your balance is $877
>>> myAcc.withdraw(99999)
Money not enough! You do not have $99999
0
```

Is it a *really* a new thing?

- Recall your previous lectures...

```
lst = [1, 2, 3]
```

```
lst.append(4)
```

```
lst →[1, 2, 3, 4]
```

- Conceptually, append is a method defined in the `List` class.
- Just like withdraw is a method defined in the `BankAccount` class

Inheritance



guess who's
inheriting the money

Let's Define a class Sportcar

```
class Sportcar:  
    def __init__(self, pos):  
        self.pos = pos  
        self.velocity = (0, 0)  
    def setVelocity(self, vx, vy):  
        self.velocity = (vx, vy)  
    def move(self):  
        self.pos = (self.pos[0]+self.velocity[0], self.pos[1]+self.velocity[1])  
        print(f"Move to {self.pos}")  
    def turnOnTurbo(self):  
        print("VR0000000M.....")  
        self.velocity = (self.velocity[0]*2, self.velocity[1]*2)  
        print(f"Velocity increased to {self.velocity}")
```



Test Run

```
>>> myCar = Sportscar((0, 0))
>>> myCar.setVelocity(0, 40)
>>> myCar.move()
Move to (0, 40)
>>> myCar.turnOnTurbo()
VR0000000M.....
Velocity increased to (0, 80)
>>> myCar.move()
Move to (0, 120)
```

How about a class Lorry?



```
class Lorry:
    def __init__(self, pos):
        self.pos = pos
        self.velocity = (0, 0)
        self.cargo = []
    def setVelocity(self, vx, vy):
        self.velocity = (vx, vy)
    def move(self):
        self.pos = (self.pos[0]+self.velocity[0], self.pos[1]+self.velocity[1])
        print(f"Move to {self.pos}")
    def load(self, cargo):
        self.cargo.append(cargo)
    def unload(self, cargo):
        if cargo in self.cargo:
            self.cargo.remove(cargo)
            print(f"Cargo {cargo} unloaded.")
        else:
            print(f"Cargo {cargo} not found.")
    def inventory(self):
        print("Inventory: "+str(self.cargo))
```

Test Run

```
>>> myTruck = Lorry((10,10))          >>> myTruck.unload("Food")
>>> myTruck.setVelocity(10,0)           Cargo Food unloaded.
>>> myTruck.move()                   >>> myTruck.inventory()
Move to (20, 10)                         Inventory: ['Supplies']
>>> myTruck.load("Food")              >>> myTruck.unload("Gold")
>>> myTruck.load("Supplies")           Cargo Gold not found.
>>> myTruck.inventory()
Inventory: ['Food', 'Supplies']
```

Compare the Two Classes

Sportscar

- Attributes
 - pos
 - velocity
- Methods
 - `__init__()`
 - `setVelocity()`
 - `move()`
 - `turnOnTurbo()`

Lorry

- Attributes
 - pos
 - velocity
 - cargo
- Methods
 - `__init__()`
 - `setVelocity()`
 - `move()`
 - `load()`
 - `unload()`
 - `inventory()`

What are the **common** attributes/methods?

Compare the Two Classes

Sportscar

- Attributes
 - `pos`
 - `velocity`
- Methods
 - `__init__()`
 - `setVelocity()`
 - `move()`
 - `turnOnTurbo()`

Lorry

- Attributes
 - `pos`
 - `velocity`
 - `cargo`
- Methods
 - `__init__()`
 - `setVelocity()`
 - `move()`
 - `load()`
 - `unload()`
 - `inventory()`

What are the common attributes/methods?

Extract the Common Pattern!

Vehicle

- Attributes
 - pos
 - velocity
- Methods
 - `__init__()`
 - `setVelocity()`
 - `move()`

Sportscar

- Methods
 - `__init__()`
 - `turnOnTurbo()`

Lorry

- Attributes
 - cargo
- Methods
 - `__init__()`
 - `load()`
 - `unload()`
 - `inventory()`

The Classes Vehicle and Sportcar

```
class Vehicle:  
    def __init__(self, pos):  
        self.pos = pos  
        self.velocity = (0, 0)  
    def setVelocity(self, vx, vy):  
        self.velocity = (vx, vy)  
    def move(self):  
        self.pos = (self.pos[0]+self.velocity[0], self.pos[1]+self.velocity[1])  
        print(f"Move to {self.pos}")
```

```
class Sportscar(Vehicle): ← Sportscar inherits EVERYTHING from Vehicle  
    def turnOnTurbo(self):  
        print("VR000000OM.....")  
        self.velocity = (self.velocity[0]*2, self.velocity[1]*2)  
        print(f"Velocity increased to {self.velocity}")
```

How about Lorry?

- In the OLD Lorry code

```
class Lorry:  
    def __init__(self, pos):  
        self.pos = pos  
        self.velocity = (0,0)  
        self.cargo = []  
    def setVelocity(self, vx, vy):  
        self.velocity = (vx, vy)  
    def move(self):  
        self.pos = (self.pos[0]+self.velocity[0], self.pos[1]+self.velocity[1])  
        print(f"Move to {self.pos}")  
    def load(self, cargo):  
        self.cargo.append(cargo)
```

Extra to the `__init__()` in Vehicle

If We Inherit Lorry from Vehicle

- Two ways to implement the constructor
- Method 1: Overriding
 - Simple redefining the method will override the one in Vehicle
- Method 2: Calling super class
 - Redefine a constructor, but call the constructor in `super ()` (`Vehicle` class) instead

or

```
class Lorry(Vehicle):  
    def __init__(self, pos):  
        self.pos = pos  
        self.velocity = (0, 0)  
        self.cargo = []
```

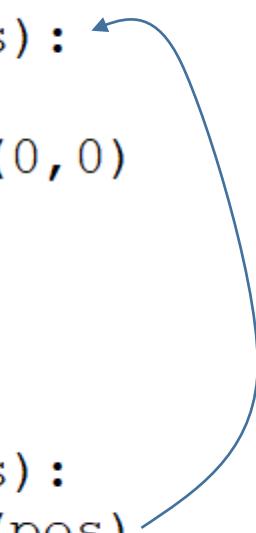
```
class Lorry(Vehicle):  
    def __init__(self, pos):  
        super().__init__(pos)  
        self.cargo = []
```

Super()

- A way to access a method in your parent/higher classes

```
class Vehicle:  
    def __init__(self, pos):  
        self.pos = pos  
        self.velocity = (0, 0)
```

```
class Lorry(Vehicle):  
    def __init__(self, pos):  
        super().__init__(pos)  
        self.cargo = []
```



Which one is better?

- Usually we prefer Method 2 because
 - No duplication of code
- Method 2: Calling super class
 - Redefine a constructor, but call the constructor in super() (Vehicle class) instead

```
class Lorry(Vehicle):  
    def __init__(self, pos):  
        self.pos = pos  
        self.velocity = (0, 0)  
        self.cargo = []
```

```
class Lorry(Vehicle):  
    def __init__(self, pos):  
        super().__init__(pos)  
        self.cargo = []
```



Class Lorry

- Inherit all what class Vehicle has
- In addition, add more functionalities like load() and unload()

```
class Lorry(Vehicle):  
    def __init__(self, pos):  
        super().__init__(pos)  
        self.cargo = []  
    def load(self, cargo):  
        self.cargo.append(cargo)  
    def unload(self, cargo):  
        if cargo in self.cargo:  
            self.cargo.remove(cargo)  
            print(f"Cargo {cargo} unloaded.")  
        else:  
            print(f"Cargo {cargo} not found.")  
    def inventory(self):  
        print("Inventory:" + str(self.cargo))
```

Overall Picture

Subclass
or
Child class

Super class
or
Parent class

Vehicle

- Attributes
 - pos
 - velocity
- Methods
 - `__init__()`
 - `setVelocity()`
 - `move()`

Sportscar

- Methods
 - `__init__()`
 - `turnOnTurbo()`

Lorry

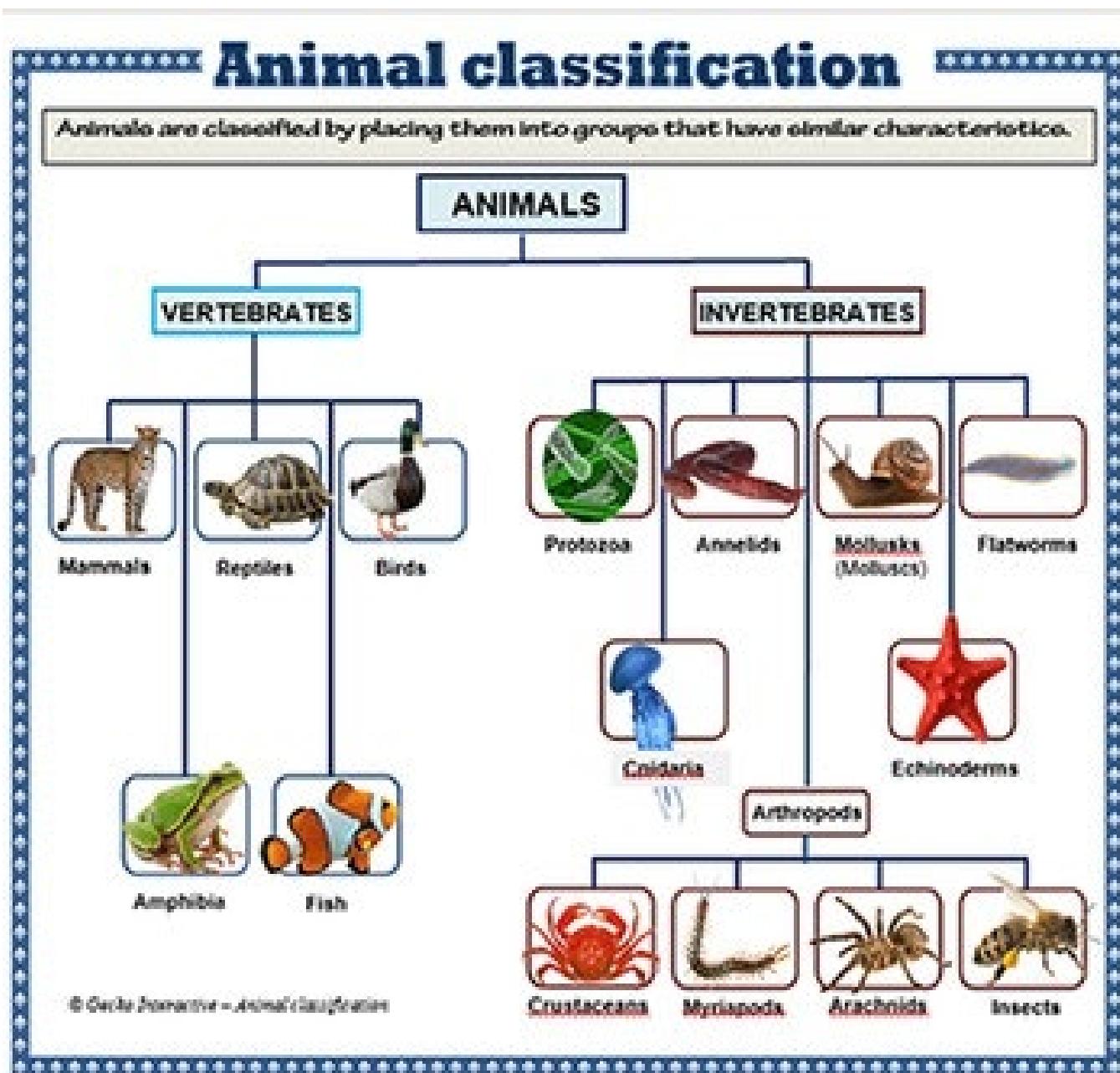
- Attributes
 - cargo
- Methods
 - `__init__()`
 - `load()`
 - `unload()`
 - `inventory()`

Inheritance

- Objects that exhibit similar functionality should “inherit” from the same base object, called the **superclass**.
- An object that inherits from another is called the **subclass**.

Subclass and Superclass

- Usually a subclass is a more specific type of its parent class
 - Like our classification of animals, any “children” in the tree is a more “specific” type of the “parents”
 - (Unless we talk about multiple inheritance later)



Let's define a new class Bisarca

- Car carrier trailer
 - It is a type of truck that carries other cars
- The truck and also load and unload, but only for cars
 - not any type of cargos



Where should
we put class
Bisarca?

Vehicle

- Attributes: **pos, velocity**
- Methods: **__init__(), setVelocity(), move()**

Sportscar

- Methods: **__init__(), turnOnTurbo()**

Lorry

- Attributes: cargo
- Methods: **__init__(), load(), unload(), inventory()**

Bisarca

- Methods: **load()**

Class Bisarca

```
class Bisarca(Lorry):
    def load(self,cargo):
        if isinstance(cargo,Vehicle):
            super().load(cargo)
        else:
            print(f'Your cargo ({cargo}) is not a vehicle!')
```

- The function `isinstance(obj, cal)` check if an instance `obj` is a class or subclass of a certain class `cal`.

Sample Run

```
>>> myDadTruck = Bisarca((0,0))
>>> myDadTruck.load("Food")
Your cargo (Food) is not a vehicle!
>>> myDadTruck.load(myCar)
>>> myDadTruck.load(myTruck)
>>> myDadTruck.inventory()
Inventory:[<__main__.Sportcar object at 0x10d3ecd50>,
<__main__.Lorry object at 0x10d39dc10>]
```

Method Overriding

- When you redefine a same method that was in your parent class
- Your own class will call your new redefined method
 - Instead of your parent's one
- This is called **overriding**

Lorry

- Attributes: cargo
- Methods: `__init__()`, `load()`, `unload()`, `inventory()`

Bisarca

- Methods: `load()`

Multiple Inheritance



Let's Create a Class Cannon



Class Cannon

- Sample run:

```
>>> myCannon = Cannon()
>>> myCannon.fire()
No more ammo
>>> myCannon.reload()
Cannon reloaded
>>> myCannon.reload()
Unable to reload
>>> myCannon.fire()
Fire!!!!!!
>>> myCannon.fire()
No more ammo
```

```
class Cannon:
    def __init__(self):
        self.numAmmo = 0
    def fire(self):
        if self.numAmmo:
            print("Fire!!!!!")
            self.numAmmo -= 1
        else:
            print("No more ammo")
    def reload(self):
        if self.numAmmo:
            print("Unable to reload")
        else:
            print("Cannon reloaded")
            self.numAmmo += 1
```

What Do You Have When You...

- Merge a cannon and a vehicle?



+



=



We Want to Have BOTH!

```
>>> myTank = Tank((0,0))
>>> myTank.setVelocity(40,10)
>>> myTank.move()
Move to (40, 10)
>>> myTank.move()
Move to (80, 20)
>>> myTank.reload()
Cannon reloaded
>>> myTank.fire()
Fire!!!!!!
```

Where should we put the class Tank?

Vehicle

- Attributes: pos, velocity
- Methods: setVelocity(), move()

Cannon

- Attributes: numAmmo
- Methods: fire()

Sportscar

- Methods: __init__(), turnOnTurbo()

Lorry

- Attributes: cargo
- Methods: __init__(), load(), unload(), inventory()

Tank

- ## Bisarca
- Methods: load()

A Bit Trouble

- Which constructor `__init__()` should the Tank call?

```
class Cannon:  
    def __init__(self):  
        self.numAmmo = 0
```

```
class Vehicle:  
    def __init__(self, pos):  
        self.pos = pos  
        self.velocity = (0, 0)
```

- Seems like we need BOTH

```
class Tank(Vehicle, Cannon):  
    def __init__(self, pos):  
        Vehicle.__init__(self, pos)  
        Cannon.__init__(self)
```

Call BOTH!!!

Resolving Methods

So far we have

Vehicle

- Attributes: pos, velocity
- Methods: setVelocity(), move()

Cannon

- Attributes: numAmmo
- Methods: fire()

Sportscar

- Methods: __init__(), turnOnTurbo()

Lorry

- Attributes: cargo
- Methods: __init__(), load(), unload(), inventory()

Tank

- ## Bisarca
- Methods: load()

What Do You Have When You...

- Merge a Bisarca and a Cannon?



+



=



Let's Construct Class BattleBisarca

```
class BattleBisarca(Bisarca,Cannon):  
    def __init__(self,pos):  
        Bisarca.__init__(self,pos)  
        Cannon.__init__(self)  
  
OptimasPrime = BattleBisarca((0,0))  
OptimasPrime.load("Food")
```

- Wait... Which load() is called?



Vehicle

- Attributes: pos, velocity
- Methods: setVelocity(), move()

Cannon

- Attributes: numAmmo
- Methods: fire()

Lorry

- Attributes: cargo
- Methods: __init__(), load()
, unload(), inventory()

Bisarca

- Methods: load()

Which
load() is
called by
BattleBisarca
?

BattleBisarca

Vehicle

- Attributes: pos, velocity
- Methods: setVelocity(), move()

Cannon

- Attributes: numAmmo
- Methods: fire()

Lorry

- Attributes: cargo
- Methods: __init__(), load()
, unload(), inventory()

Bisarca

- Methods: **load()**

The nearest one will be called

BattleBisarca

Let's Construct Class BattleBisarca

```
class BattleBisarca(Bisarca,Cannon):  
    def __init__(self,pos):  
        Bisarca.__init__(self,pos)  
        Cannon.__init__(self)
```

```
OptimasPrime = BattleBisarca((0,0))  
OptimasPrime.load("Food")
```

- Wait... Which load() is called?

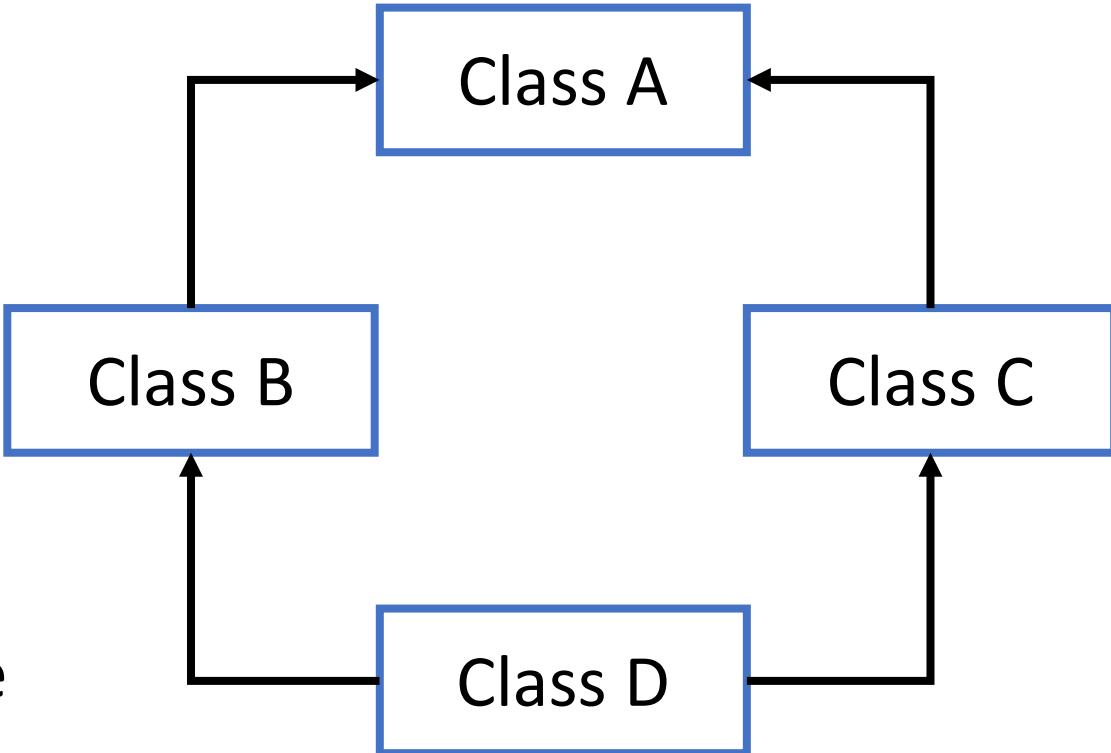
```
>>> OptimasPrime = BattleBisarca((0,0))  
>>> OptimasPrime.load("Food")
```

Your cargo (Food) is not a vehicle!



Multiple Inheritance

- Complication arises when the same method is available in two distinct superclasses
- And how about **Diamond Inheritance**
- But if you are really interested in these
 - Check out:
 - <http://python-history.blogspot.com/2010/06/method-resolution-order.html>



Multiple Inheritance

- Not many OOP Language support MI
 - E.g. no MI in C++, Java
- MI causes more trouble sometime because you may call the unexpected method in a complicated inheritance structure
- Recommendation is, only use MI if the parents are very different
 - E.g. Vehicle and Cannon
 - Or Tablet (computer) + calling device = smart phone

Private vs Public

Private vs Public

- So far, all our methods in a class are all *public*
- Meaning they can be called with the instance
- E.g. for the class BankAccount
 - Even we set up the method withdraw() to “prevent” illegal access

```
def withdraw(self, amount):  
    if self.balance < amount:  
        print(f"Money not enough! You do not have ${amount}")  
        return 0  
    else:  
        self.balance -= amount  
        return amount
```

- But we can still do

```
>>> myAcc.showBalance()  
Your balance is $1000  
>>> myAcc.balance -= 9999  
>>> myAcc.showBalance()  
Your balance is -$8999
```

Another Example: Remember the Bisarca?

```
>>> myDadTruck.load(myCar)
>>> myDadTruck.load(myTruck)
>>> myDadTruck.inventory()
```

So ugly

```
Inventory: [<__main__.Sportcar object at
0x10d3ecd50>, <__main__.Lorry object at
0x10d39dc10>]
```

- What I really want is

```
>>> myDadTruck.inventory()
Inventory: ['Sportscar', 'Lorry']
```

So I change my Bisarca class into

```
class Bisarca(Lorry):
    def convertCargo(self):
        output = []
        for c in self.cargo:
            output.append(str(type(c)).split('.')[1].split('\\')[0])
        return output
    def inventory(self):
        print("Inventory:"+str(self.convertCargo()))
```

- Wait, but I actually do not want anyone to use the method convertCargo(), it's not for anyone
 - I want to make it **private**

Private Methods

- If you add two **underscore** before the method name

```
class Bisarca(Lorry):  
    def __convertCargo(self):  
        output = []  
        for c in self.cargo:  
            output.append(str(type(c)).split('.')[1].split('\\')[0])  
        return output  
  
    def inventory(self):  
        print("Inventory:"+str(self.__convertCargo()))
```

- That function can be used inside your class but cannot be called outside!

```
>>> myDadTruck.__convertCargo()  
Traceback (most recent call last):  
  File "<pyshell#1>", line 1, in <module>  
    myDadTruck.__convertCargo()  
AttributeError: 'Bisarca' object has no attribute '__convertCargo'
```

Private but not Private

- However, it's not very true...
- You can add ‘_’ and the class name to access it

```
>>> myDadTruck._Bisarca__convertCargo()  
['Sportscar', 'Lorry']
```

- But why do we have this?!

“Private” Methods

- Originally, in a lot of other OOP languages (e.g. C++, Java), a private method/variable will NOT be accessible by anyone other than the class itself.
- The purpose is to prevent any programmers to access the method/variable in a wrong way
 - E.g. directly change the balance of a bank account like

```
myAcc.balance = 100000000
```
- However, Python does not have that “full protection”

Don't forget Archipelagos

Conclusion

Benefits of OOP

- Pros
 - Simplification of complex, possibly hierarchical structures
 - Easy reuse of code
 - Easy code modifiability
 - Intuitive methods
 - Hiding of details through message passing and polymorphism
- Cons
 - Overhead associated with the creation of classes, methods and instances

Major Programming Paradigms

- Imperative Programming
 - C, Pascal, Algol, Basic, Fortran
- Functional Programming
 - Scheme, ML, Haskell,
- Logic Programming
 - Prolog, CLP
- Object-oriented programming
 - Java, C++, Smalltalk

Python??

Image Processing

Installing Python Packages

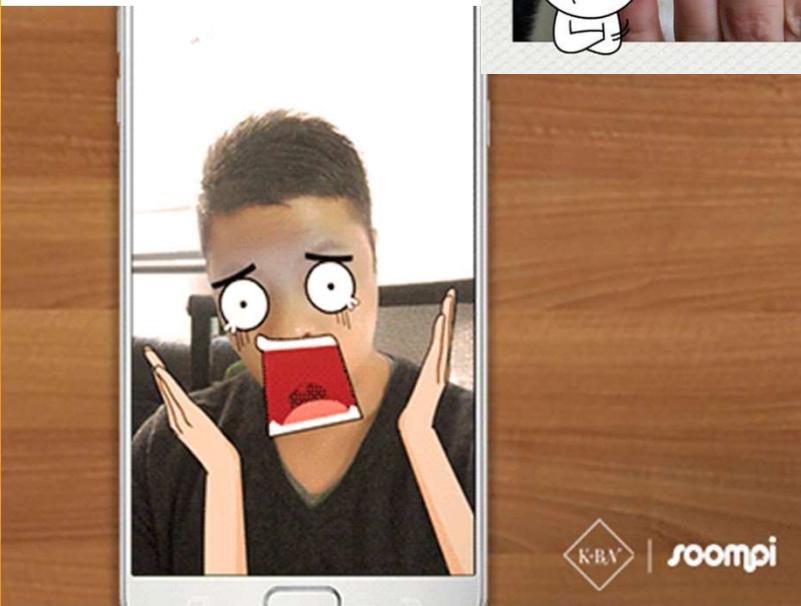
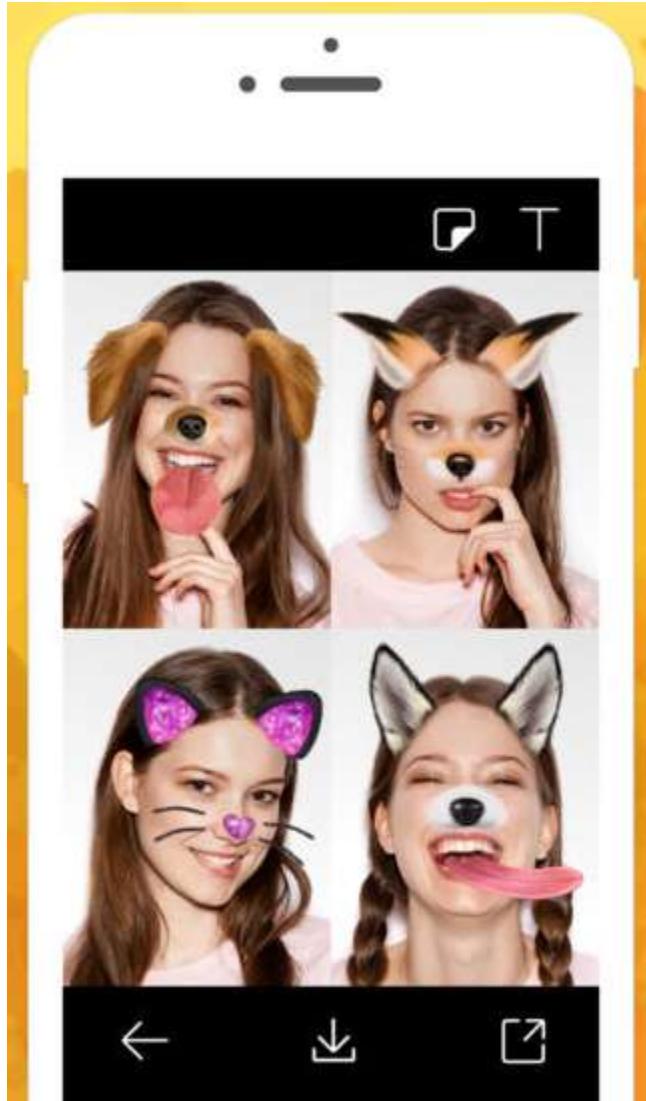
- Python comes with built-in functions
- However, you need to manually install additional packages
 - In Assignment 0, the instructions asked you to install, imageio, numpy, etc
- In this lecture we will need “imageio”
 - To install “imageio” (or any other packages), go to cmd.exe
 - Type “pip install imageio”

- Provided you have your internet connected
- **pip** will download the package and install it for you

```
Microsoft Windows [Version 10.0.17763.737]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\dcschl>pip install scipy
Collecting scipy
  Downloading https://files.pythonhosted.org/packages/e1/9e/454b2dab5ee21f66ebf02ddbc63c5f02
/scipy-1.3.1-cp37-cp37m-win32.whl (27.1MB)
    100% |██████████| 27.1MB 1.7MB/s
Requirement already satisfied: numpy>=1.13.3 in c:\users\dcschl\appdata\local\programs\pytho
ges (from scipy) (1.17.0)
Installing collected packages: scipy
Successfully installed scipy-1.3.1
```

We have all these photo apps



<https://www.everydayfamily.com/slideshow/10-hilariously-awful-photoshop-fails/>



M...il.com>

to me ▾

Hey, just wondering if you could edit this photo of me and my boyfriend. I was hoping you could make his corn dog whole again... with no bites taken out... thanks!



James Fridman <fjamie013@gmail.com>

Done.



Image Processing

- To load an image, you can use the package “imageio”

```
import imageio  
import matplotlib.pyplot as plt
```

```
cat_pic = imageio.imread('cute_cat.jpg')
```

```
plt.imshow(cat_pic)  
plt.show()  
  
>>> type(cat_pic)  
<class 'imageio.core.util.Array'>  
>>> cat_pic.shape  
(600, 600, 3)  
>>>
```

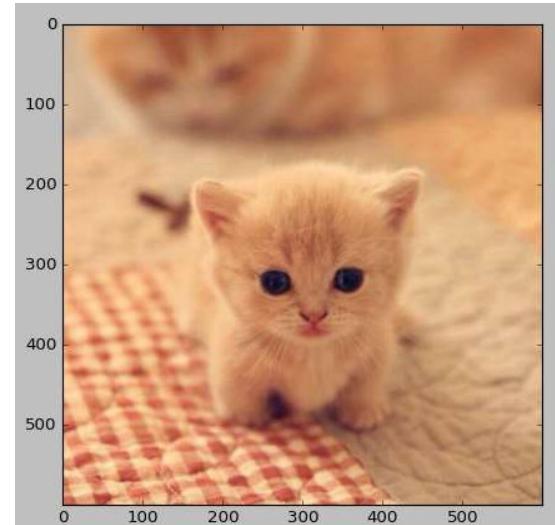


Image Processing

- 600 x 600 pixel
 - And each pixel has three values of R, G and B
 - [R, G, B]

```
>>> type(cat_pic)
<class 'imageio.core.util.Array'>
>>> cat_pic.shape
(600, 600, 3)
>>>
```

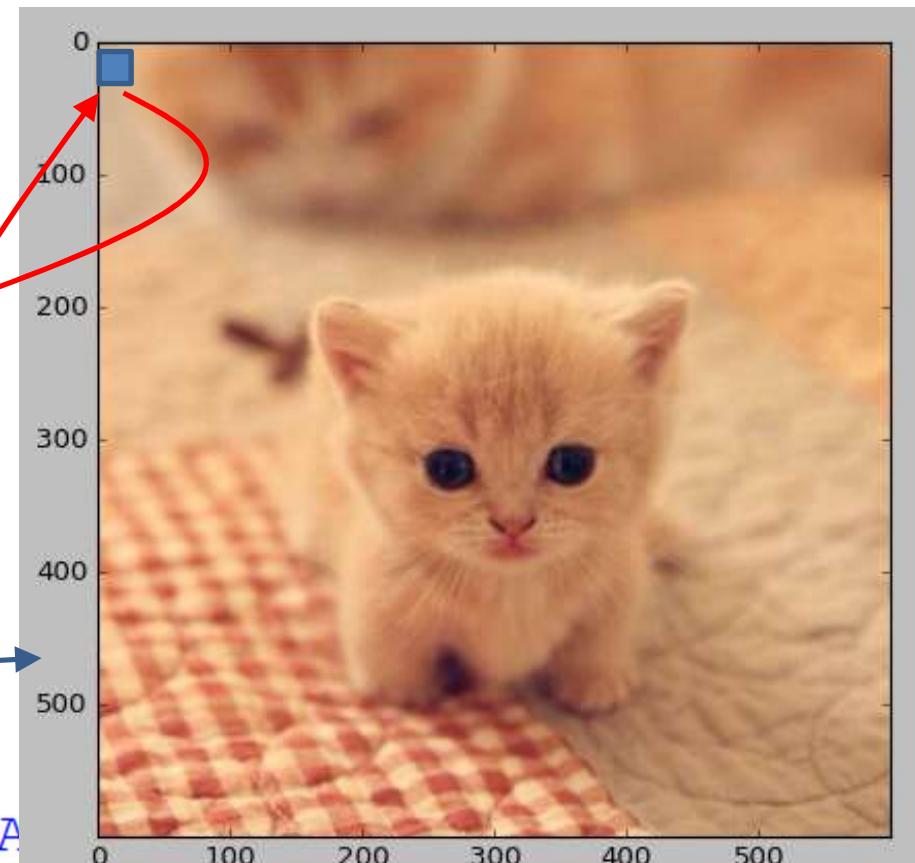
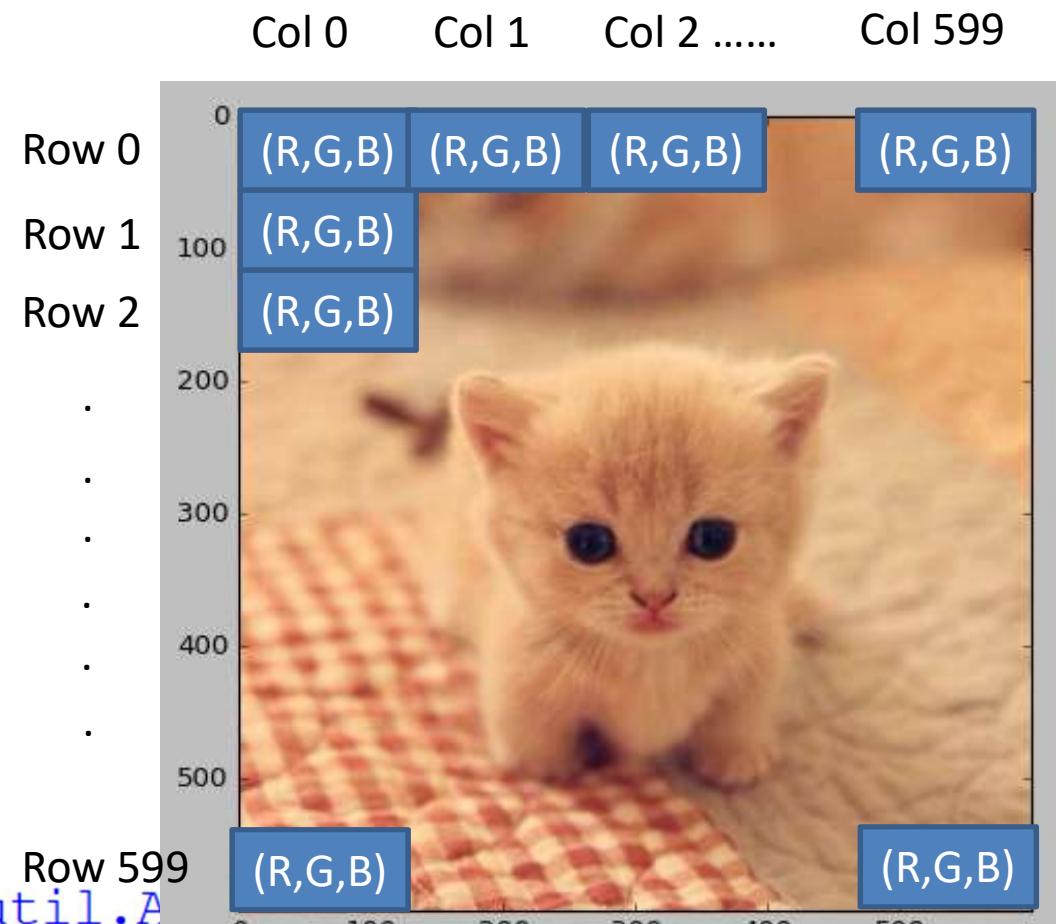


Image Processing

- 600 x 600 pixel
 - [R, G, B]
 - $0 \leq R, G, B \leq 255$

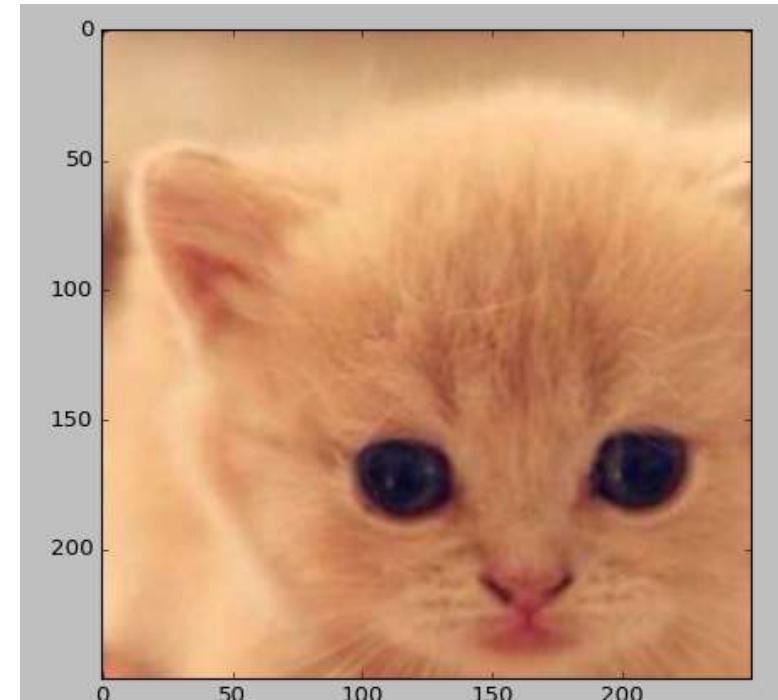


```
>>> type(cat_pic)
<class 'imageio.core.util.Array'>
>>> cat_pic.shape
(600, 600, 3)
>>>
```

Image Processing

- Remember sub-matrix, string slicing, etc.?

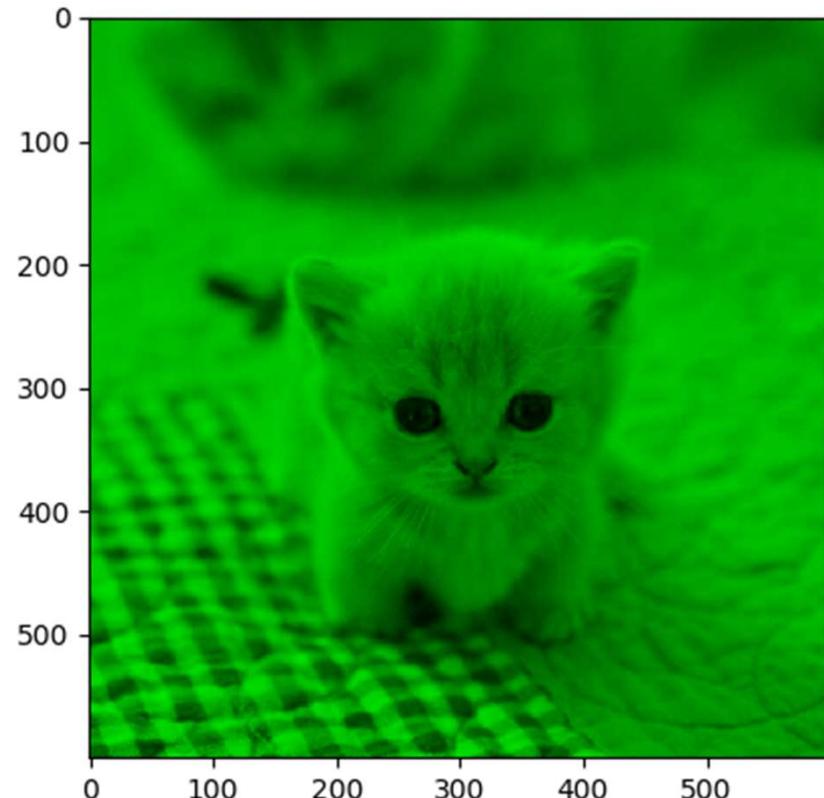
```
cat_pic2 = cat_pic[150:400,150:400]  
plt.imshow(cat_pic2)  
plt.show()
```



Broadcasting

```
cat_pic2 = cat_pic * [0, 1, 0]
plt.imshow(cat_pic2)
plt.show()
```

- every pixel multiply by
 - $[R,G,B] \times [0,1,0] =$
 - $[R \times 0, G \times 1, B \times 0]$
 - $[0, G, 0]$



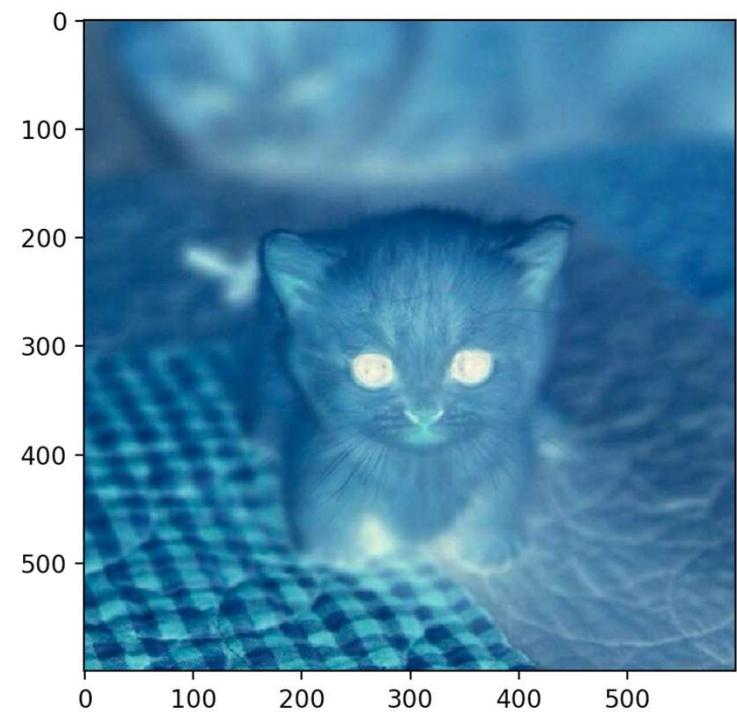
Array Broadcasting

```
>>> a = np.array([1,2,3,4,5])  
>>> a + 1 ← "Broadcasting"  
array([2, 3, 4, 5, 6])  
>>> a * 3 ← Different from LIST  
array([ 3,  6,  9, 12, 15])  
>>> a > 5  
array([False, False, False, False, False], dtype=bool)
```

Create another array with the Boolean results

Negative Image

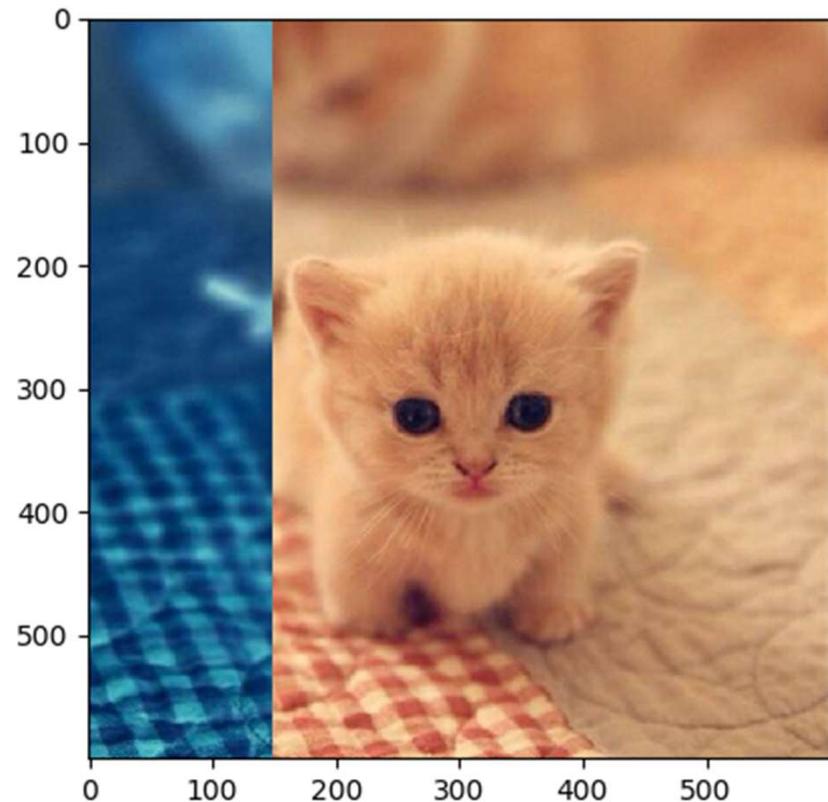
```
cat_pic2 = 255 - cat_pic  
plt.imshow(cat_pic2)  
plt.show()
```



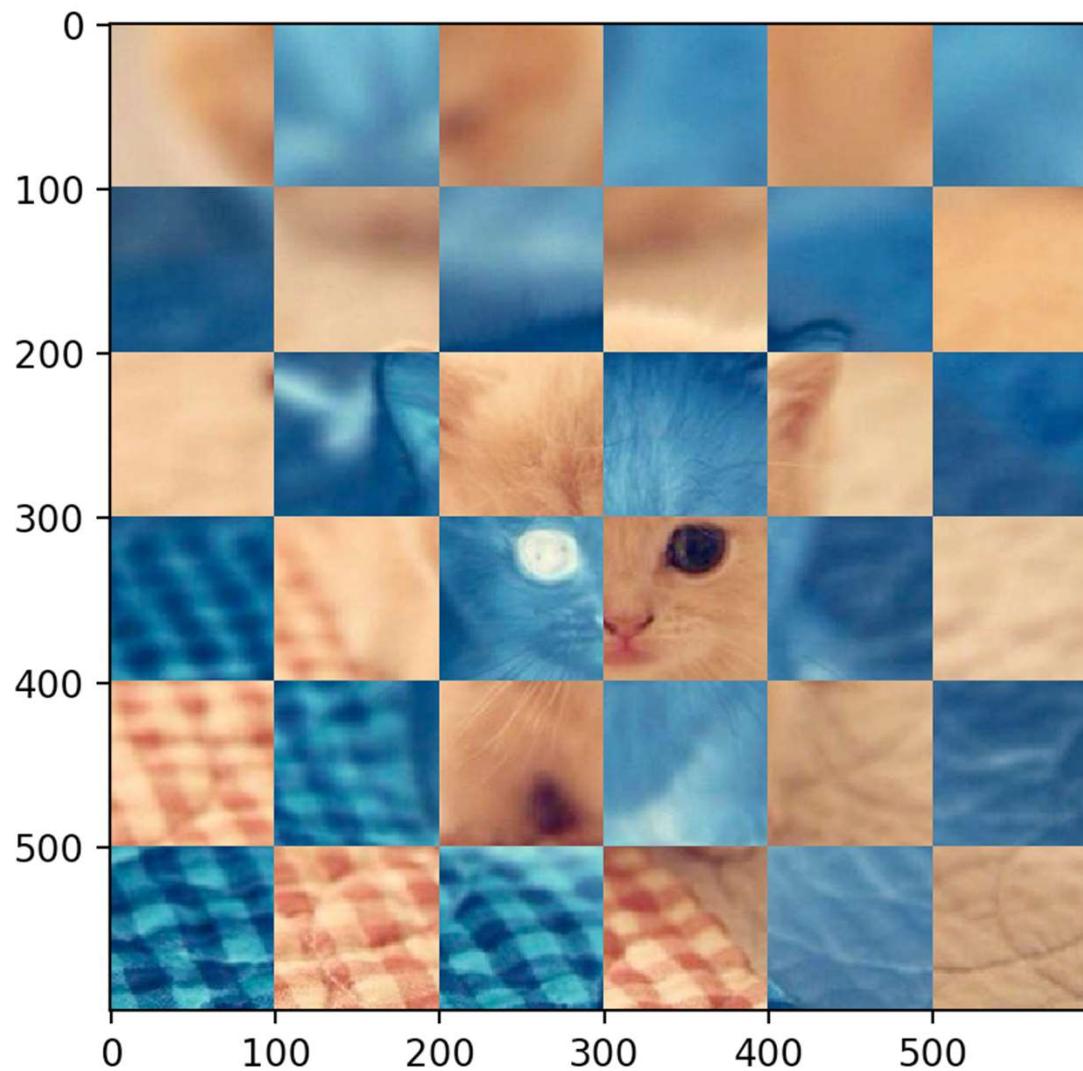
```
for i in range(cat_pic.shape[0]):  
    for j in range(cat_pic.shape[1]):  
        if j < cat_pic.shape[1]/4:  
            cat_pic[i][j] = 255 - cat_pic[i][j]
```

2D Array looping

```
plt.imshow(cat_pic)  
plt.show()
```



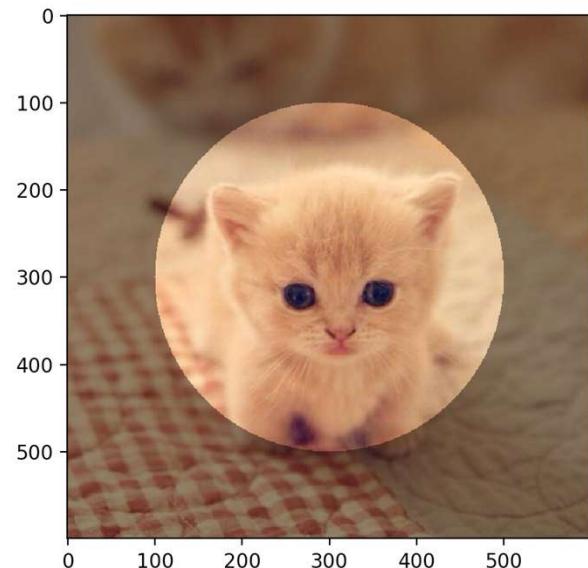
How to....?

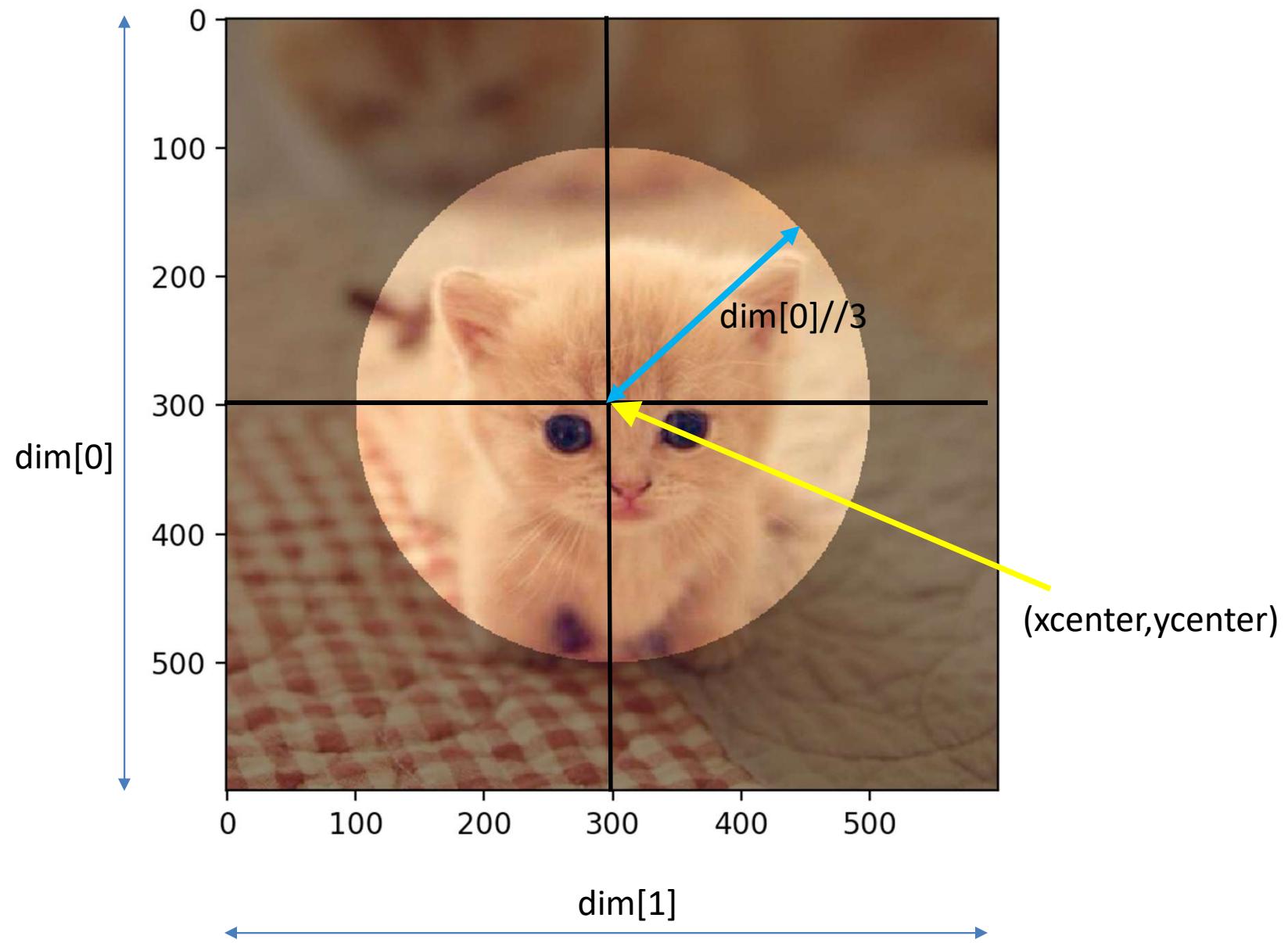


Making a Mask

```
dim = cat_pic.shape
xcenter = dim[1]//2
ycenter = dim[0]//2

for i in range(cat_pic.shape[0]):
    for j in range(cat_pic.shape[1]):
        if (i-xcenter)**2 + (j-ycenter)**2 > (dim[0]//3)**2:
            cat_pic[i][j] = cat_pic[i][j]*0.3
```



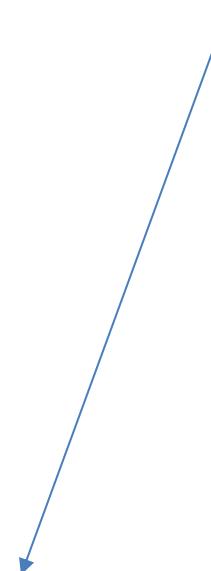


Making a Mask

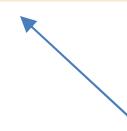
```
dim = cat_pic.shape
xcenter = dim[1]//2
ycenter = dim[0]//2

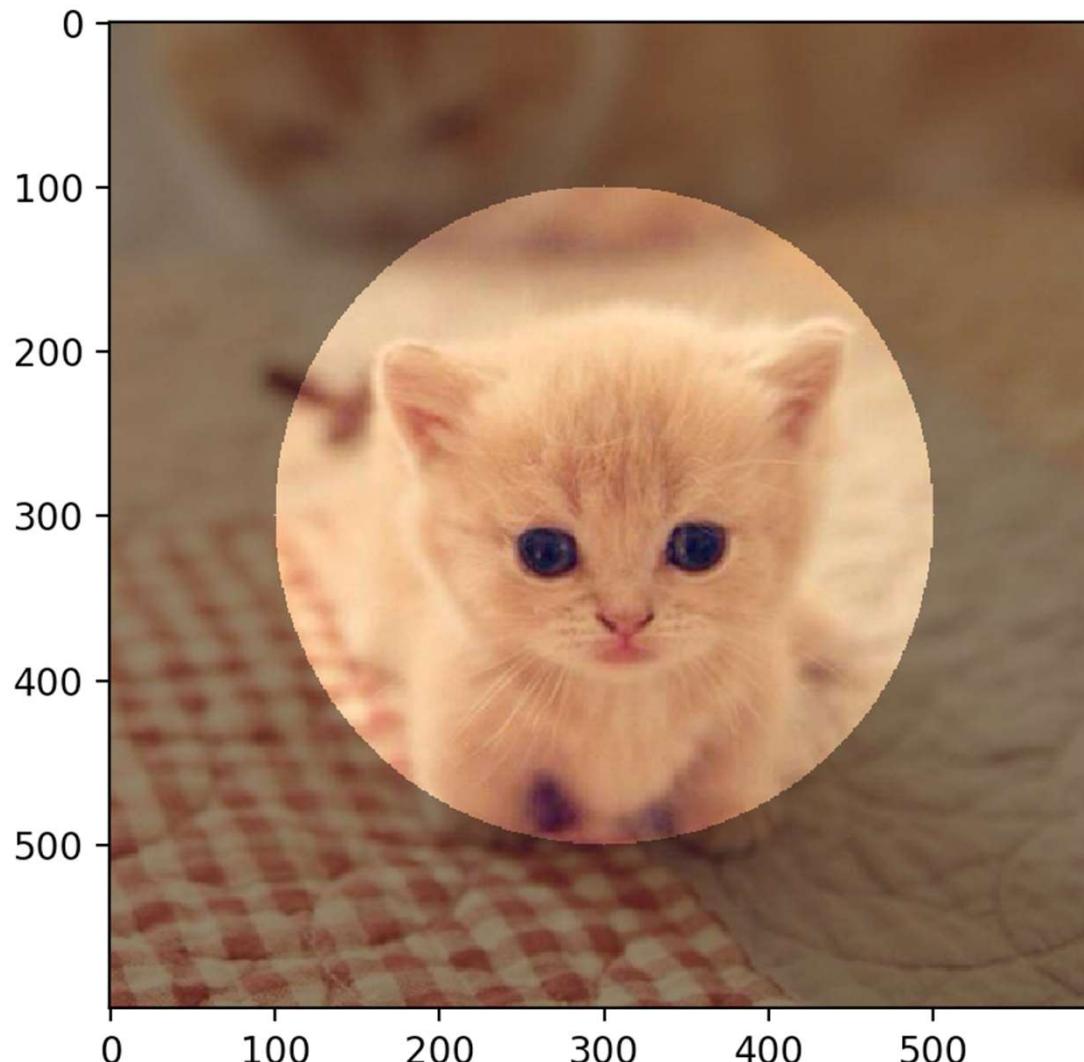
for i in range(cat_pic.shape[0]):
    for j in range(cat_pic.shape[1]):
        if (i-xcenter)**2 + (j-ycenter)**2 > (dim[0]//3)**2:
            cat_pic[i][j] = cat_pic[i][j]*0.3
```

If the pixel is out
of the circle



Each color of the
pixel is divided by 2



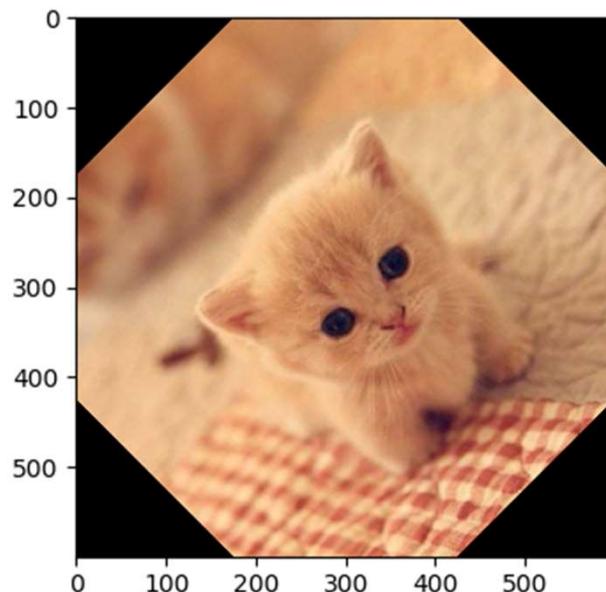
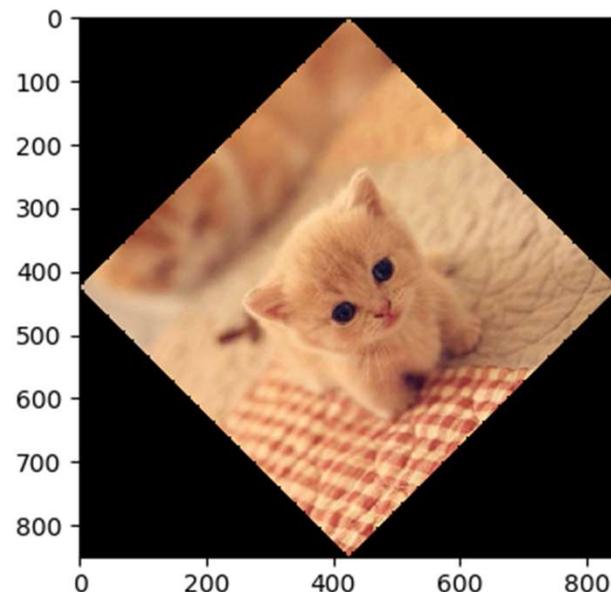


Your
picture
array

- Any time you want to save an image:
`imageio.imwrite('file name.png', cat_pic)`

Rotating an Image

```
from scipy import ndimage
rcat1 = ndimage.rotate(cat_pic, 45)
rcat2 = ndimage.rotate(cat_pic, 45, reshape=False)
plt.subplot(121)
plt.imshow(rcat1)
plt.subplot(122)
plt.imshow(rcat2)
plt.show()
```



Applying Filters

```
from scipy import misc,ndimage  
import matplotlib.pyplot as plt  
import numpy as np
```

```
cat_pic = misc.imread('cute_cat.jpg')  
blurred_cat_pic = ndimage.gaussian_filter(  
    cat_pic, sigma=(9,9,1))
```

```
plt.imshow(blurred_cat_pic)  
plt.show()
```

Blending on x

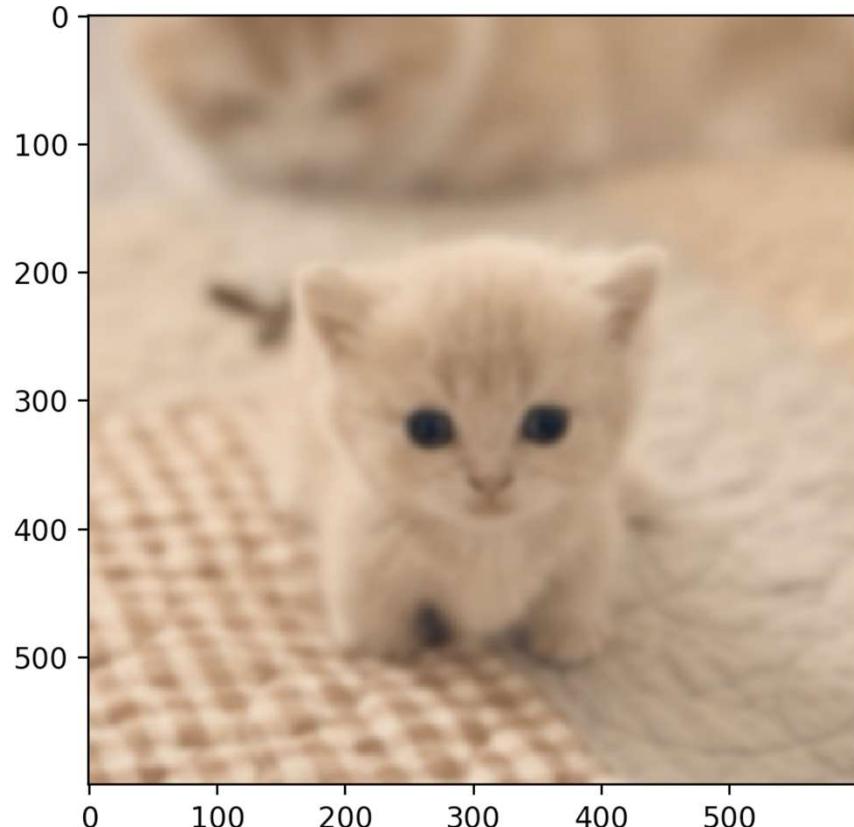
Blending on y

NO Blending on
colors

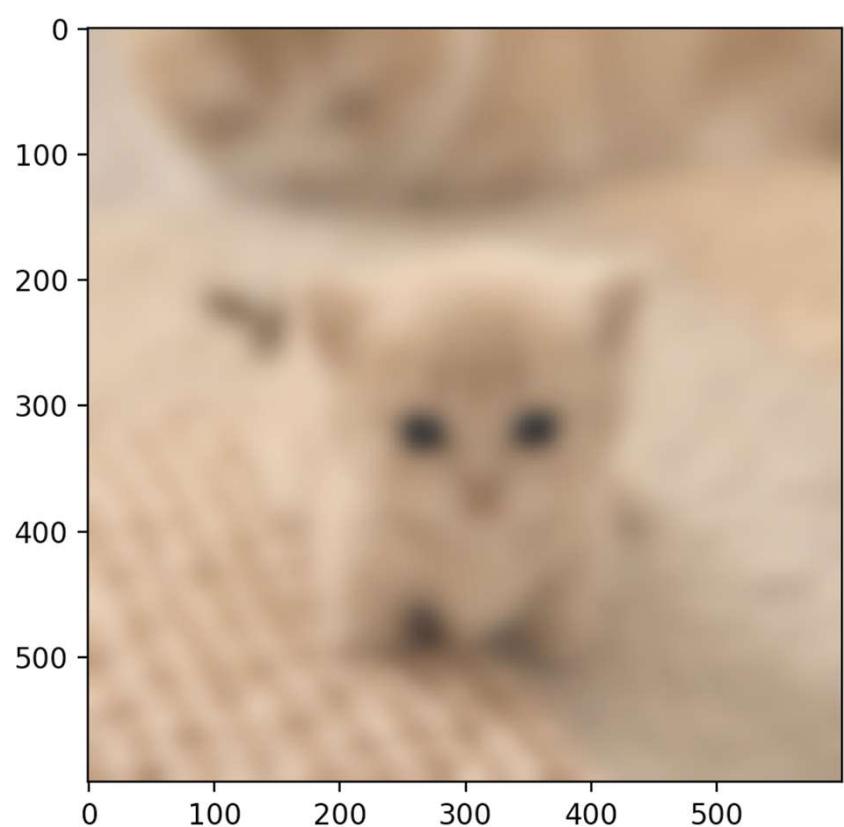
Applying Filters

```
blurred_cat = ndimage.gaussian_filter(cat_pic,sigma=(9,9,1))
```

- $\text{sigma} = (3,3,1)$



- $\text{sigma} = (9,9,1)$



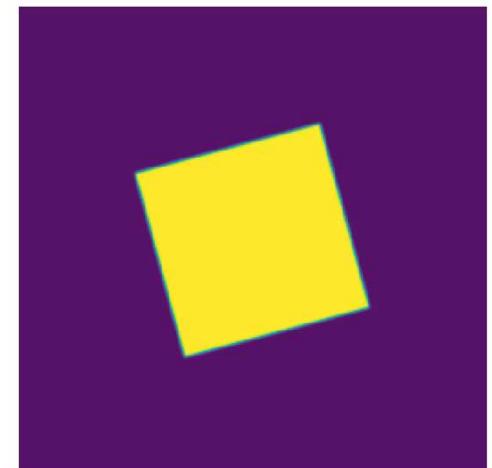
Edge Detection

- Anyhow generate an image

```
img = np.zeros((256, 256))
img[64:-64, 64:-64] = 1
```

```
img = ndimage.rotate(img, 15, mode='constant')
img = ndimage.gaussian_filter(img, 1)
```

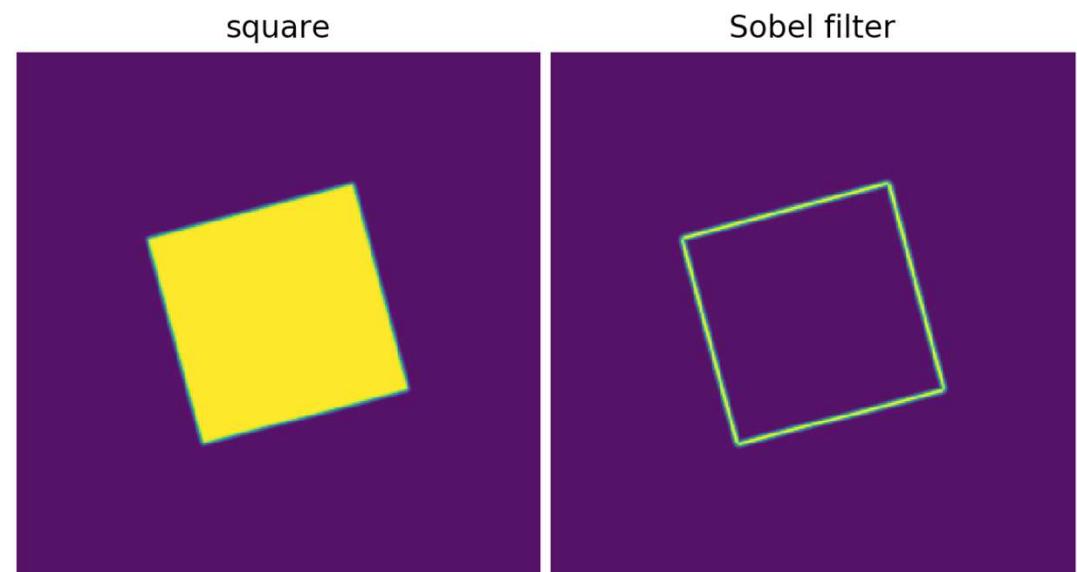
square



```
# Applying Sobel filter to the image
sx = ndimage.sobel(img, axis=0, mode='constant')
sy = ndimage.sobel(img, axis=1, mode='constant')
sob = np.hypot(sx, sy)
```

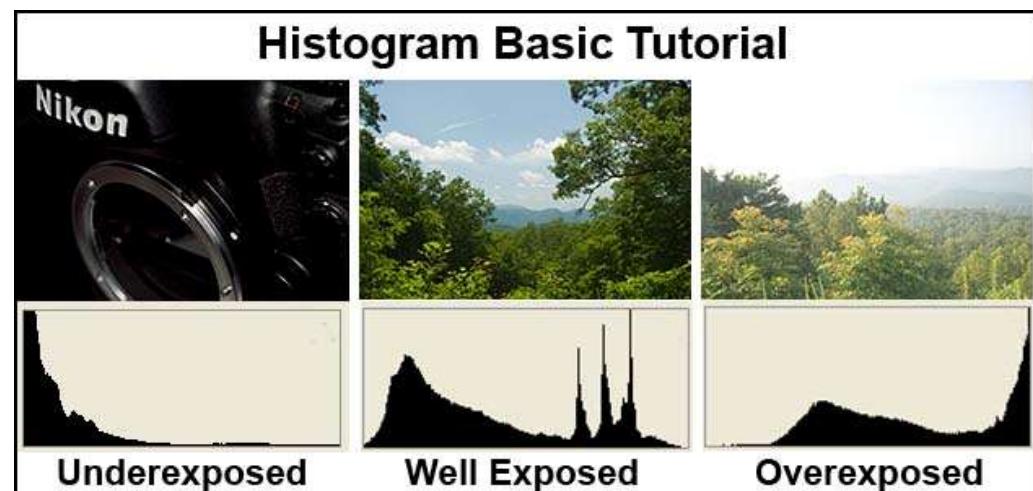
```
plt.subplot(121)
plt.imshow(img)
plt.axis('off')
plt.title('square')
```

```
plt.subplot(122)
plt.imshow(sob)
plt.axis('off')
plt.title('Sobel filter')
```



More in Numpy and Scipy

- Fourier Transform
- Uniform filter
- Histogram
- Laplace... etc



- More on
 - <https://docs.scipy.org/doc/scipy/reference/ndimage.html>

PILLOW

A Fork in PIL

PILLOW is a fork of PIL

- PIL stands for Python Imaging Library

```
from PIL import Image  
  
pic = Image.open('my flight delay.JPG')  
pic.show()  
|
```

Let's get the secret out

STD	FLIGHT	VIA/TO	GATE	STATUS
14:05	DZ6237		43	A
15:25	HU7663	Nanjing	15	Last Call
15:25	I9923	C Nantong	60	Delayed
15:30	ZH9809	Wuxi	59	Last Call
15:40	HU7263	Hohhot	16	Delayed
15:45		C Huangshan	39	Gate
15:55	HU7255	Zhengzhou	17	W
15:55	HU7326	Taiyuan	46	Delayed
15:55	I076	Wuxi	78	Delayed
16:00	314	Beijing	14	Delayed
16:00	672	Tianjin	26	Delayed
16:05	010		32	Delayed
16:05			61A	Delayed
16:20	I454		71	Delayed
16:25	053	Beijing	62B	Delayed
16:30	ZH9877	Hangzhou	54	Delayed
16:30	350	CZ971	Sh: 44	Delayed
16:35	3U8784	Chongqing	48	A
16:35	KN5851	Nanyuan	62A	Cancelled
16:35	CA4846	Yantai	50A	Delayed
16:40	I509	Nanchang	61B	Delayed
16:50	CZ8250		Ch: 29	Delayed
STD	FLIGHT	VIA/TO	GATE	STATUS
16:55	3U	Chengdu	53	Delayed
16:55	I915	MF1411	21	Delayed
16:55	I9915	KY! Qingdao	58	Delayed
17:00	ZH1		27	Delayed
17:00	I48	ZH434 Chongqing	61A	Delayed
17:05	CZ6473	Bijie	41	Delayed
17:05	I44	MF1561 IN	50B	Delayed
17:10	I10	MF196 Shenyang	30	Delayed
17:15	ZF	Hefei	32	Delayed
17:20	I14	ZH431 Chengdu	35	Delayed
17:20	MU575E		36	Delayed
17:25	HU7708	Beijing	20	Delayed
17:25	HU7715	Jinan	25	Delayed
17:25	BK2860	Tianjin	54	Delayed
17:30	CZ6913	Nanjing	34	Cancelled
17:30	FM9334	Iao	51	Delayed
17:30	CZ8649	Xining	28	Delayed
17:35	I63	MF135 Hangzhou	38	Cancelled
17:35	ZF	Nanjing	77	Cancelled
17:40		Chongqing	62A	Delayed
17:40	I91	MF10E Beijing	63	Delayed
17:40	HU7067	Xuzhou	44	Delayed
STD	FLIGHT	VIA/TO	GATE	STATUS
17:45		C Hangzhou	59	Weather Co
17:45	TV9832	Igdu/Linzhi	24	Weather Co
17:45	Z9334	Ibo/Qingdao	41	Weather Co
17:50	A3272	Ighai Hongqiao	42	Delayed
17:50	9C8776	Ighai Hongqiao	49	Weather Co
17:55	HU7763	Nanjing	25	Weather Co
17:55	Z9215	Nanjing	46	Weather Co
18:00		C Beijing	14	Weather Co
18:00		Ighai Pudong	76	Weather Co
18:05		Hangzhou	61A	Weather Co
18:05		Izhou/Shenyang	52	Weather Co
18:10	I21	Chongqing	18	Weather Co
18:10	I7	CA370 Chengdu	78	Weather Co
18:15		Igzhou/Hohhot	34	Weather Co
18:15	ZH9171	Hot/Haila'er	72	Weather Co
18:15		Ihen/Shanghai Hi	17	Weather Co
18:20	CZ6328	Dalian	33	Weather Co
18:20	I912	MU315 Ning/Nanyuan	43	Weather Co
18:25	ZH9111	Yichun/Beijing	77	
18:30		Guiyang	22	Weather Co
18:30	MU5354	Ighai Hongqiao		Cancelled
18:35	C2	Chengdu	63	Weather Co

尊敬的旅客请注意:受本场天气影响, 部

尊敬的旅客请注意:受本场天气影响, 部

心等候, 不便之处敬请谅解!

Page: 2/3

Page: 3/3

```
from PIL import Image
from PIL.ExifTags import TAGS, GPSTAGS

pic = Image.open('my flight delay.JPG')

def get_exif_data(image):
    exif_data = {}
    info = image._getexif()
    if info:
        for tag, value in info.items():
            decoded = TAGS.get(tag, tag)
            if decoded == "GPSInfo":
                gps_data = {}
                for t in value:
                    sub_decoded = GPSTAGS.get(t, t)
                    gps_data[sub_decoded] = value[t]
                exif_data[decoded] = gps_data
            else:
                exif_data[decoded] = value

    return exif_data

print(get_exif_data(pic) ['GPSInfo'])
pic.show()
```

PILLOW

- Cannot escape!

```
{'GPSLatitudeRef': 'N', 'GPSLatitude': ((22, 1),  
(38, 1), (1484, 100)), 'GPSLongitudeRef': 'E', 'G  
PSLongitude': ((113, 1), (48, 1), (2726, 100)), '  
GPSAltitudeRef': b'\x00', 'GPSAltitude': (2761, 2  
25), 'GPSTimeStamp': ((10, 1), (34, 1), (1420, 10  
0)), 'GPSSpeedRef': 'K', 'GPSSpeed': (0, 1), 'GPS  
ImgDirectionRef': 'T', 'GPSImgDirection': (11511,  
542), 'GPSDestBearingRef': 'T', 'GPSDestBearing':  
(11511, 542), 'GPSDateStamp': '2017:07:17', 'GPSH  
PositioningError': (1414, 1)}
```

PILLOW

```
from PIL import Image
from PIL import ImageFilter
pic = Image.open('cute cat.jpg')

pic.show()

blurred_pic = pic.filter(ImageFilter.BLUR)
blurred_pic.show()

sharpen_pic = pic.filter(ImageFilter.SHARPEN)
sharpen_pic.show()
```



Original



Blurred



Sharpen

Copy And Paste

```
from PIL import Image
```

```
pic = Image.open('cute_cat.JPG')
```

```
part = pic.crop((200, 200, 400, 400))
```

```
pic.paste(part, (0, 400))
```

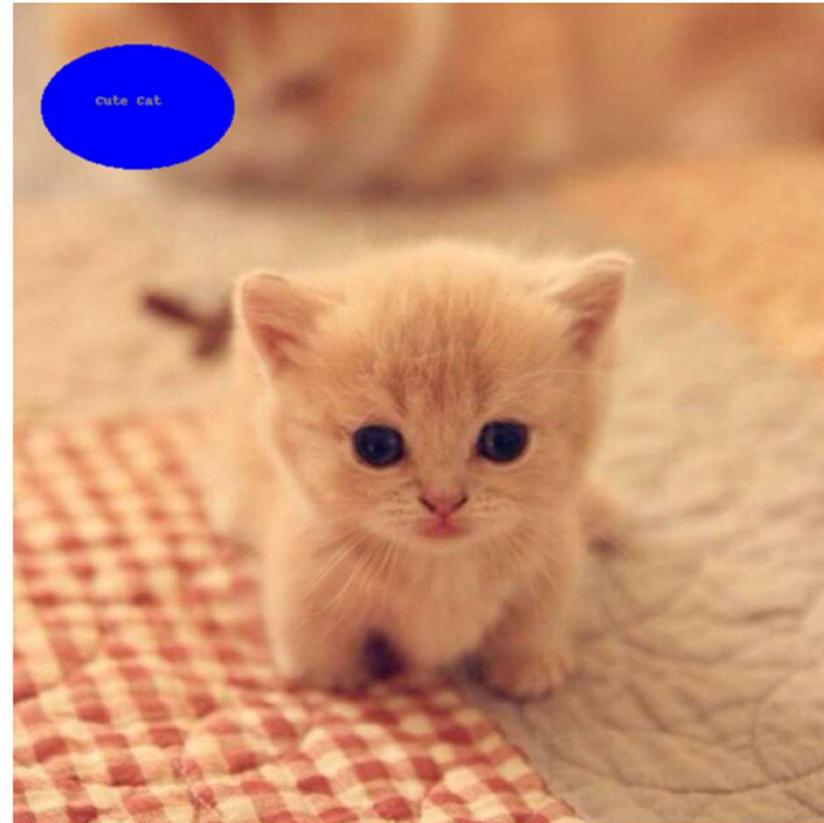
```
pic.show()
```

Paste it on
the position
(0,400)

Copy (crop)
the part of
the picture



```
from PIL import Image, ImageDraw, ImageFont  
  
pic = Image.open('cute cat.JPG')  
  
draw = ImageDraw.Draw(pic)  
draw.ellipse((20, 30, 160, 120), fill='blue')  
draw.text((60, 65), 'Cute Cat', fill = 'gray')  
  
pic.show()
```



Other operations

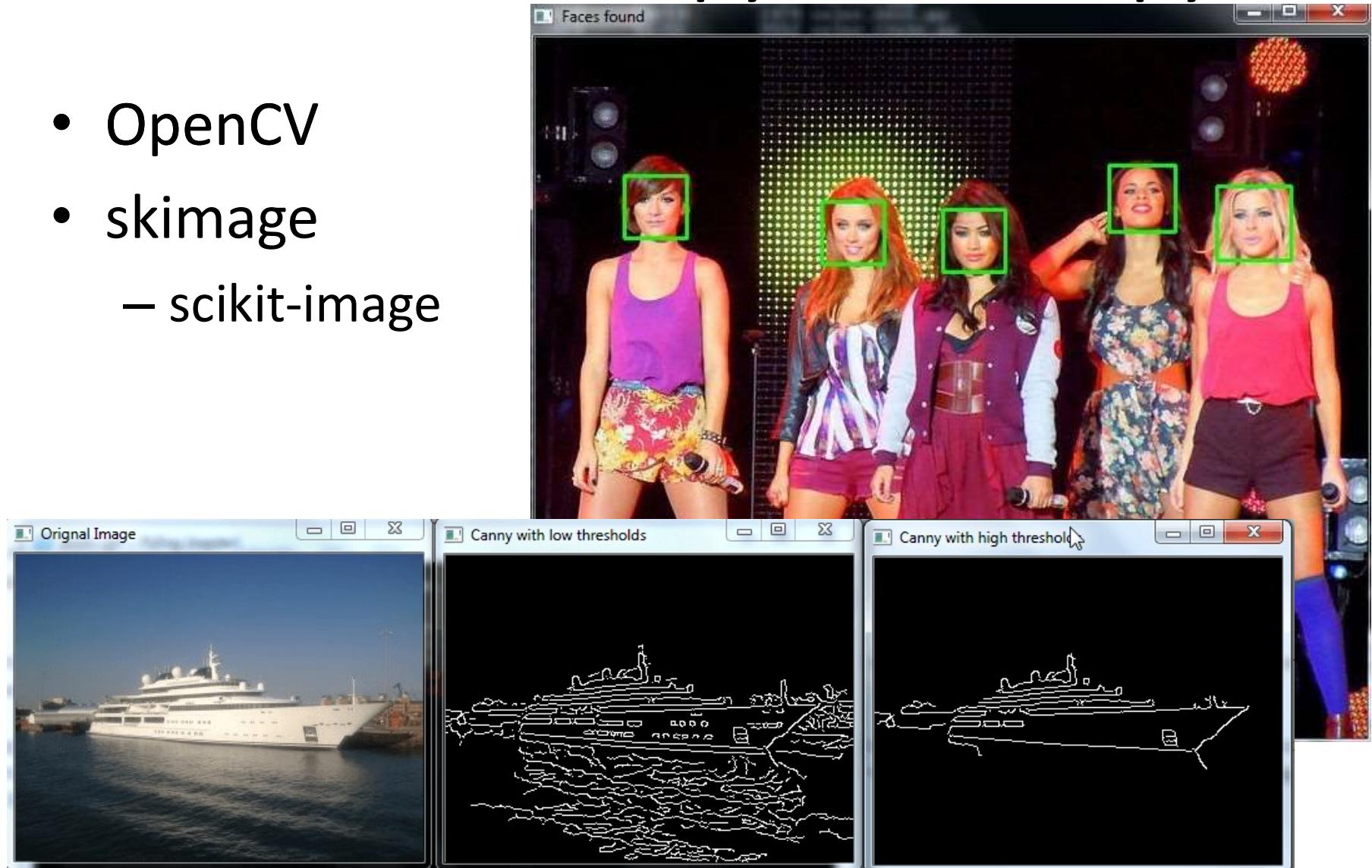
- resize
- rotation/flipping
- transpose
- Drawing shapes
- etc. etc..

<https://pillow.readthedocs.io/en/stable/>

- `Image` Module
- `ImageChops` ("Channel Operations") Module
- `ImageColor` Module
- `ImageCms` Module
- `ImageDraw` Module
- `ImageEnhance` Module
- `ImageFile` Module
- `ImageFilter` Module
- `ImageFont` Module
- `ImageGrab` Module (macOS and Windows only)
- `ImageMath` Module
- `ImageMorph` Module
- `ImageOps` Module
- `ImagePalette` Module
- `ImagePath` Module
- `ImageQt` Module
- `ImageSequence` Module
- `ImageStat` Module
- `ImageTk` Module
- `ImageWin` Module (Windows-only)
- `ExifTags` Module
- `TiffTags` Module
- `PSDraw` Module
- `PixelAccess` Class
- `PyAccess` Module

Other Than Scipy and Numpy

- OpenCV
- skimage
 - scikit-image



Searching

Searching

- You have a list.
- How do you find something in the list?
- Basic idea: go through the list from start to finish one element at a time.

Linear Search

- Idea: go through the list from start to finish

5	2	3	4
---	---	---	---

- Example: Search for 3

5	2	3	4
---	---	---	---

3 not found, move on

5	2	3	4
---	---	---	---

3 not found, move on

5	2	3	4
---	---	---	---

Found 3.

Linear Search

Idea: go through the list from start to finish

```
# equivalent code
for i in [5, 2, 3, 4]:
    if i == 3:
        return True
```

Linear Search

Implemented as a function:

```
def linear_search(value, lst):
    for i in lst:
        if i == value:
            return True
    return False
```

What kind of performance can we expect?

Large vs small lists?

Sorted vs unsorted lists?

$O(n)$

Can we do better?

Of course Ia!

Searching

IDEA:

If the elements in the list were sorted in order, life would be much easier.

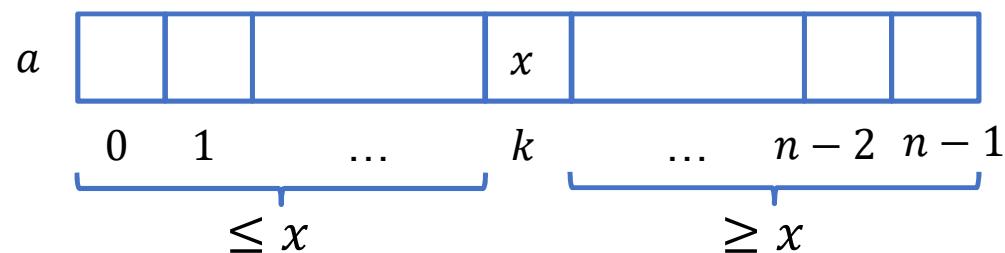
Why?

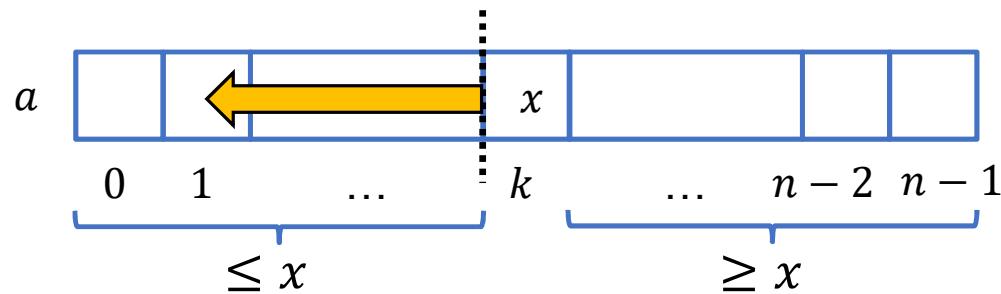


IDEA

If list is sorted, we can
“divide-and-conquer”

Assuming a list is sorted in ascending order:





if the k^{th} element is larger than what we are looking for, then we only need to search in the indices $< k$

Binary Search

1. Find the middle element.
2. If it is what we are looking for (key), return `True`.
3. If our key is smaller than the middle element, repeat search on the left of the list.
4. Else, repeat search on the right of the list.

Binary Search

Looking for 25 (key)

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

Find the middle element: 34

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

Not the thing we're looking for: $34 \neq 25$

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

$25 < 34$, so we repeat our search on the left half:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

Binary Search

Find the middle element: 12

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

$25 > 12$, so we repeat the search on the right half:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

Find the middle element: 25

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

Great success: 25 is what we want

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

Binary Search

“Divide and Conquer”

In large sorted lists, performs much better
than linear search on average.

Binary Search

Algorithm (assume sorted list):

1. Find the middle element.
2. If it is we are looking for (key), return True.
3. A) If our key is **smaller** than the middle element, repeat search on the left of the element.
B) Else, repeat search on the right of the element.

Binary Search

```
def binary_search(key, seq):
    if seq == []:
        return False
    mid = len(seq) // 2
    if key == seq[mid]:
        return True
    elif key < seq[mid]:
        return binary_search(key, seq[:mid])
    else:
        return binary_search(key, seq[mid+1:])
```

Binary Search

```
def binary_search(key, seq): # seq is sorted
    def helper(low, high):
        if low > high:
            return False
        mid = (low + high) // 2 # get middle
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

```
def binary_search(key, seq):
    def helper(low, high):
        if low > high:
            return False
        mid = (low + high) // 2
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 1. Find the middle element.

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

```
def binary_search(key, seq):                                key → 11
    def helper(low, high):
        if low > high:
            return False
        mid = (low + high) // 2
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)                            helper(0, 10-1)
```

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

```
def binary_search(key, seq):
    def helper(low, high): # 0, 9
        if low > high:
            return False
        mid = (low + high) // 2 # mid=4
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 1. Find the middle element.

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

```
def binary_search(key, seq):
    def helper(low, high): # 0, 9
        if low > high:
            return False
        mid = (low + high) // 2 # mid=4
        if key == seq[mid]: # 11 == 25
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 2. If it is what we are looking for, return True

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

```
def binary_search(key, seq):
    def helper(low, high): # 0, 9
        if low > high:
            return False
        mid = (low + high) // 2 # mid=4
        if key == seq[mid]: # 11 == 25
            return True
        elif key < seq[mid]: # 11 < 25
            return helper(low, mid-1) # helper(0, 4-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 3a. If key is smaller, look at left side

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

```
def binary_search(key, seq):
    def helper(low, high): # 0, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=4
        if key == seq[mid]: # 11 == 25
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 3a. If key is smaller, look at left side

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

```
def binary_search(key, seq):
    def helper(low, high): # 0, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=1
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 1. Find the middle element

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

```
def binary_search(key, seq):
    def helper(low, high): # 0, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=1
        if key == seq[mid]: # 11 == 9
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 2. If it is what we are looking for, return True

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

```
def binary_search(key, seq):
    def helper(low, high): # 0, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=1
        if key == seq[mid]: # 11 == 9
            return True
        elif key < seq[mid]: # 11 < 9
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 3a. If key is smaller, look at left side

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

```
def binary_search(key, seq):
    def helper(low, high): # 0, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=1
        if key == seq[mid]: # 11 == 9
            return True
        elif key < seq[mid]: # 11 < 9
            return helper(low, mid-1)
        else:
            return helper(mid+1, high) # helper(1+1, 3)
    return helper(0, len(seq)-1)
```

Step 3b. Else
look at right
side

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

```
def binary_search(key, seq):
    def helper(low, high): # 2, 3
        if low > high:
            return False
        mid = (low + high) // 2
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 3b. Else
look at right
side

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

```
def binary_search(key, seq):
    def helper(low, high): # 2, 3
        if low > high:
            return False
        mid = (low + high) // 2
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 3b. Else
look at right
side

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

```
def binary_search(key, seq):
    def helper(low, high): # 2, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=2
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 1. Find
the middle
element

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

```
def binary_search(key, seq):
    def helper(low, high): # 2, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=2
        if key == seq[mid]: # 11 == 12
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 2. If it is what we are looking for, return True

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

```
def binary_search(key, seq):
    def helper(low, high): # 2, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=2
        if key == seq[mid]: # 11 == 12
            return True
        elif key < seq[mid]: # 11 < 12
            return helper(low, mid-1) # helper(2, 2-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 3a. If key is smaller, look at left side

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

```
def binary_search(key, seq):
    def helper(low, high): # 2, 1
        if low > high:
            return False
        mid = (low + high) // 2
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 3a. If key is smaller, look at left side

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345
---	---	----	----	----	----	----	-----	-----	-----

```
def binary_search(key, seq):
    def helper(low, high): # 2, 1
        if low > high: # 2 > 1
            return False
        mid = (low + high) // 2
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Key cannot be
found. Return
False

Binary Search

- Each step eliminates the problem size by half.
 - The problem size gets reduced to 1 very quickly
- This is a simple yet powerful strategy, of halving the solution space in each step
- What is the order of growth?

$O(\log n)$

Wishful Thinking

We assumed the list was sorted.

Now, let's deal with this assumption!

Sorting

Sorting

- High-level idea:
 1. some objects
 2. function that can order two objects

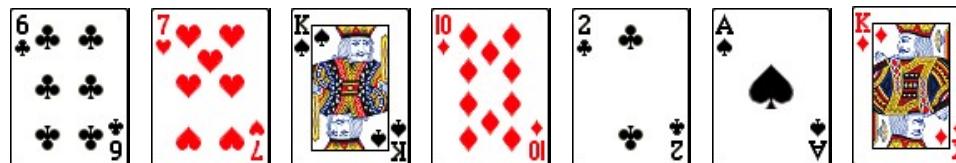
⇒ order all the objects

How Many Ways to Sort?

Too many. ☺

Example

Let's sort some playing cards?

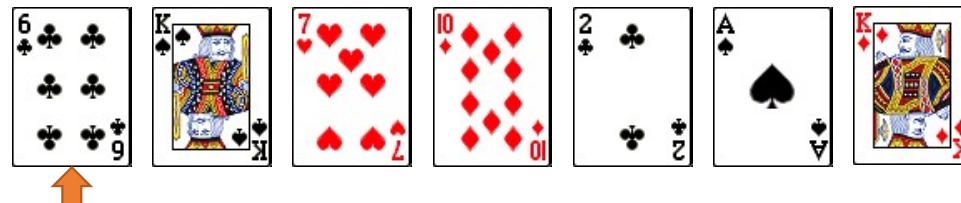


What do you do when
you play cards?

Obvious Way

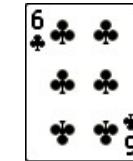
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

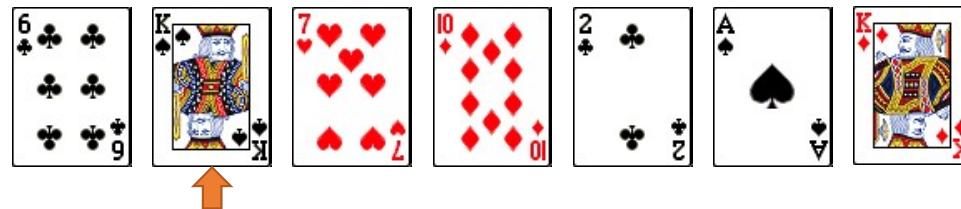
Smallest



Obvious Way

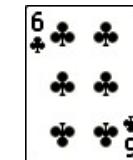
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

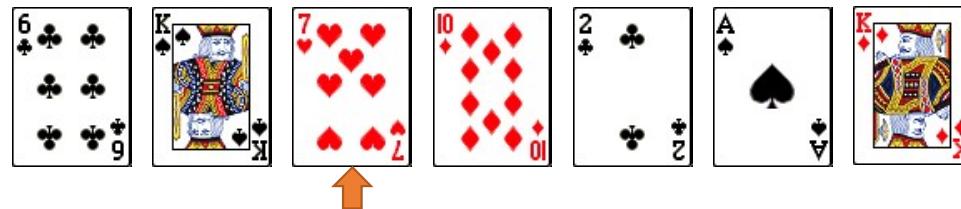
Smallest



Obvious Way

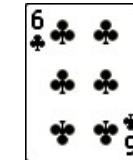
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

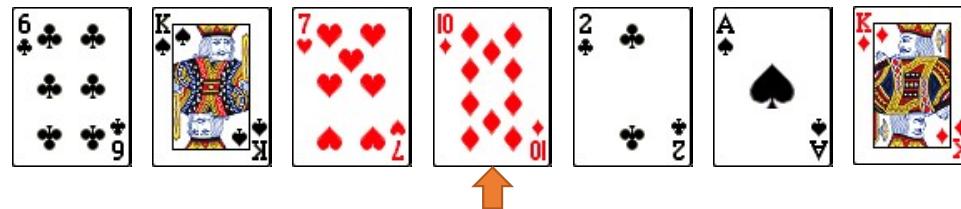
Smallest



Obvious Way

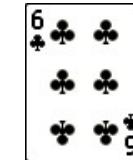
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

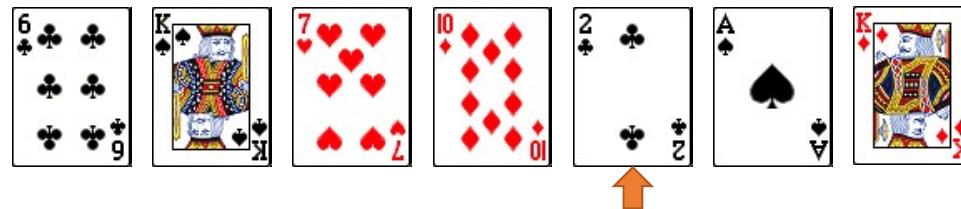
Smallest



Obvious Way

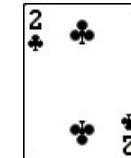
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

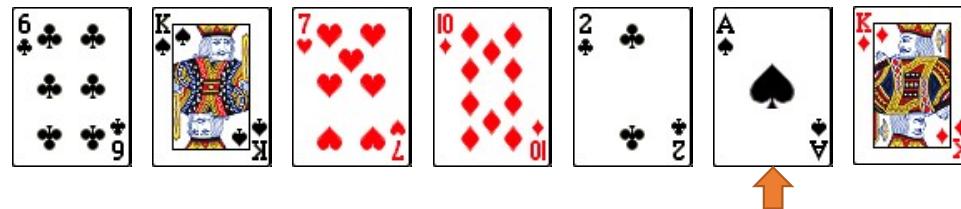
Smallest



Obvious Way

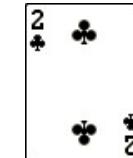
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

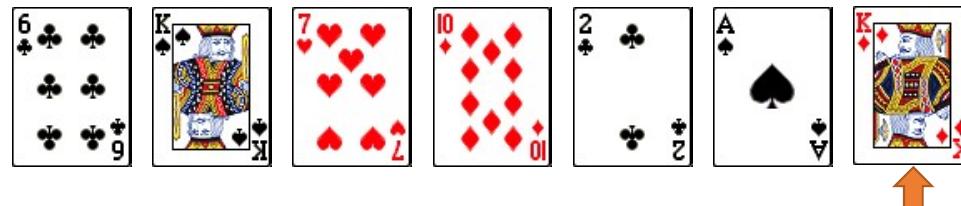
Smallest



Obvious Way

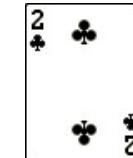
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

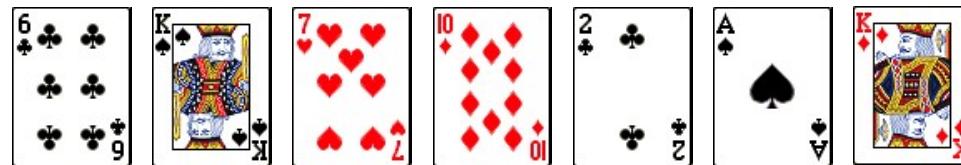
Smallest



Obvious Way

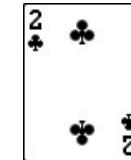
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

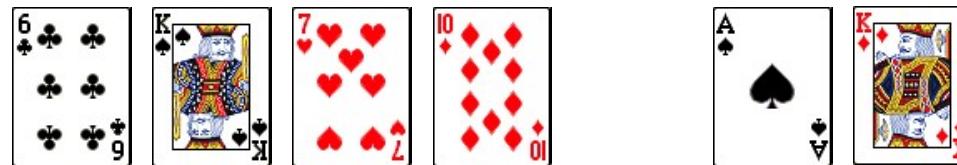
Smallest



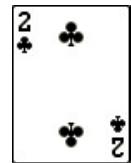
Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

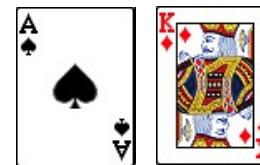
Unsorted



Sorted



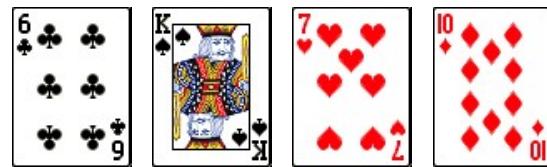
Smallest



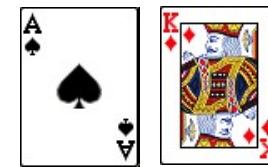
Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

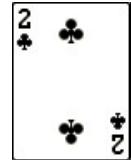
Unsorted



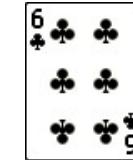
Repeat



Sorted



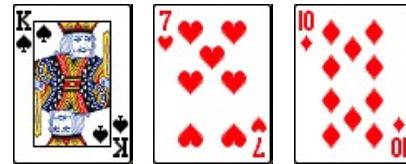
Smallest



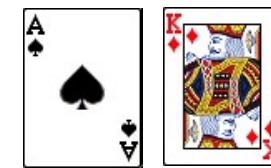
Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

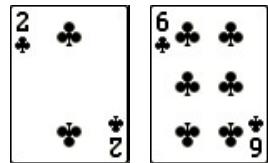
Unsorted



Repeat



Sorted

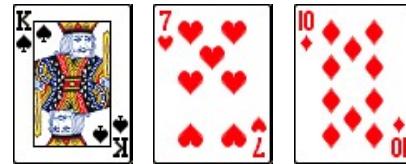


Smallest

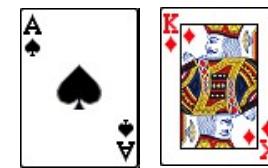
Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

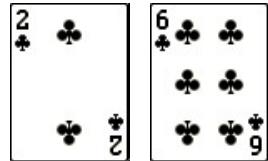
Unsorted



Repeat



Sorted



Smallest



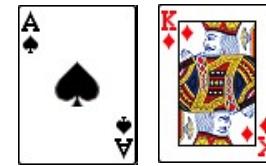
Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

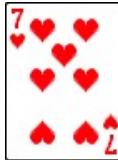
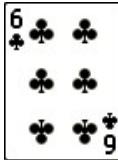
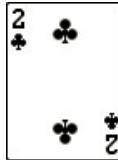
Unsorted



Repeat



Sorted



Smallest

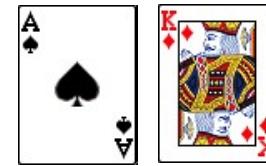
Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

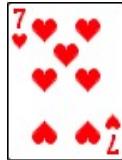
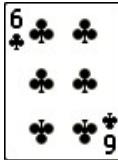
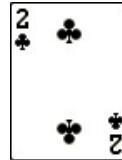
Unsorted



Repeat



Sorted



Smallest



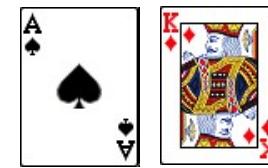
Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

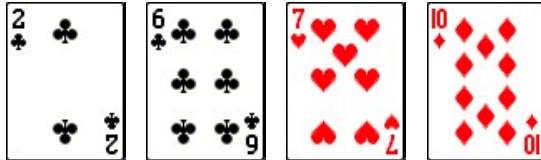
Unsorted



Repeat



Sorted



Smallest

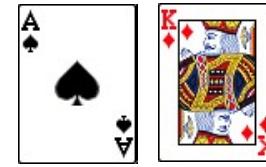
Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

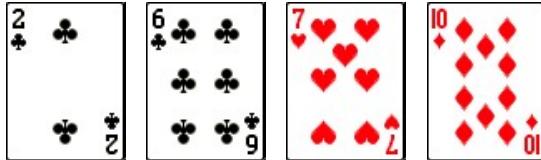
Unsorted



Repeat



Sorted



Smallest



Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

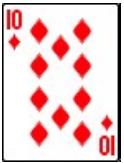
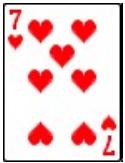
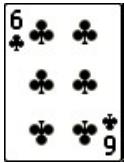
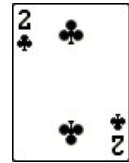
Unsorted



Repeat



Sorted



Smallest

Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

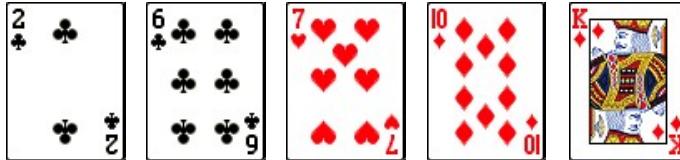
Unsorted



Repeat



Sorted



Smallest



Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

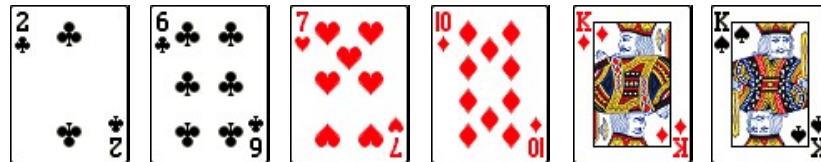
Unsorted

Repeat



Sorted

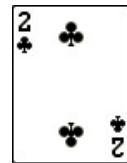
Smallest



Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

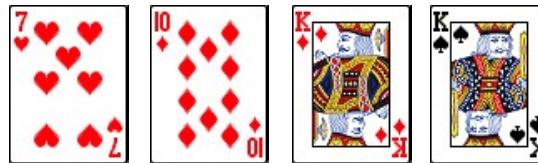
Unsorted



Repeat



Sorted



Smallest



Obvious Way

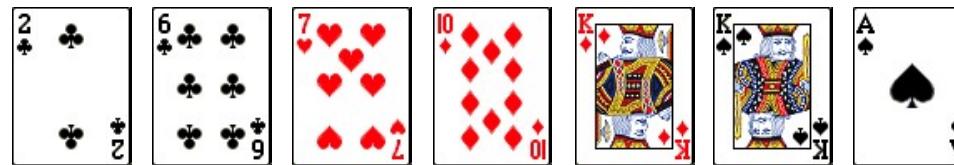
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted

Done

Sorted

Smallest



There is actually a name for this:
Selection Sort!

Let's Implement it!

```
a = [4,12,3,1,11]
sort = []

while a:    # a is not []
    smallest = a[0]
    for element in a:
        if element < smallest:
            smallest = element
    a.remove(smallest)
    sort.append(smallest)
print(a)
```

Output

```
[4, 12, 3, 11]
[4, 12, 11]
[12, 11]
[12]
[]
print(a)
[]

print(sort)
[1, 3, 4, 11, 12]
```

Order of Growth?

- Time:

Worst	$O(n^2)$
Average	$O(n^2)$
Best	$O(n^2)$
- Space: ~~$O(n)$~~ $O(1)$

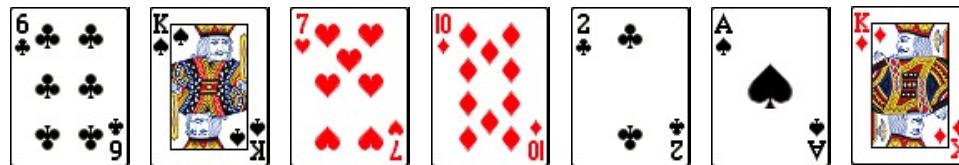
Let's try something
else...

suppose you have a
friend

Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

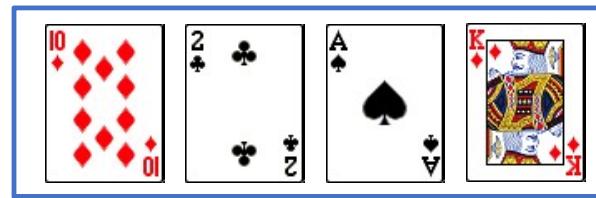
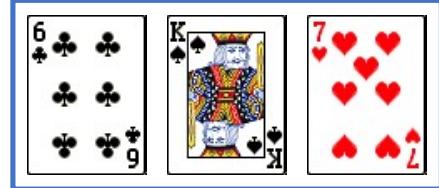
Split into halves



Doing it with a friend

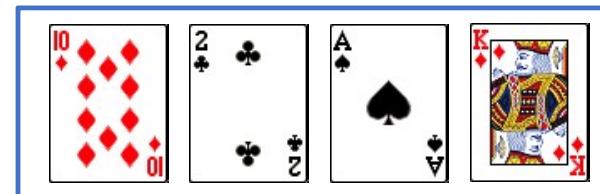
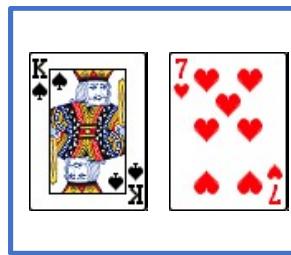
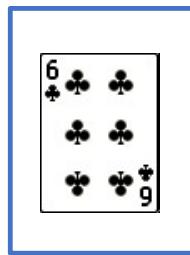
- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

Split into halves



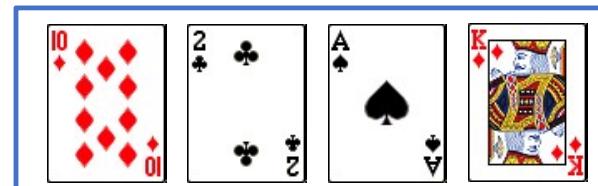
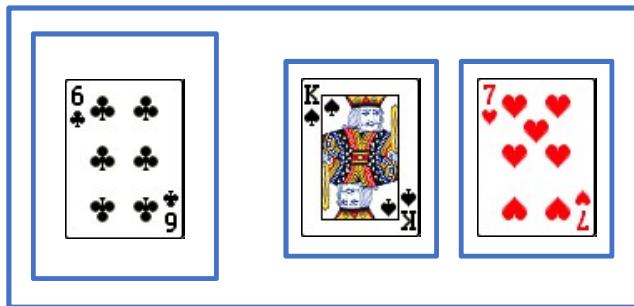
Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



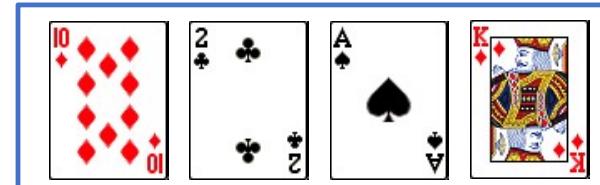
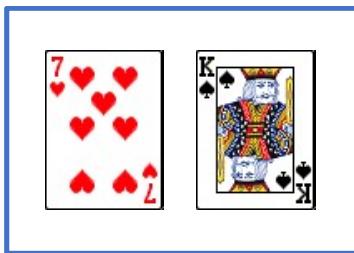
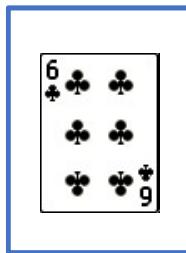
Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



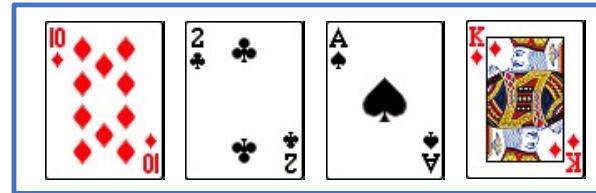
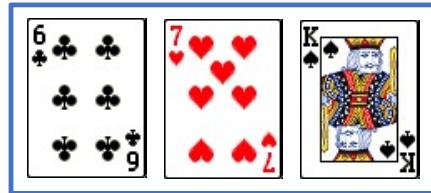
Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



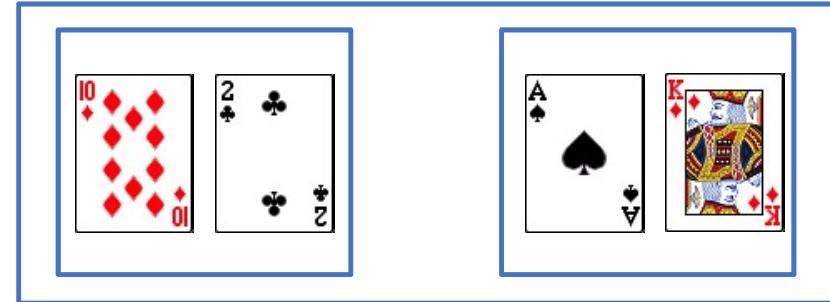
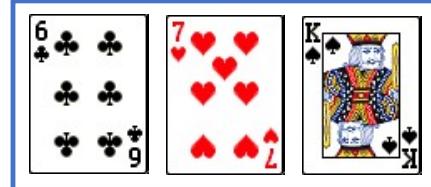
Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



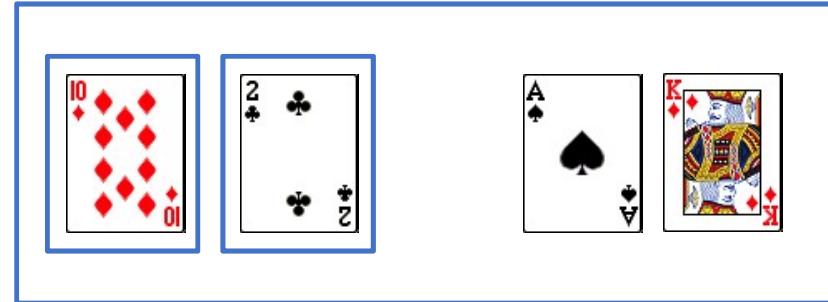
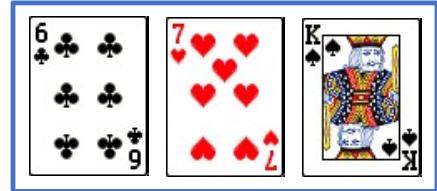
Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



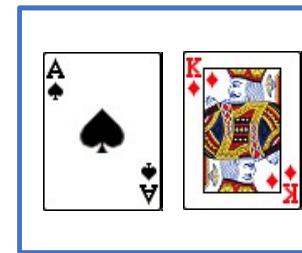
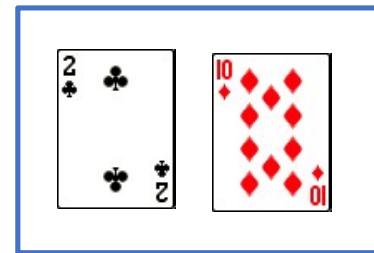
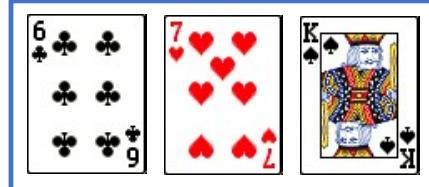
Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



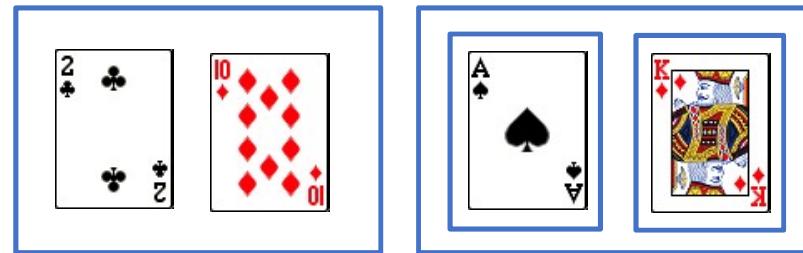
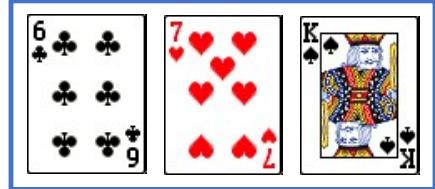
Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



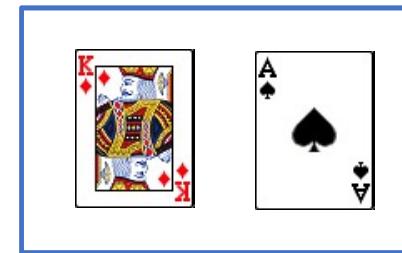
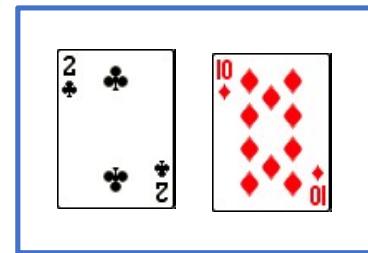
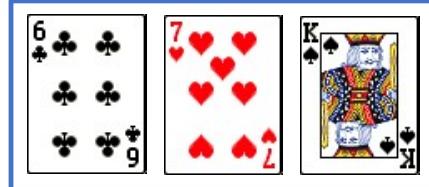
Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



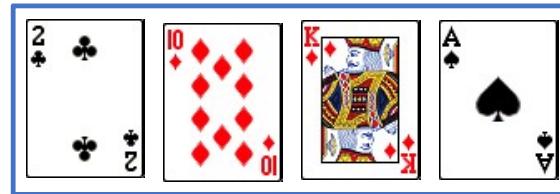
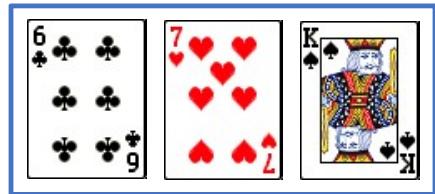
Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



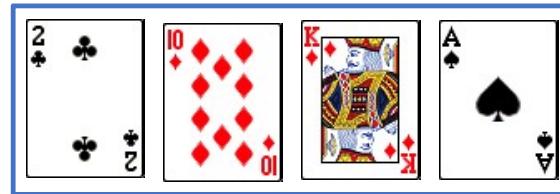
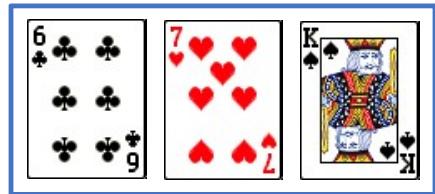
Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

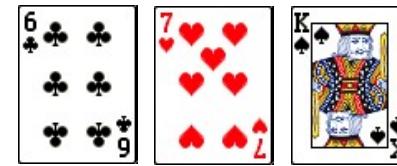
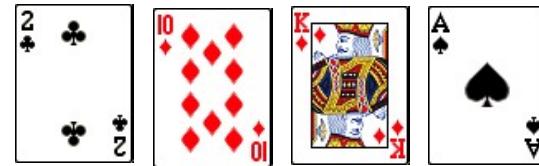


How to combine the 2 sorted halves?

Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

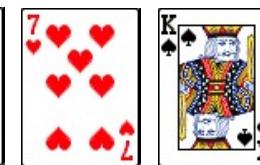
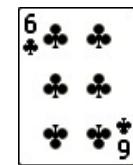
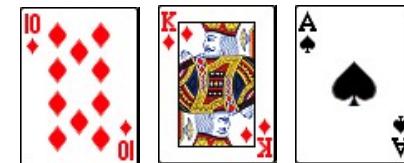
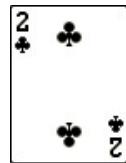
Compare first elements



Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

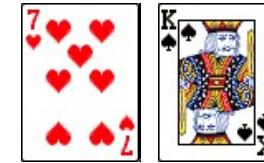
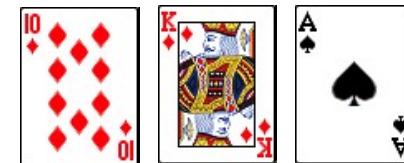
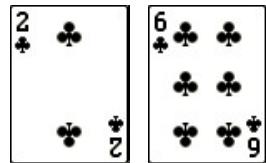
Compare first elements



Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

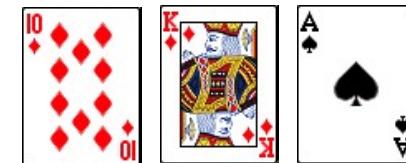
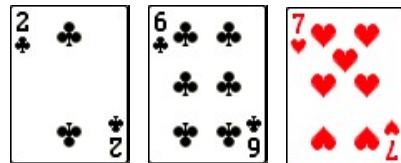
Compare first elements



Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

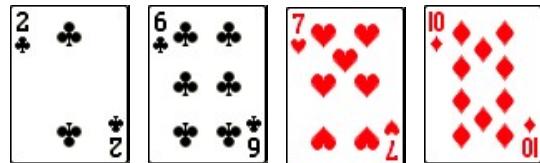
Compare first elements



Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

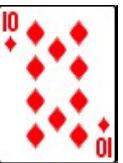
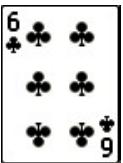
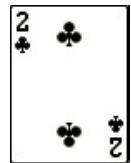
Compare first elements



Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

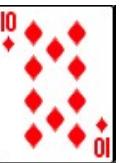
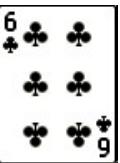
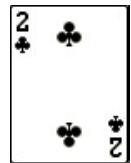
Compare first elements



Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

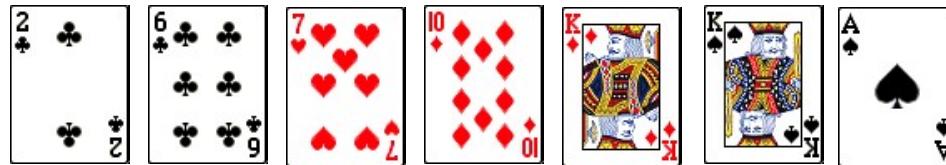
Compare first elements



Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

Compare first elements



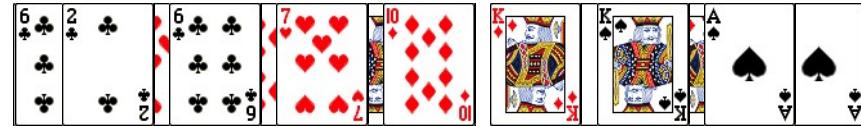
There is also a name for this:

Merge Sort!

Let's Implement It!

First observation: **RECURSION!**

- Base case: $n < 2$, return lst
- Otherwise:



- Divide list into two
- Sort each of them
- Merge!

Merge Sort

```
def merge_sort(lst):
    if len(lst) < 2: # Base case!
        return lst
    mid = len(lst) // 2
    left = merge_sort(lst[:mid]) #sort left
    right = merge_sort(lst[mid:]) #sort right
    return merge(left, right)
```

How to merge?

How to merge?

- Compare first element
- Take the smaller of the two
- Repeat until no more elements

Merging

```
def merge(left, right):
    results = []
    while left and right:
        if left[0] < right[0]:
            results.append(left.pop(0))
        else:
            results.append(right.pop(0))
    results.extend(left)
    results.extend(right)
    return results
```

Order of Growth?

- Time:

Worst	$O(n \log n)$
Average	$O(n \log n)$
Best	$O(n \log n)$
- Space: $O(n)$

No need to memorize

Sort Properties

In-place: uses a small, constant amount of extra storage space, i.e., $O(1)$ space

Selection Sort: No (Possible)

Merge Sort: No (Possible)

Sort Properties

Stability: maintains the relative order of items with equal keys (i.e., values)

Selection Sort: Yes (maybe)

Merge Sort: Yes

How Many Ways to Sort?

Too many. ☺

Summary

- Python Lists are mutable data structures
- Searching
 - Linear Search
 - Binary Search: Divide-and-conquer
- Sorting
 - Selection Sort
 - Merge Sort: Divide-and-conquer + recursion
 - Properties: In-place & Stability

Ok, now I know how to guess a number
quickly

SO WHAT?



Google?



why professor

- why professor **x still alive in logan**
- why professor **x can't walk**
- why professor **not in nba**
- why professor **mcgonagall is the best**
- why professor **x died in logan**
- why professor **green name**
- why professors **are arrogant**
- why professor **snape killed dumbledore**
- why professors **don't reply**
- why professors **are liberal**

How much data does Google handle?

- About 10 to 15 Exabyte of data
 - 1 Exabyte(EB)= 1024 Petabyte(PB)
 - 1 Petabyte(PB) = 1024 Terabytes(TB)
 - 1 Terabyte(PB) = 1024 Gigabytes(TB)
 - = 4 X 256GB iPhone
- So Google is handling about 60 millions of iPhones

How fast is my desktop?

```
import time

def create_list(n):
    start_time = time.time()
    l = [i for i in range(n)]
    end_time = time.time()
    print('Duration = ' +
          str(end_time-start_time) + ' s')
```

```
create_list(1000)
create_list(1000*1000)
create_list(1000*1000*100)
```

Output:

```
Duration = 0.0 s
Duration = 0.04769182205200195 s
Duration = 6.026865720748901 s
```

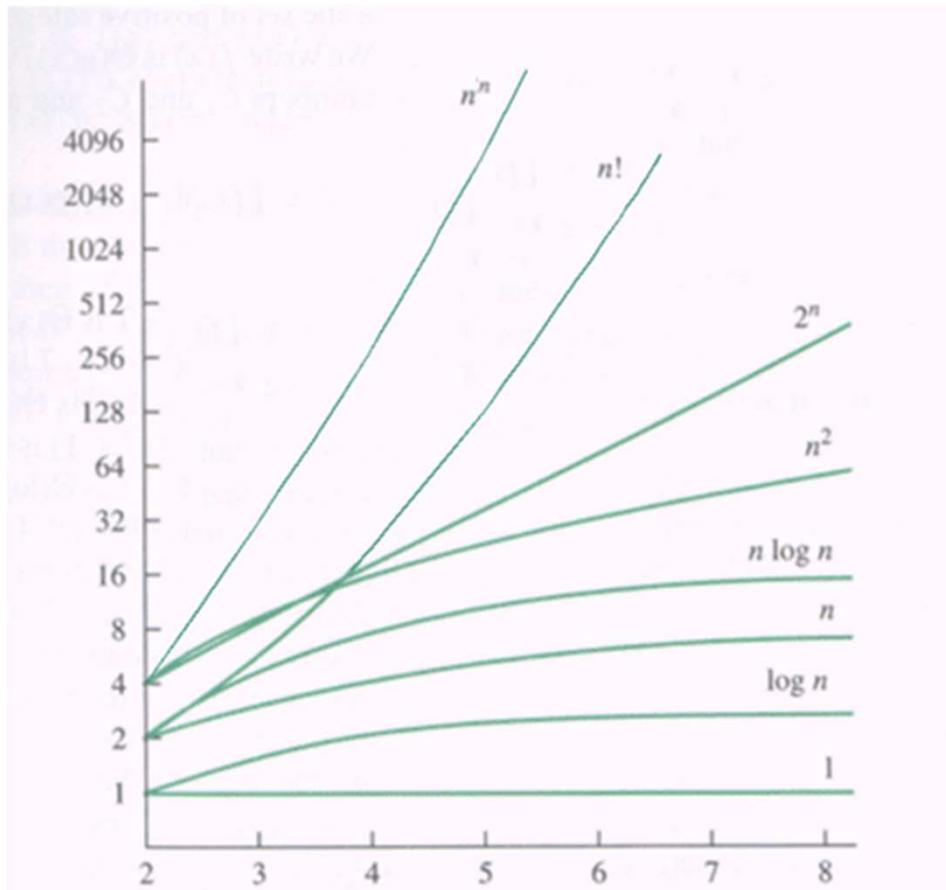
Return the time in seconds since the epoch (1970/1/1 00:00) as a floating point number.

Create 100M of numbers, estimated to be 400MB of data

Let's calculate

- 400M of data needs 6 seconds
- 15 Exabyte of data needs how long?
 - $15 \text{ EB} = 15 \times 1024 \times 1024 \times 1024 \times 1024 \text{ MB}$
 - To search through 15EB of data.....
 - 7845 years....
- If we do it with Binary Search
 - $\log_2 (15\text{EB}) = \text{43 steps!!!!}$

The Power of Algorithm



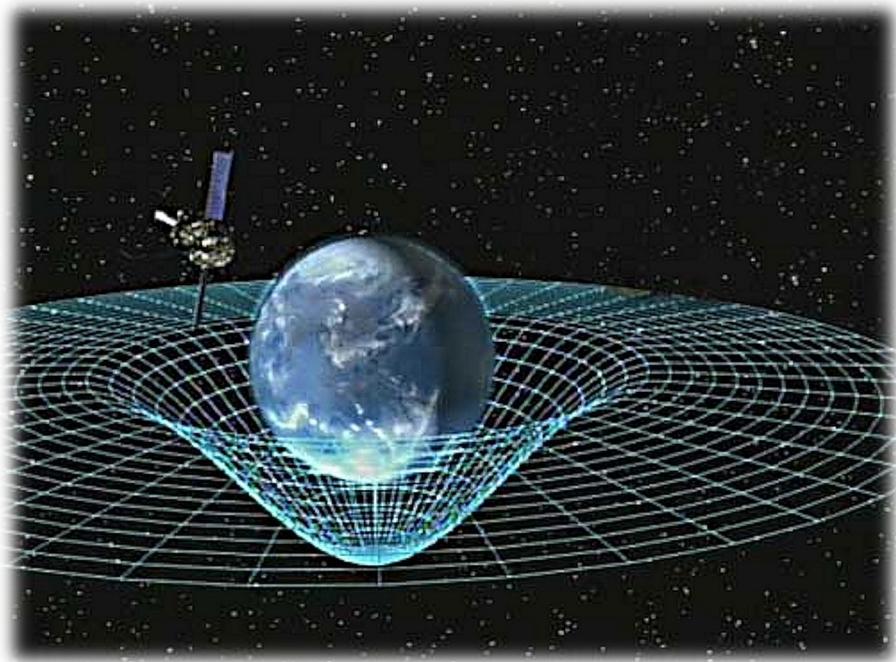
Order of Growth

In Physics, We consider

- Time

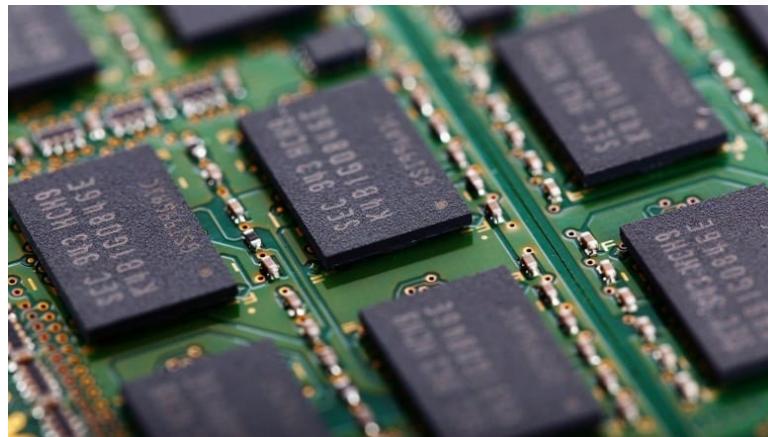


- Space



In CS, we consider

- Time
 - how long it takes to run a program
- Space
 - how much memory do we need to run the program



Order of Growth Analogy

- Suppose you want to buy a Blu-ray movie from Amazon (~40GB)
- Two options:
 - Download
 - 2-day Prime Shipping
- Which is faster?



The Infinity Saga Box Set



Order of Growth Analogy

- Buy the full set?
 - 23 movies
- Two options:
 - Download
 - 2-day Prime Shipping
- Which is faster?

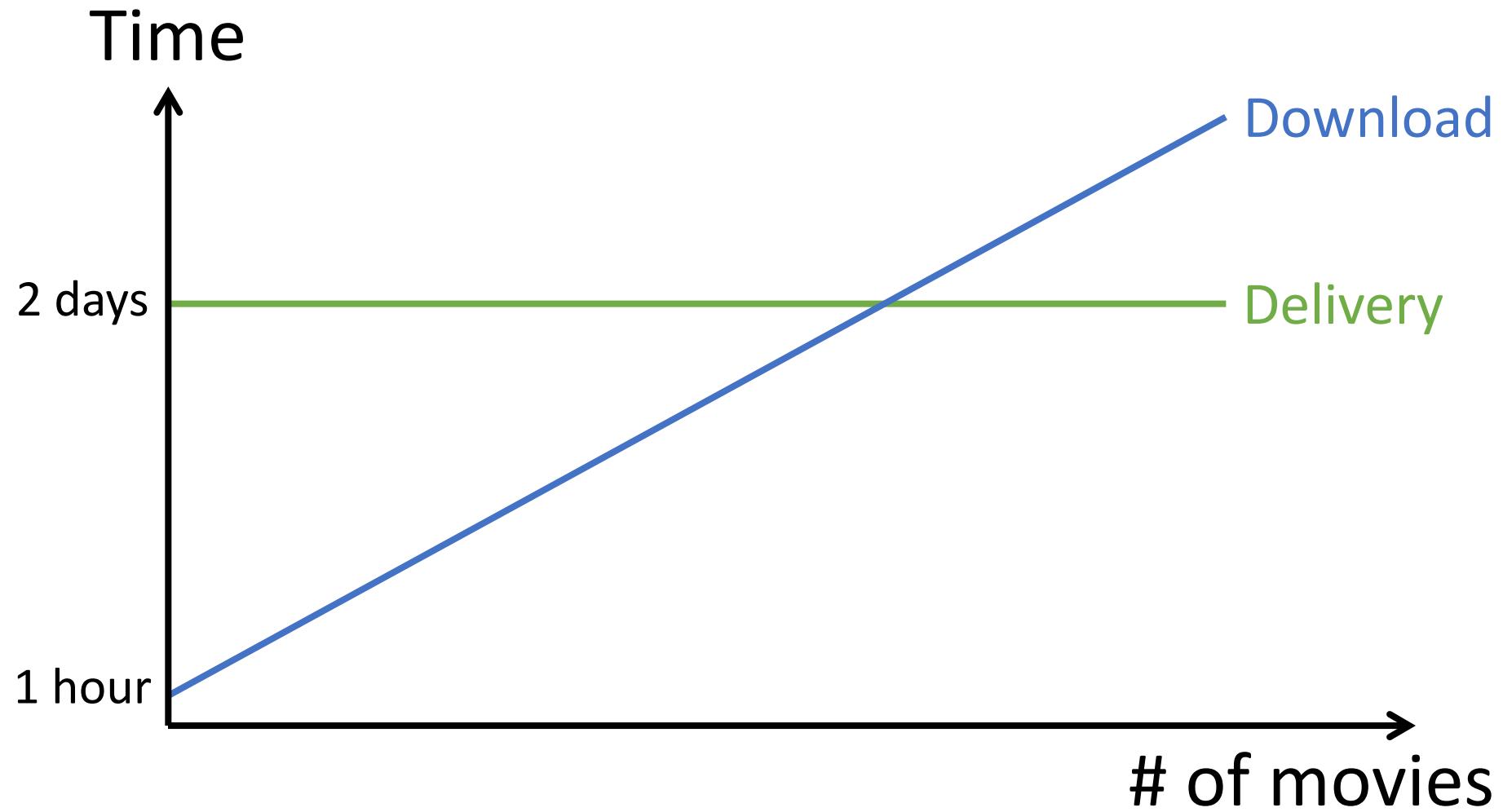


Order of Growth Analogy

- Or even more movies?



Download vs Delivery



Ultimate Question

- If the “volume” increased
- How much more resources, namely **time** and **space**, grow?

Will they grow in the same manner?

- From
 - factorial(10)
- To
 - factorial(20)
- To
 - factorial(100)
- To
 - factorial(10000)
- From
 - fib(10)
- To
 - fib(20)
- To
 - fib(100)
- To
 - fib(10000)

Order of Growth

- is NOT...
 - The **absolute** time or space a program takes to run
- is
 - the **proportion of growth** of the time/space of a program **w.r.t.** the growth of the input

Let's try it on something we know

```
def factorial(n):  
  
    if n <= 1:  
        return 1  
    else:  
        return n * factorial(n - 1)  
  
def fib(n):  
  
    if (n == 0):  
        return 0  
    elif (n == 1):  
        return 1  
    else:  
        return fib(n - 1) + fib(n - 2)
```

Let's try it on something we know

```
nfact, nfib = 0, 0
def factorial(n):
    global nfact
    nfact +=1
    if n <= 1:
        return 1
    else:
        return n * factorial(n - 1)

def fib(n):
    global nfib
    nfib +=1
    if (n == 0):
        return 0
    elif (n == 1):
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Compare

```
>>> factorial(5)
```

```
120
```

```
>>> nfact
```

```
5
```

```
>>> nfact = 0
```

```
>>> factorial(10)
```

```
3628800
```

```
>>> nfact
```

```
10
```

```
>>> nfact = 0
```

```
>>> factorial(20)
```

```
2432902008176640000
```

```
>>> nfact
```

```
20
```

```
>>> fib(5)
```

```
5
```

```
>>> nfib
```

```
15
```

```
>>> nfib = 0
```

```
>>> fib(10)
```

```
55
```

```
>>> nfib
```

```
177
```

```
>>> nfib = 0
```

```
>>> fib(20)
```

```
6765
```

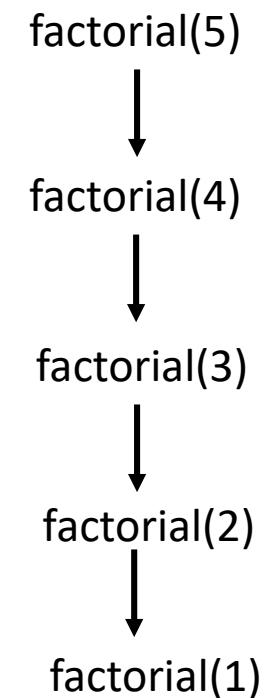
```
>>> nfib
```

```
21891
```

Order of Growth of Factorial

```
>>> factorial(5)
120
>>> nfact
5
>>> nfact = 0
>>> factorial(10)
3628800
>>> nfact
10
>>> nfact = 0
>>> factorial(20)
2432902008176640000
>>> nfact
20
```

- Factorial is simple
 - If the input is n , then the function is called n times
 - Because each time n reduced by 1
- So the number of times of calling the function is proportional to n



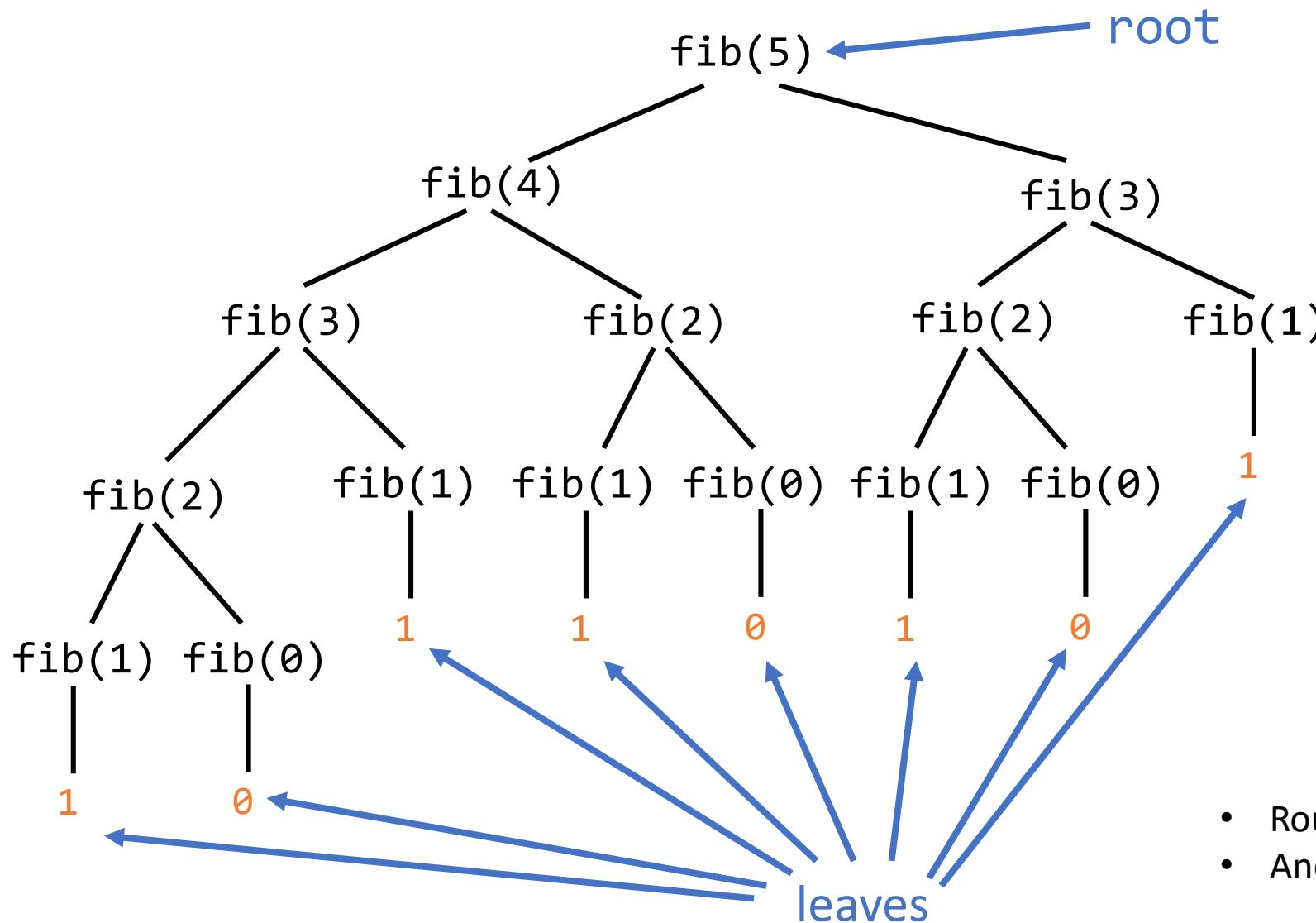
Fib

- More complicated

Why?

```
>>> fib(5)
5
>>> nfib1
5
>>> nfib = 0
>>> fib(10)
55
>>> nfib
177
>>> nfib = 0
>>> fib(20)
6765
>>> nfib
21891
```

Fibonacci: Tree recursion



- Roughly half of a full tree
- And a full tree has $2^n - 1$ nodes

Compare

```
>>> factorial(5)
```

```
120
```

```
>>> nfact
```

```
5
```

```
>>> nfact = 0
```

```
>>> factorial(10)
```

```
3628800
```

```
>>> nfact
```

```
10
```

```
>>> nfact = 0
```

```
>>> factorial(20)
```

```
2432902008176640000
```

```
>>> nfact
```

```
20
```

```
>>> fib(5)
```

```
5
```

```
>>> nfib1
```

```
5
```

```
>>> nfib = 0
```

```
>>> fib(10)
```

```
55
```

```
>>> nfib
```

```
177
```

```
>>> nfib = 0
```

```
>>> fib(20)
```

```
6765
```

```
>>> nfib
```

```
21891
```

No of calls proportional to n

No of calls proportional to 2^n

Searching in a list of n items

- Linear search
 - # comparisons proportional to n
 - (Because in average, the expected number of search is $n/2$)
- Binary search
 - # comparisons proportional to $\log n$
 - Because, we divide the list into half for at most $\log n$ times

Bubble Sort

8	4	5	9	2	3	7	1	6	0
4	8	5	9	2	3	7	1	6	0
4	5	8	9	2	3	7	1	6	0
4	5	8	9	2	3	7	1	6	0
4	5	8	2	9	3	7	1	6	0
4	5	8	2	3	9	7	1	6	0
4	5	8	2	3	7	9	1	6	0
4	5	8	2	3	7	1	9	6	0
4	5	8	2	3	7	1	6	9	0
4	5	8	2	3	7	1	6	0	9

Bubble Sort

4	5	8	2	3	7	1	6	0	9
4	5	8	2	3	7	1	6	0	9
4	5	8	2	3	7	1	6	0	9
4	5	2	8	3	7	1	6	0	9
4	5	2	3	8	7	1	6	0	9
4	5	2	3	7	8	1	6	0	9
4	5	2	3	7	8	1	6	0	9
4	5	2	3	7	1	8	6	0	9
4	5	2	3	7	1	8	6	0	9
4	5	2	3	7	1	6	8	0	9
4	5	2	3	7	1	6	0	8	9

Bubble Sort

```
def bubble(my_list):
    for i in range(len(my_list)-1):
        if my_list[i] > my_list[i+1]:
            if my_list[i] > my_list[i+1]:
                my_list[i], my_list[i+1] = my_list[i+1], my_list[i]

def bubblesort(my_list):
    for i in range(len(my_list)-1):
        bubble(my_list)

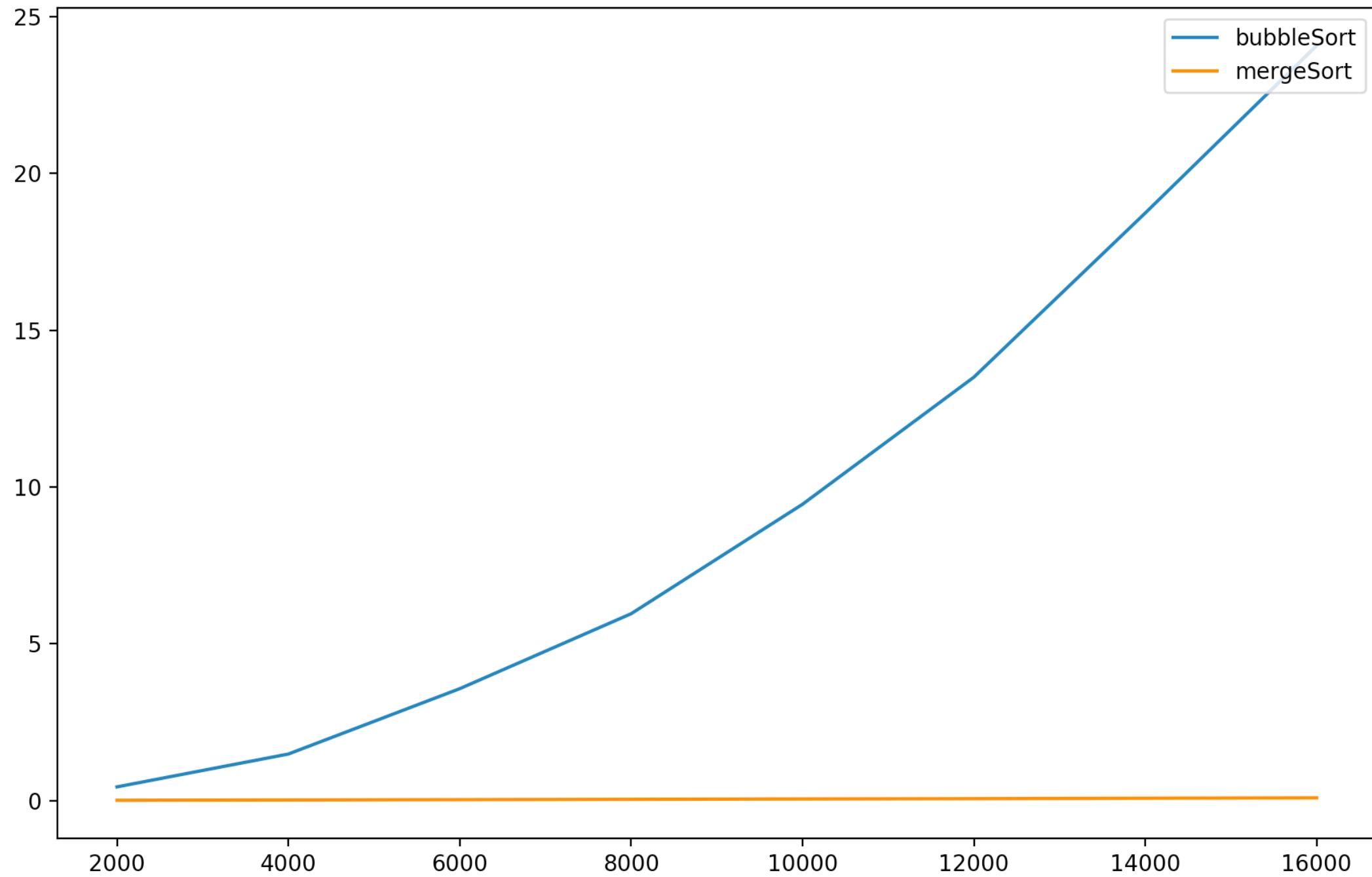
my_list_1 = [38,2,10,3,1]
bubblesort(my_list_1)
print(my_list_1)
```

Sorting a list of n Items

- Selection/Bubble Sort
 - # comparisons proportional to n^2
 - Because we looped n times, and each time you need to arrange 1 to n items
- Merge sort
 - # comparisons proportional to $n \log n$
 - Because, we divide the list into half for at most $\log n$ times
 - And each time arrange n items

```
from random import randint
from time import time
ln = [2000,4000,6000,8000,10000,12000,14000,16000]

bstat = []
mstat = []
for n in ln:
    rl = [randint(1,100000) for i in range(n)]
    st = time()
    bubbleSort(rl)
    btime = time()-st
    st = time()
    mergeSort(rl)
    mtime = time()-st
    print(f'For n = {n}, bubbleSort: {btime}s mergeSort: {mtime}s')
    bstat.append(btime)
    mstat.append(mtime)
```

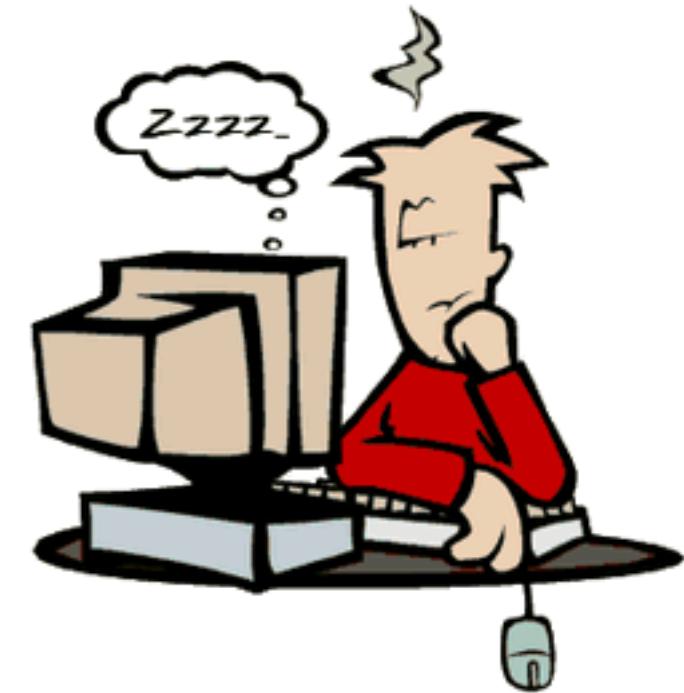


Algorithm

Anyone can give some algorithms

BogoSort

- BogoSort (L)
 - Repeat:
 - Choose a random permutation of the list L .
 - If L is sorted, return L .
 - If you wait enough time, L is sorted?



Bogo Sort

- Randomly shuffle the list till the list is sorted

```
import random
def is_not_sorted(shuffled_list):
    for i in range(len(shuffled_list)-1):
        if shuffled_list[i] > shuffled_list[i+1]:
            return True
    return False

def bogosort(my_list):
    while is_not_sorted(my_list):
        random.shuffle(my_list)

my_list = [38,2,10,3,1]
bogosort(my_list)
print(my_list)
```

Can we do better?

Hill-Climbing for Sorting

- Optimization algorithm
 - Require an evaluation function
- Which metric is better for evaluation?
 - Let my_list be our list
 - Number of index pairs i, j such that $my_list[i] > my_list[j]$

- Example:

```
>>> my_list = [38, 2, 10, 3, 1]
>>> my_list
[38, 2, 10, 3, 1]
```


$$Value(my_list) = 4 + 1 + 2 + 1 = 8$$

```
>>> my_list = [1, 2, 3, 10, 38]
>>> my_list
[1, 2, 3, 10, 38]
```


$$Value(my_list) = 0 + 0 + 0 + 0 = 0$$

Hill-Climbing for Sorting

- Repeat the following either till value of the list is zero or a predetermined number of times
 - Shuffle the list
 - Accept the shuffled list if its value is lower than that of current list

Algorithm

Anyone can give some algorithm

But how fast is your algorithm?

How about

QuantumBogoSort ($A[1..n]$)

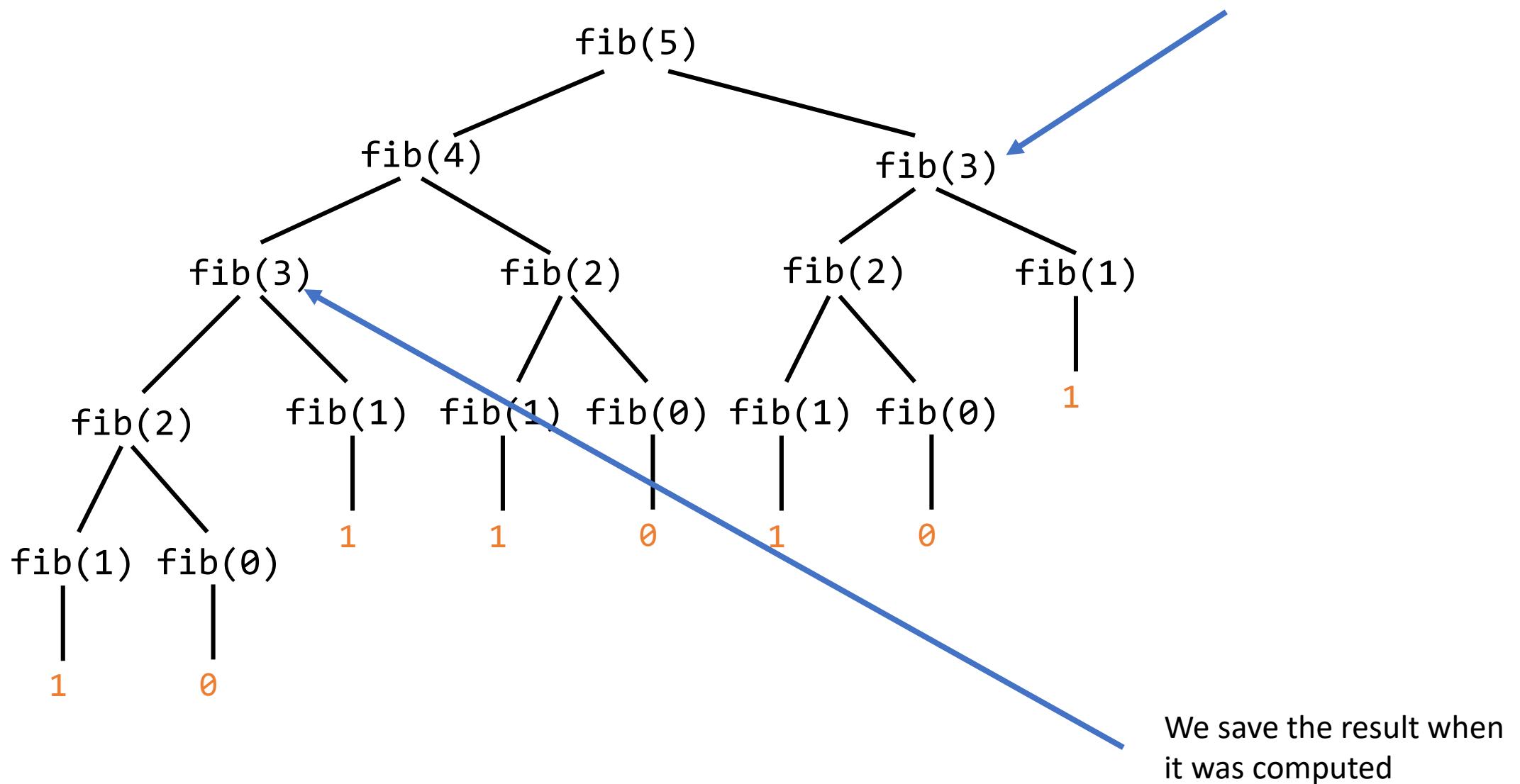
- a) Choose a random permutation of the array A .
- b) If A is sorted, return A .
- c) If A is not sorted, destroy the universe.

- Remember QuantumBogoSort when you learn about non-deterministic Turing Machines

Improvement?

Let's try $\text{fib}(n)$

Easy Way: Memoization



Memoization

- Create a dictionary to remember the answer if `fibm(n)` is computed before
- If the *ans* was computed before, get the answer from the dictionary
- Otherwise, compute the *ans* and put it into the dictionary for later use

```
fibans = {}

def fibm(n):
    if n in fibans.keys():
        return fibans[n]

    if (n == 0):
        ans = 0
    elif (n == 1):
        ans = 1
    else:
        ans = fibm(n - 1) + fibm(n - 2)
    fibans[n] = ans
    return ans
```

Can you use Memoization to
compute nCk ?



Recursion Removal

- Store all the answers in an array
- Add a new fib(i)
 - as $\text{fib}(i-1) + \text{fib}(i-2)$
- Wait a min...
 - Do we need all the past numbers if we only need $\text{fib}(n)$?

```
def fibi(n):
    ans = [0,1,1]
    if n < 3:
        return ans[n]
    for i in range(3,n+1):
        ans.append(ans[i-1]+ans[i-2])
    return ans[n]
```

Recursion Removal 2

- Add a new $\text{fib}(i)$
 - as $\text{fib}(i-1) + \text{fib}(i-2)$
- And I only need to keep $\text{fib}(i-1)$ and $\text{fib}(i-2)$

```
def fibi2(n):
    if n < 3:
        return 1
    fibminus1, fibminus2 = 1, 1
    for i in range(3,n+1):
        fibminus2, fibminus1 = fibminus1, fibminus1 + fibminus2
    return fibminus1
```

Improvement

- For IT5001, you should know how to compute $\text{fib}(n)$ with time proportional to n
 - The fastest algorithm to compute $\text{fib}(n)$ with time proportional to $\log n$
- To know more about this ~~to improve human race and save the world~~
 - CS1231, CS2040, CS3230, etc
- What you learn today is called the Big O notation
 - $O(n)$, $O(\log n)$, $O(n^2)$, $O(n \log n)$, $O(2^n)$, etc

Errors And Exception



Types of Errors

- Until now error messages haven't been more than mentioned, but you have probably seen some
- Two kinds of errors (in Python):
 1. Syntax errors
 2. Exceptions

Syntax Errors

```
>>> while True print('Hello world')
SyntaxError: invalid syntax
```

Exceptions

- Errors detected during execution are called exceptions
- Examples:

Type of Exception	Description
NameError	If an identifier is used before assignment
TypeError	If wrong type of parameter is sent to a function
ValueError	If function parameter has invalid value (Eg: log(-1))
ZeroDivisionError	If 0 is used as divisor
StopIteration	Raised by next(iter)
IndexError	If index is out of bound for a sequence
KeyError	If non-existent key is requested for set or dictionary
IOError	If I/O operation fails (eg: opening a file)
EOFError	If end of file is reached for console or file input
AttributeError	If an undefined attribute of an object is used

NameError

```
>>> 4 + spam*3
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#4>", line 1, in <module>
```

```
    4 + spam*3
```

```
NameError: name 'spam' is not defined
```

TypeError

```
>>> '2' + 2
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    '2' + 2
TypeError: Can't convert 'int' object to str
implicitly
```

ValueError

```
>>> int('one')
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    int('one')
ValueError: invalid literal for int() with base
10: 'one'
```

ZeroDivisionError

```
>>> 10 * (1/0)
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#3>", line 1, in <module>
```

```
    10 * (1/0)
```

```
ZeroDivisionError: division by zero
```

Other Common Errors

StopIteration Error

```
>>> x = range(2)
>>> x_iter = iter(x)
>>> next(x_iter)
0
>>> next(x_iter)
1
>>> next(x_iter)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    next(x_iter)
StopIteration
>>>
```

IndexError

```
>>> x = [1,2,3,4]
>>> x[5]
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    x[5]
IndexError: list index out of range
```

KeyError

```
>>> x = {1:'abc',2:'def'}
>>> x[4]
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    x[4]
KeyError: 4
```

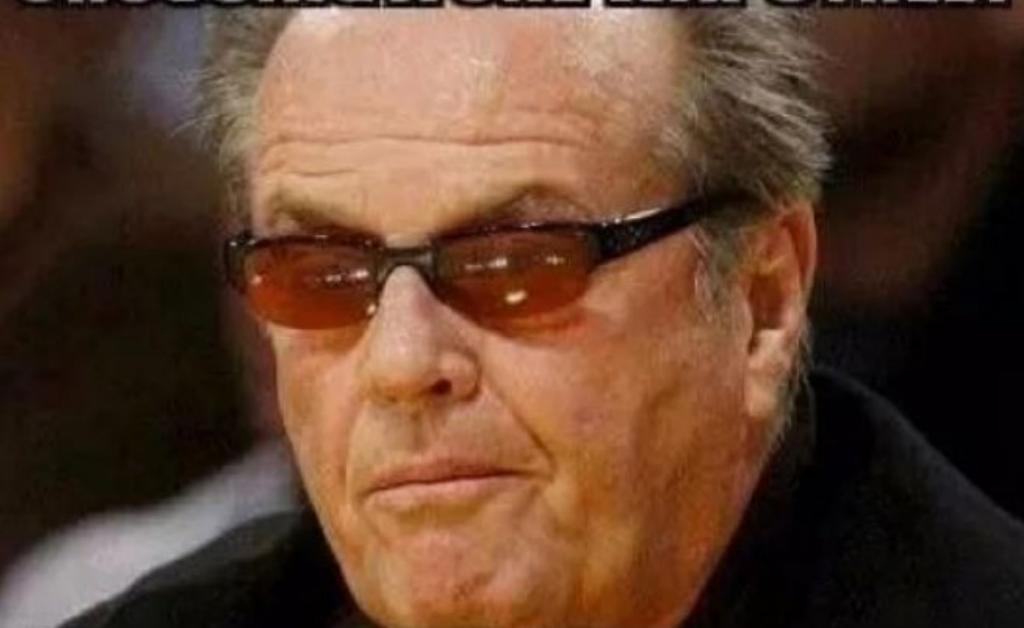
Handling Exceptions (Errors)



Handling Exceptions

- Two Approaches
 - Using Guard Clauses
 - Using Try-Except-Else constructs

I LOOK BOTH WAYS BEFORE
CROSSING A ONE WAY STREET



THAT'S HOW LITTLE FAITH
I HAVE LEFT IN HUMANITY

Guard Clauses



Guard is a Boolean expression that must evaluate to *True* if the program execution is to continue in the branch in question

Raising Exceptions

```
def add_two_integers(x,y):  
    """  
        Arguments:  
            x and y must be of type integers  
        Returns:  
            sum of two integers x and y  
    """  
    return x+y  
  
add_two_integers('abc','def')
```

Raising Exceptions

- The `raise` statement allows the programmer to force a specific exception to occur:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

Raising Exceptions

```
def add_two_integers_1(x,y):  
    """  
    Arguments:  
        x and y must be of type integers  
    Returns:  
        sum of two integers (or floats) x and y  
    """  
    if not isinstance(x,int):  
        raise TypeError('First argument must be of type integer')  
    if not isinstance(y,int):  
        raise TypeError('Second argument must be of type integer')  
    return x+y
```

```
z = add_two_integers_1('abc','def')
```

raise terminates the function and shows the message

```
raise TypeError('First argument must be of type integer')  
TypeError: First argument must be of type integer
```

Raising Exceptions

checking for multiple types

```
def add_two_numbers(x,y):
    """
    Arguments:
        x and y must be of either type integer or float
    Returns:
        sum of two integers (or floats) x and y
    """
    if not isinstance(x,(int,float)):
        raise TypeError('Only numerics are allowed for first argument')
    if not isinstance(y,(int,float)):
        raise TypeError('Only numerics are allowed for second argument')
    return x+y
```

```
add_two_numbers(2.0,1)
```

```
add_two_numbers('abc','def')
```

Guard Clauses: Use with Caution

- Python can raise exceptions without explicit guard clauses
- Checking for a specific exception may consume resources (eg: time)
 - Especially if it is done within a loop with several iterations

```
def divide(x,y):  
    if y == 0:  
        raise ZeroDivisionError('Second argument should be nonzero')  
    return x/y
```

```
def divide(x,y):  
    return x/y
```



```
>>> divide(2,0)  
Traceback (most recent call last):  
  File "<pyshell#7>", line 1, in <module>  
    divide(2,0)  
  File "<pyshell#6>", line 2, in divide  
    return x/y  
ZeroDivisionError: division by zero
```

Handling Exceptions

“It's easier to ask forgiveness than it is to get permission.”

Admiral Grace Hopper
computing pioneer, born December 9, 1906

Dobson's Improbable Quote of the Day



Handling Exceptions

- The simplest way to catch and handle exceptions is with a `try-except` block:

```
x, y = 5, 0
try:
    z = x/y
except ZeroDivisionError:
    print("divide by zero")
```

How it works

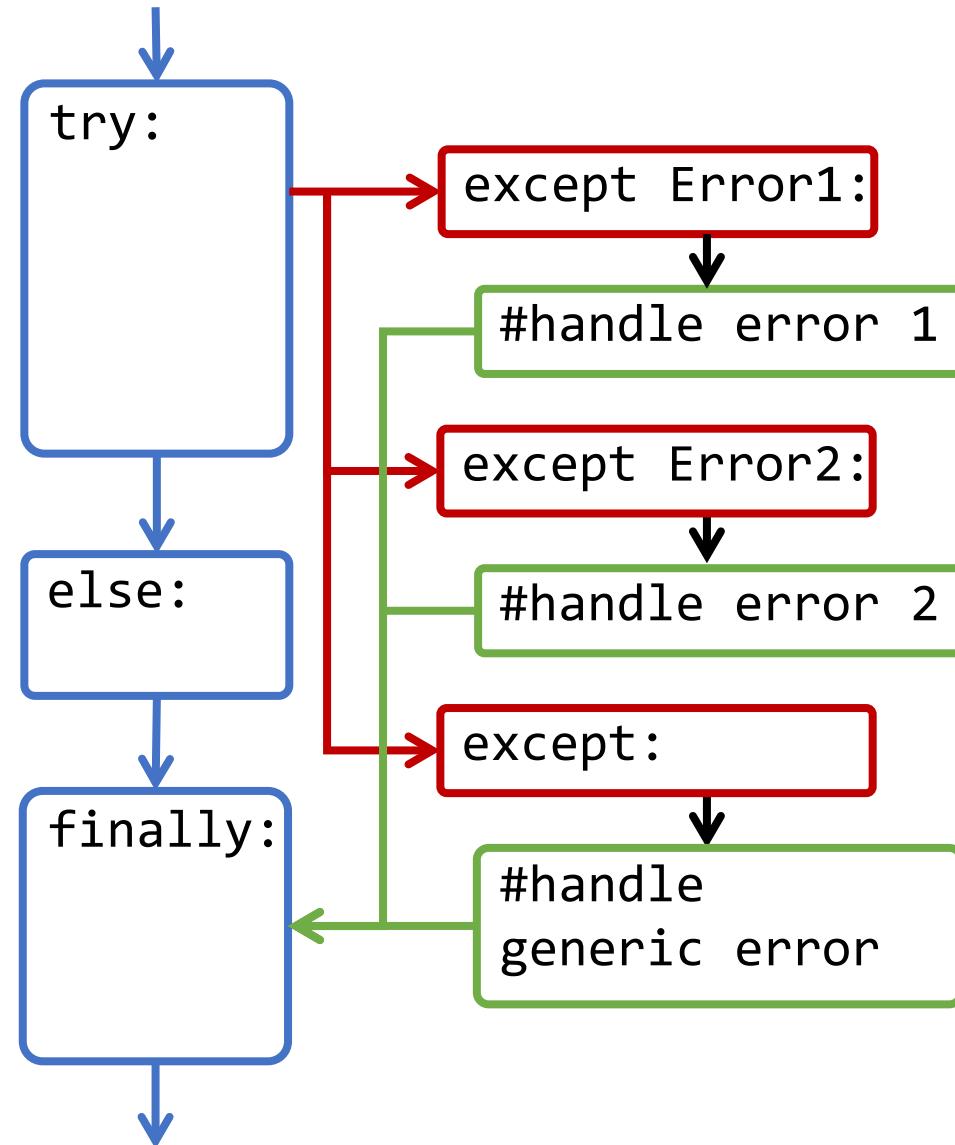
- The `try` clause is executed
- If an exception occurred, skip the rest of the `try` clause, to a matching `except` clause
- If no exception occurs, the `except` clause is skipped (go to the `else` clause, if it exists)
- The `finally` clause is always executed before leaving the `try` statement, whether an exception has occurred or not.

Try-Except

- A **try** clause may have more than 1 **except** clause, to specify handlers for different exception.
- At most one handler will be executed.
- Similar to **if-elif-else**
- **finally** will always be executed

Try-Except

```
try:  
    # statements  
except Error1:  
    # handle error 1  
except Error2:  
    # handle error 2  
except: # wildcard  
    # handle generic error  
else:  
    # no error raised  
finally:  
    # always executed
```



Try-Except Example

```
def divide_test(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
```

Try-Except Blocks

```
>>> divide_test(2, 1)
result is 2.0
executing finally clause
```

```
>>> divide_test(2, 0)
division by zero!
executing finally clause
```

```
>>> divide_test("2", "1")
executing finally clause
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    File "<stdin>", line 3, in divide
      TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

```
def divide_test(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally
              clause")
```

Simulation of *for* loop using *while* Loop

```
x = range(4)
x_iter = iter(x)
while True:
    try:
        item = next(x_iter)
        print(item)
    except StopIteration:
        break
```

Example: How to use exceptions

- Remember our tic-tac-toe?
- We would like the user to input the number from 1 to 9
 - We assume that the user is good to enter it obediently
- But not the real life situation in life
 - There is always mistake or naughty users

1|2|3

4|5|6

7|8|9

```
Player X move:what
Traceback (most recent call last):
  File "/Volumes/Google Drive/My Drive/Python/arrays/TTT.py", line 10, in tttGamePlay()
    pos = int(input("Player X move:what"))
ValueError: invalid literal for int() with argument 'what'
```

How to make sure your user input is a number?

- Original code:

```
pos = int(input(f'Player {piece[player]} move:')) - 1
```

- You can do a lot of checking, e.g.

```
userinput = input(f'Player {piece[player]} move: ')
if userinput.isnumeric():
    #play as normal
else:
    #error and input again
```

- However, it requires:

- You can consider ALL wrong situations
- And you can check them all out with codes

Example:

```
while True:  
    try:  
        pos = int(input("Input:"))  
        break  
    except:  
        print("Wrong")  
    '
```

- If the user input an integer
 - Nothing wrong
 - break, exit the while loop
- Otherwise, go to “except:”
 - Hence, will not break the while loop

Try-Except: Checking for Single Exception

If exceptions occur rarely, Try-Except is better than guard clauses

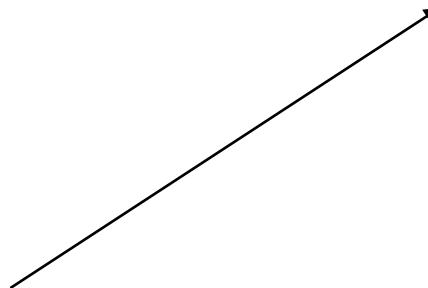
```
def divide(x, y):  
    try:  
        return x/y  
    except ZeroDivisionError:  
        print("Dividing with Zero")  
        return "NaN"
```

We can check multiple exceptions

Try-Except: Multiple Exceptions

```
import math
def my_function(x, y):
    try:
        return math.log(x) / y
    except ValueError:
        print('First argument must be nonzero positive; returning nan')
        return float("NaN")
    except ZeroDivisionError:
        print('Second argument must be nonzero; returning nan')
        return float("NaN")
```

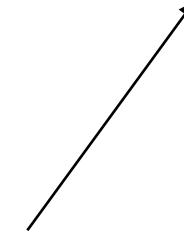
Check each exception and provide specific message



Try-Except: Multiple Exceptions

```
import math
def my_function_1(x, y):
    try:
        return math.log(x) / y
    except (ZeroDivisionError, ValueError):
        print('First argument must be nonzero positive')
        print('Second argument must be nonzero')
        print('Returning nan')
        return float("NaN")
```

Check for multiple exceptions
simultaneously



Checking for all exceptions

```
import math
def my_function(x,y):
    try:
        return math.log(x)/y
    except Exception as e:
        print(e)
        return float("NaN")
```

Try-Except-Finally

```
import math
def my_function(x,y):
    try:
        return math.log(x)/y
    except ValueError:
        print('first argument must be positive')
        return "NaN"
    except ZeroDivisionError:
        print('second argument must be nonzero')
        return "NaN"
    finally:
        print('Hello')
```

This block is always executed

Exception Types

- Built-in Exceptions:
<https://docs.python.org/3/library/exceptions.html>
- User-defined Exceptions

User-defined Exceptions I

```
class MyError(Exception):  
    def __init__(self, value):  
        self.value = value  
    def __str__():  
        return repr(self.value)
```

User-defined Exceptions II

```
try:  
    raise MyError(2*2)  
except MyError as e:  
    print('Exception value:', e.value)  
Exception value: 4  
  
raise MyError('oops!')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in ?  
    __main__.MyError: 'oops!'
```

Assertion

- For example, in tic-tac-toe, you also assume the position is from 1 to 9
- For a lot of situations, you "assume" certain conditions in your code, e.g.
 - A sorting function will only take sequences as input
 - A function checking prime number will only take in integers
 - In a certain part of your code, you expect some index i will not exceed a certain range
- In Python, you can simply add an assertion
 - If the statement following in the assertion is False, then EXCEPTIONS!
 - Raises an AssertionError

Example

- Assert that the pos must be within range

```
while True:  
    try:  
        pos = int(input("Input:"))  
        assert 0 < pos < 10  
        break  
    except:  
        print("Wrong")
```

Example

- In order to catch the particular exception of the assertion, we can

```
while True:  
    try:  
        pos = int(input("Input:"))  
        assert 0 < pos < 10  
        break  
    except AssertionError:  
        print("Your number is not in the range")  
    except:  
        print("Wrong")
```

Why use Exceptions?

- In the good old days of C, many procedures returned special ints for special conditions, i.e. -1

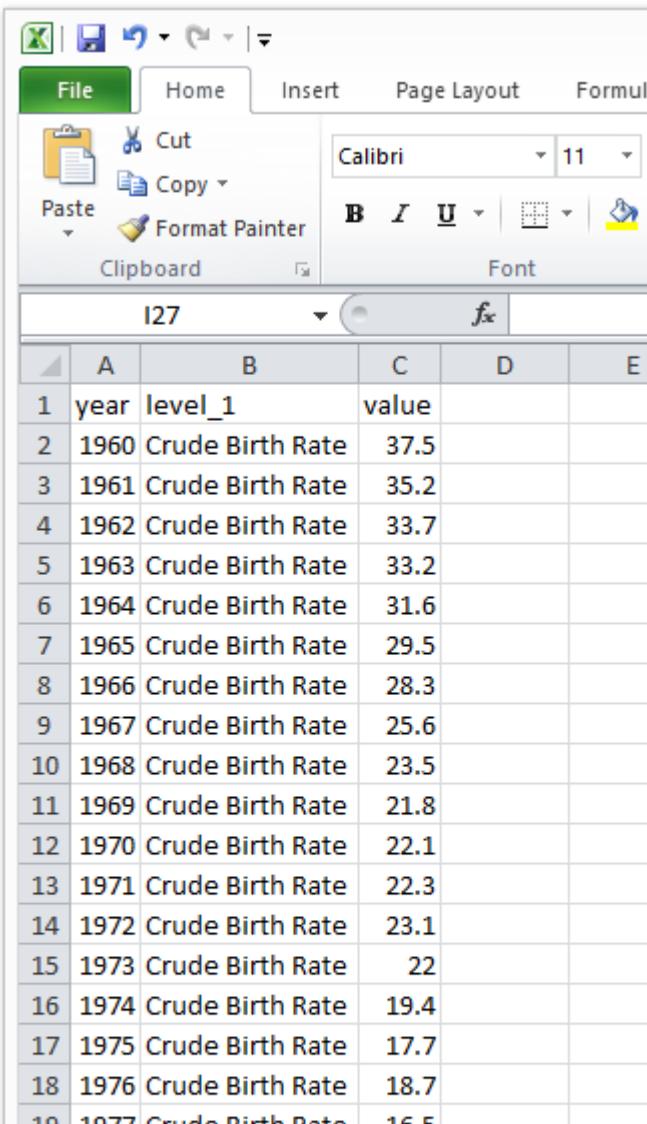
Why use Exceptions?

- But Exceptions are better because:
 - More natural
 - More easily extensible
 - Nested Exceptions for flexibility

Data Visualization with Python

“A picture is worth a thousand words”

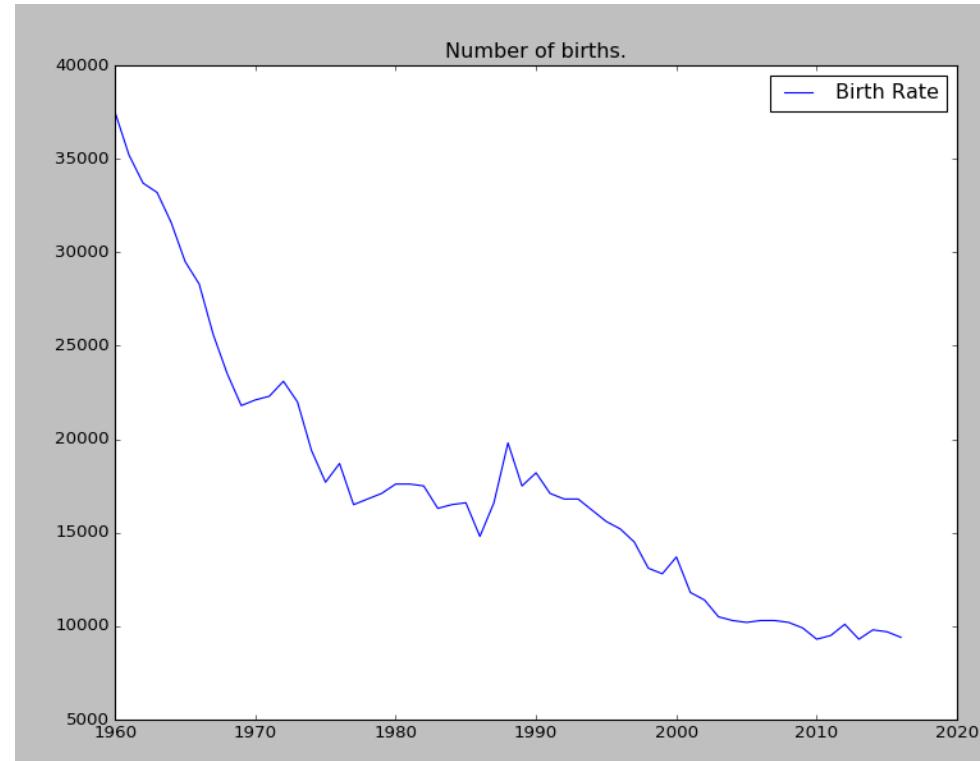
Why Visualization?!



A screenshot of Microsoft Excel showing a table of data. The table has columns labeled 'year', 'level_1', and 'value'. The data shows the Crude Birth Rate from 1960 to 1977.

	A	B	C	D	E
1	year	level_1	value		
2	1960	Crude Birth Rate	37.5		
3	1961	Crude Birth Rate	35.2		
4	1962	Crude Birth Rate	33.7		
5	1963	Crude Birth Rate	33.2		
6	1964	Crude Birth Rate	31.6		
7	1965	Crude Birth Rate	29.5		
8	1966	Crude Birth Rate	28.3		
9	1967	Crude Birth Rate	25.6		
10	1968	Crude Birth Rate	23.5		
11	1969	Crude Birth Rate	21.8		
12	1970	Crude Birth Rate	22.1		
13	1971	Crude Birth Rate	22.3		
14	1972	Crude Birth Rate	23.1		
15	1973	Crude Birth Rate	22		
16	1974	Crude Birth Rate	19.4		
17	1975	Crude Birth Rate	17.7		
18	1976	Crude Birth Rate	18.7		
19	1977	Crude Birth Rate	16.5		

VS



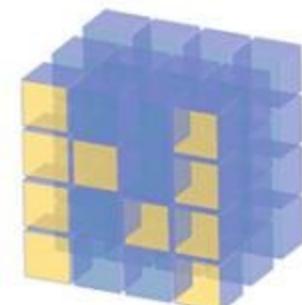
Why Visualization?

- Meet the need for speed
- For yourself, a tool to understand your data
- For your audience
 - Addressing data quality
 - Displaying meaningful results
 - Dealing with outliers
- For yourself, for your supervisors, for your boss!

Numpy

- A convenient package
- You can get around with “normal” Python method but Numpy can shorten your code a lot
- For example, if you want to create a list of numbers from 0 to π without Numpy you have to:

```
x = [i/100 for i in range(0, 314)]
```



NumPy

Numpy

```
>>> import numpy as np
>>> x1 = np.arange(0, 3.14, 0.1)
>>> x1
array([ 0. ,  0.1,  0.2,  0.3,  0.4,
       0.5,  0.6,  0.7,  0.8,  0.9,  1. ,
       1.1,  1.2,  1.3,  1.4,  1.5,
       1.6,  1.7,  1.8,  1.9,  2. ,  2.1,
       2.2,  2.3,  2.4,  2.5,  2.6,
       2.7,  2.8,  2.9,  3. ,  3.1])
>>> len(x1)
32
```

Be Careful!!!

This is not “arrange”
but “arange”

Another Way

```
>>> x2 = np.linspace(0,3.14,100)
>>> x2
array([ 0.43434,  0.09515152,  0.15858586,  0.20202,  0.25373737,
       0.31717172,  0.39393939,  0.47979797,  0.56565656,  0.65151515,
       0.73737373,  0.82323232,  0.90909091,  0.99494949,  1.07979798,
       1.16464646,  1.24954545,  1.33444444,  1.41934343,  1.50424242,
       1.58914141,  1.6740404,  1.75893939,  1.84383838,  1.92873737,
       2.01363636,  2.09853535,  2.18343434,  2.26833333,  2.35323232,
       2.43813131,  2.5230303,  2.60792929,  2.69282828,  2.77772727,
       2.86262626,  2.94752525,  3.03242424,  3.11732323,  3.14])
```

>>> len(x2)

100

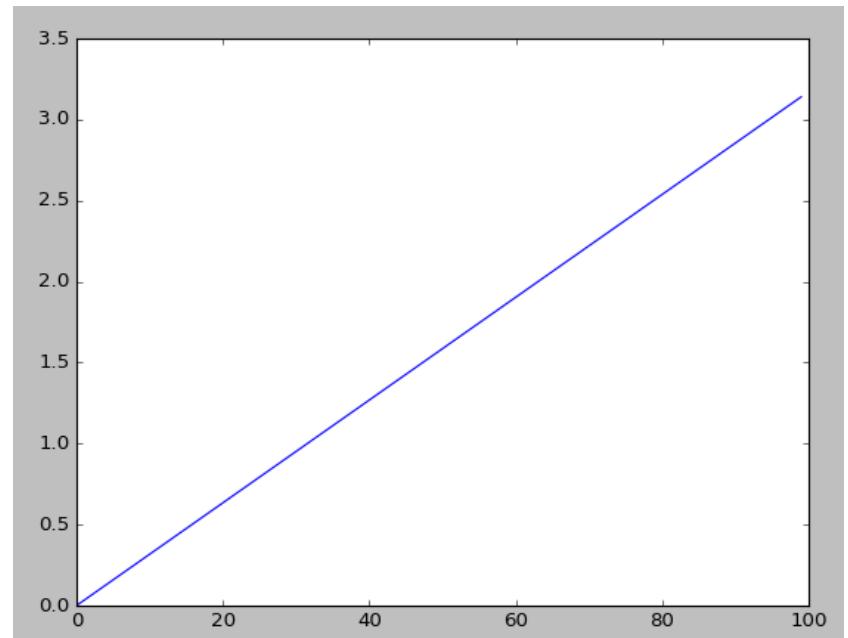
Be Careful!!!

This is not
“linspace” but
“linspace”

Using matplotlib

You can Simply

```
import numpy as np  
import matplotlib.pyplot as plt  
  
x1 = np.linspace(0, 3.14, 100)  
plt.plot(x1)  
  
plt.show()
```



Plotting $\cos(x)$

- Without Numpy, you have to

```
y = [cos(i) for i in x1]
```

- With Numpy, you simply do

```
import numpy as np  
import matplotlib.pyplot as plt
```

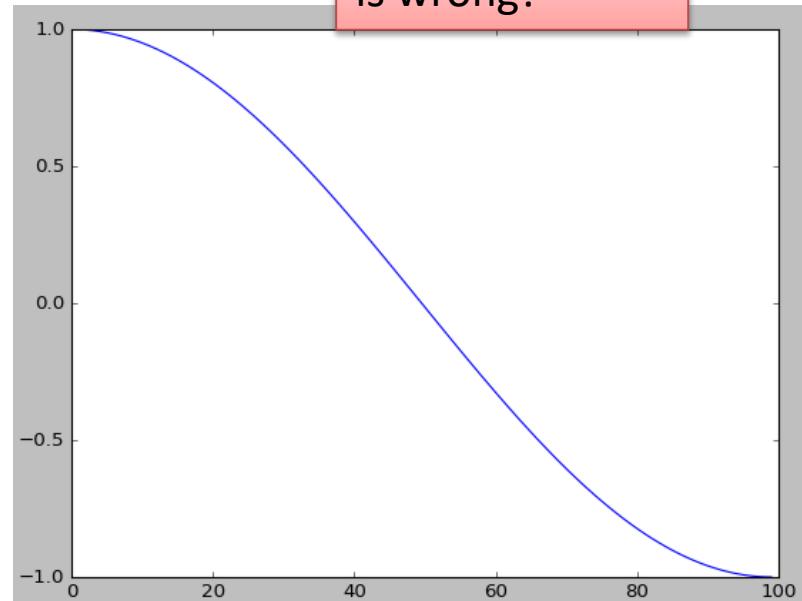
```
x1 = np.linspace(0, 3.14, 100)
```

```
y1 = np.cos(x1)
```

```
plt.plot(y1)
```

```
plt.show()
```

But the x-range
is wrong!



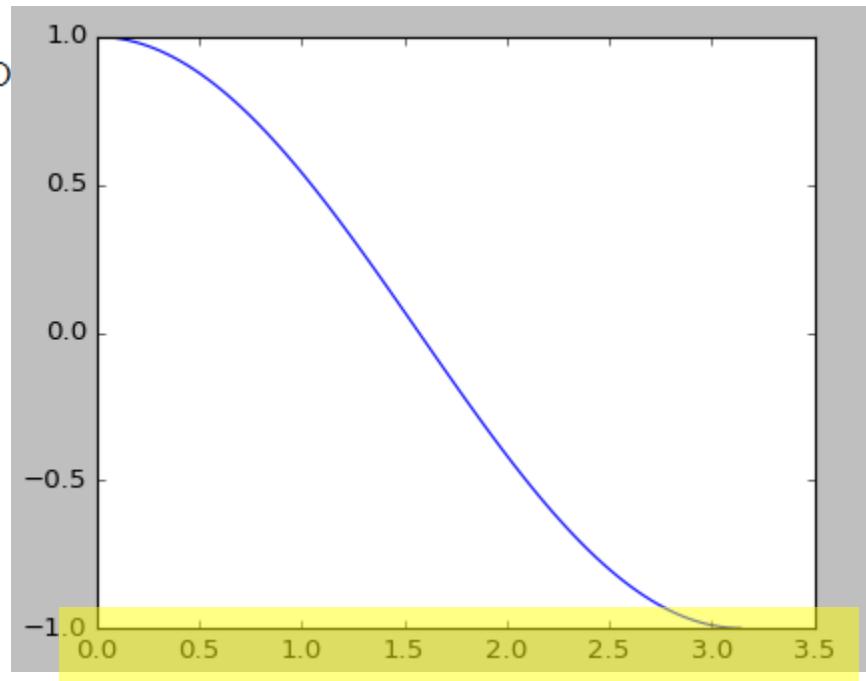
- If you only specify one list (or any sequence), each item will be plotted against just an integer

```
import numpy as np
import matplotlib.pyplot as p

x1 = np.linspace(0, 3.14, 100)
y1 = np.cos(x1)

plt.plot(x1, y1)

plt.show()
```



More Curves (Lab 00)

```
import numpy as np
import matplotlib.pyplot as plt

# Create x, evenly spaced between 0 to 12pi
x = np.linspace(0, 3.14 * 4 , 1000)

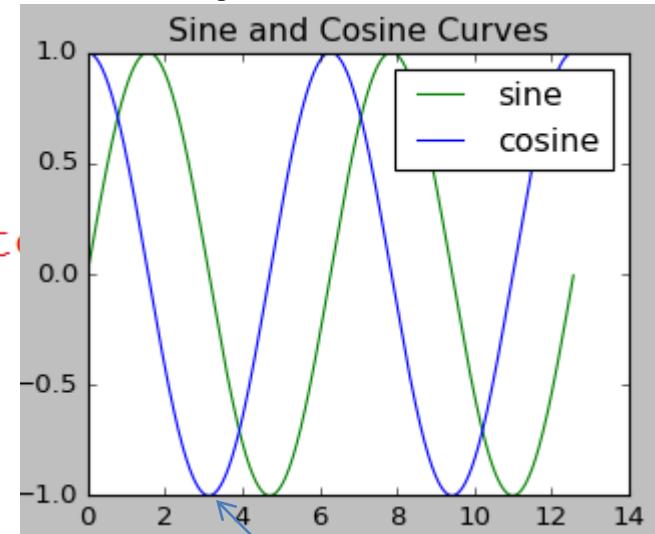
y1 = np.sin(x)
y2 = np.cos(x)

# Plot the sin and cos functions
plt.plot(x , y1, "-g", label="sine")
plt.plot(x , y2, "-b", label="cosine")

# The legend should be in the top right corner
plt.legend(loc="upper right")
plt.title('Sine and Cosine Curves')

plt.show()
```

Green color
and label



Not very nice
looking when the
curve is touching
the boundary

Position
of the
legend

Title

```
import numpy as np
import matplotlib.pyplot as plt

# Create x, evenly spaced from 0 to 14
x = np.linspace(0, 14)

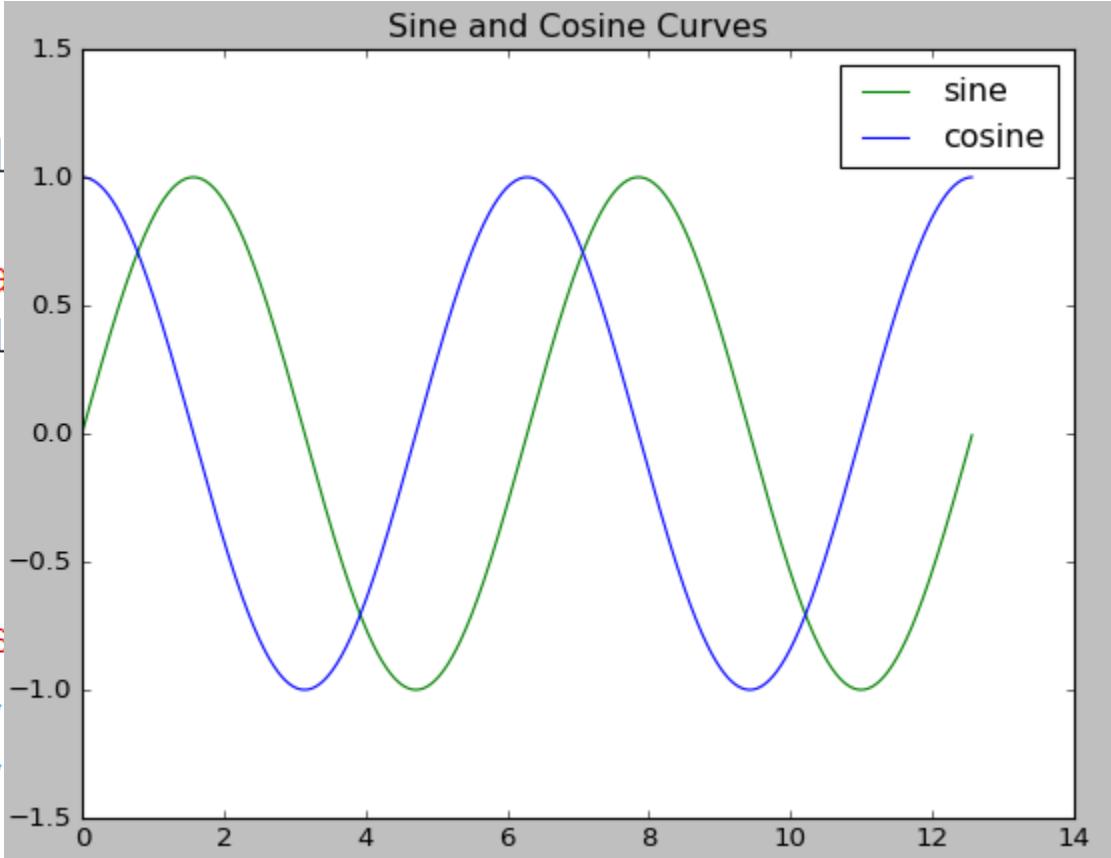
y1 = np.sin(x)
y2 = np.cos(x)

# Plot the sin and cosine curves
plt.plot(x , y1, "-g", label="sine")
plt.plot(x , y2, "-b", label="cosine")

# The legend should be in the top right corner
plt.legend(loc="upper right")
plt.title('Sine and Cosine Curves')

# Limit the y axis to -1.5 to 1.5
plt.ylim(-1.5, 1.5)

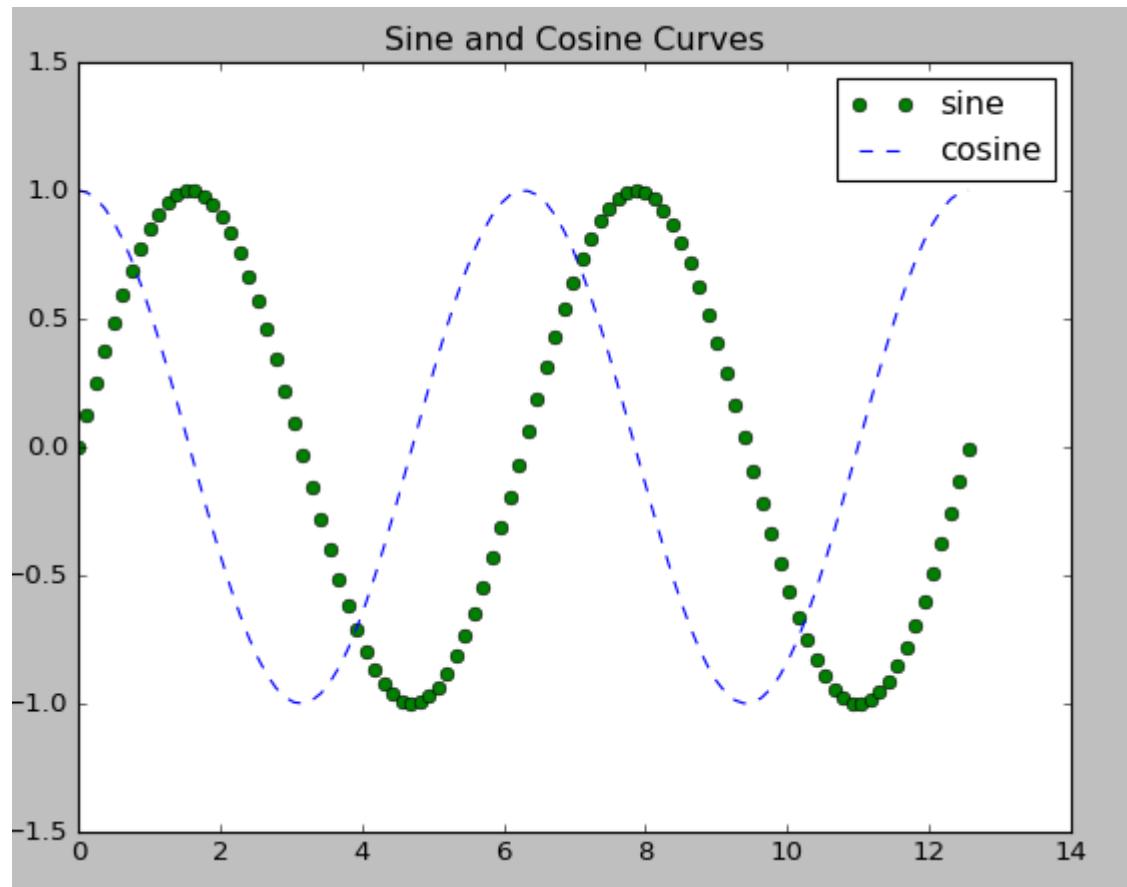
plt.show()
```



Set the vertical limits to be -1.5 to 1.5

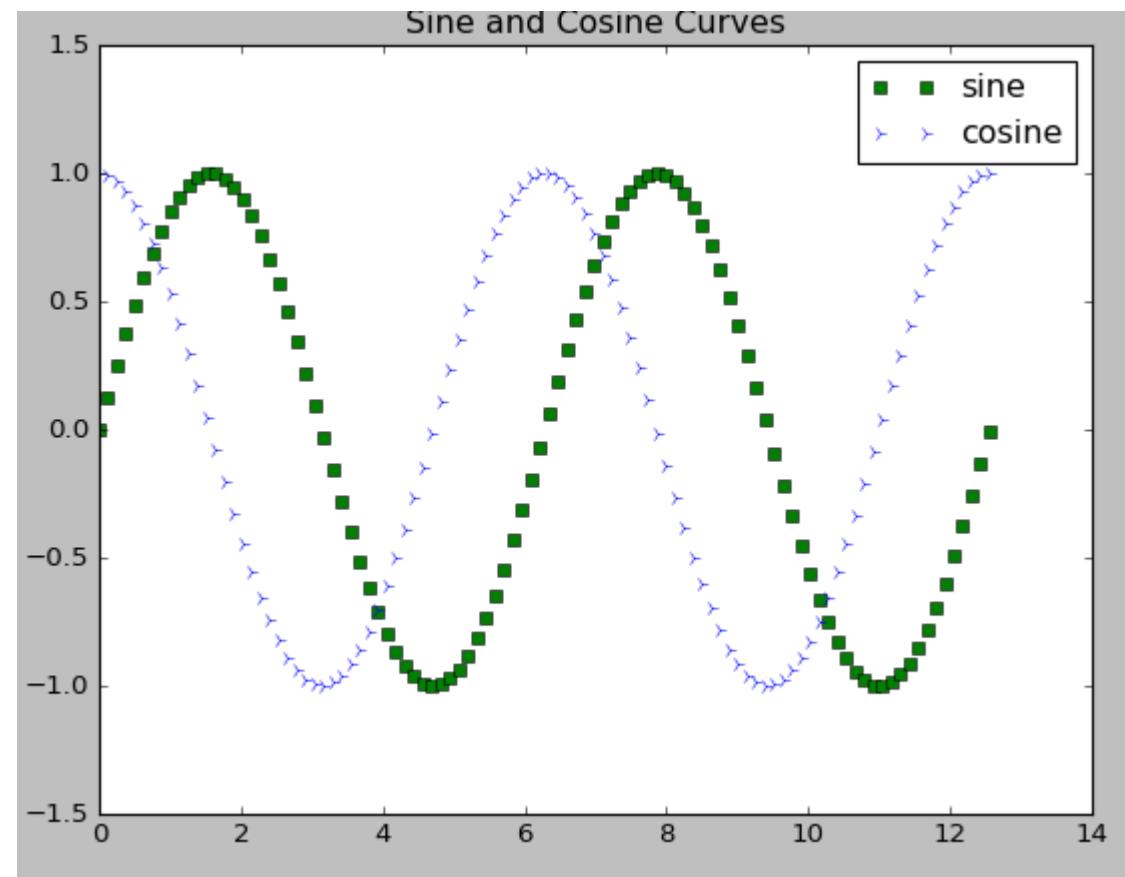
```
# Plot the sin and cos functions
plt.plot(x , y1, "-g", label="sine")
plt.plot(x , y2, "-b", label="cosine")

plt.plot(x , y1, "og", label="sine")
plt.plot(x , y2, "--b", label="cosine")
```



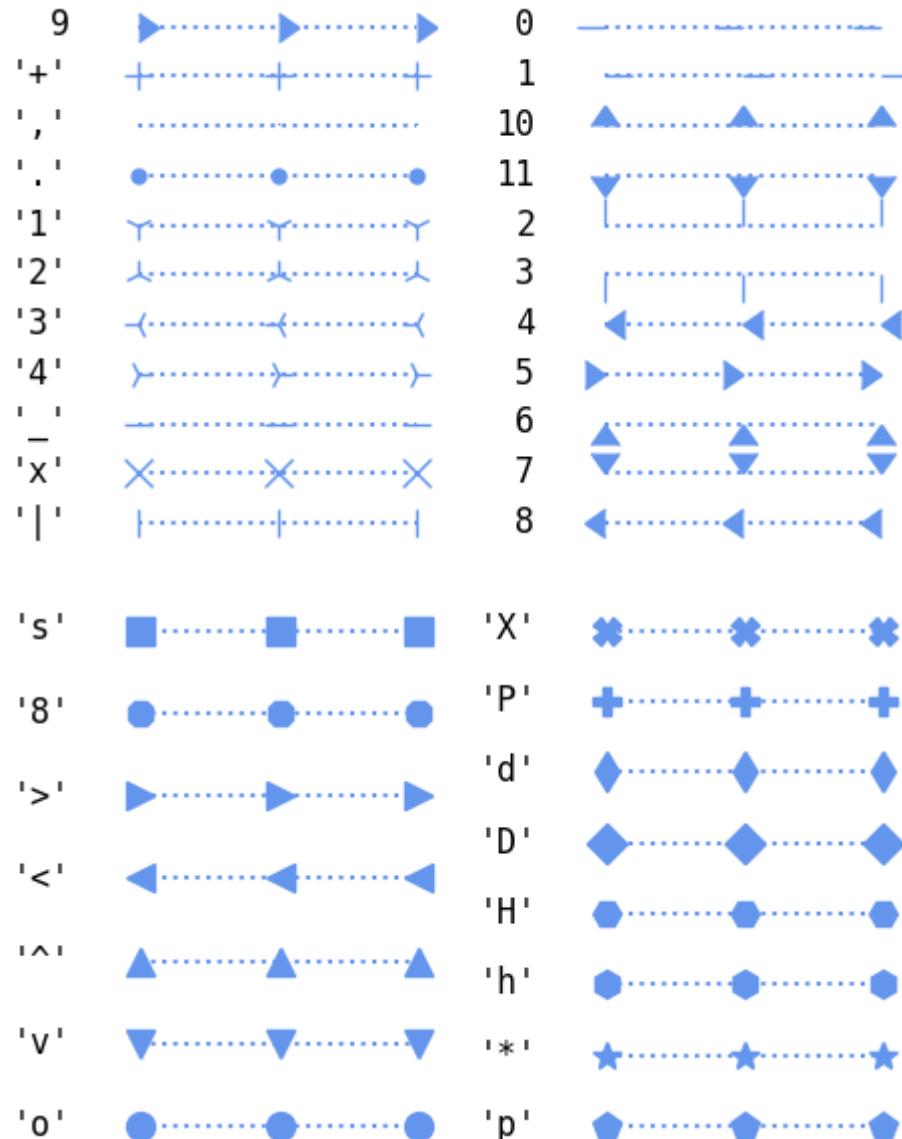
Changing Markers/Line Style

```
# Plot the sin and cos functions  
plt.plot(x , y1, "sg", label="sine")  
plt.plot(x , y2, "4b", label="cosine")
```



Try it out

-- dashed
: dotted
-. dash dotted
- solid
o circle
< triangle_left
+ plus



Details of ALL markers:

https://matplotlib.org/api/markers_api.html

Multiple Figures

```
# Plot the sin and cos functions
```

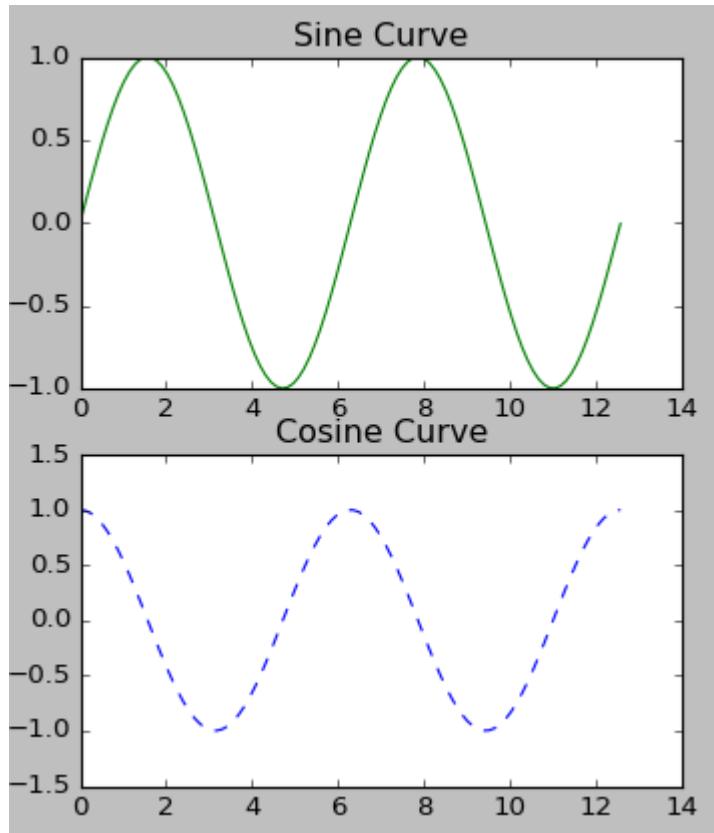
```
plt.subplot(211)
plt.plot(x , y1, "-g")
plt.title('Sine Curve')
plt.subplot(212)
plt.plot(x , y2, "--b")
plt.title('Cosine Curve')
```

```
# Limit the y axis to -1.5 to 1.5
```

```
plt.ylim(-1.5, 1.5)
```

```
plt.show()
```

Only affect
the most
recent graph



Multiple Figures

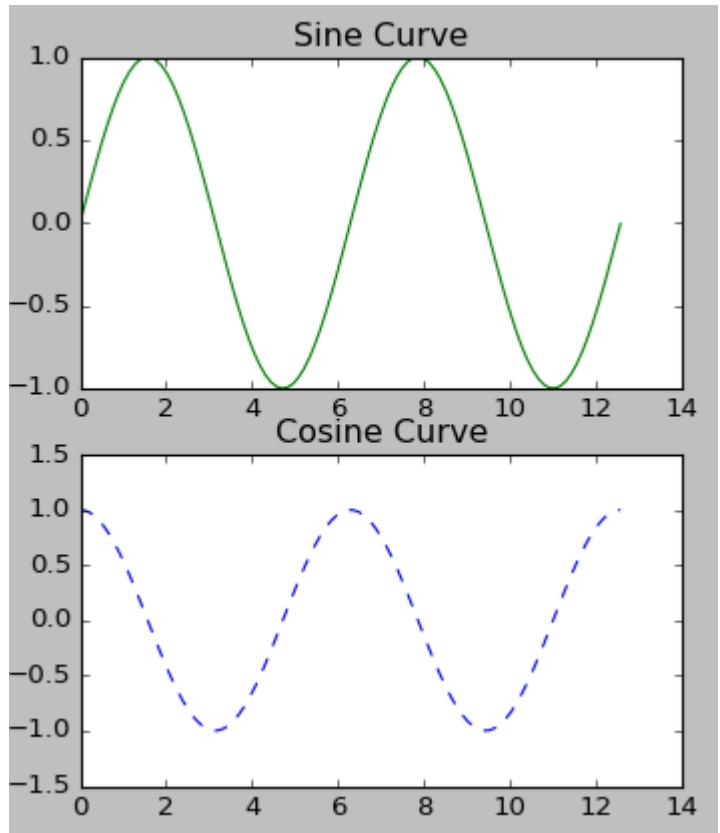
```
# Plot the sin and cos functions
```

```
plt.subplot(211)
```

How many rows r are there

How many columns c are there

After dividing the figure into $r \times c$ places, which one do you want to place the current plotting

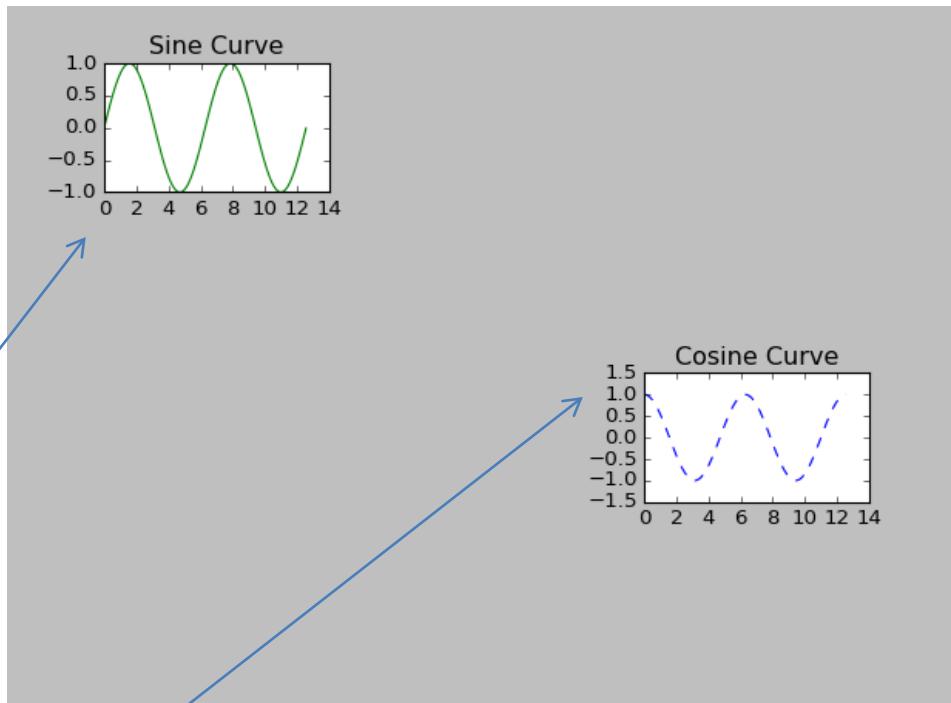


```
# Plot the sin and cos functions  
plt.subplot(431)  
plt.plot(x , y1, "-g")  
plt.title('Sine Curve')  
plt.subplot(439)  
plt.plot(x , y2, "--b")  
plt.title('Cosine Curve')
```

4 rows
3 columns

Position 1

Position 9



If you really have a lot of figures

```
# Plot the sin and cos functions
```

```
plt.figure(1) ←
```

The following plotting will be in Figure 1

```
plt.subplot(431)
```

```
plt.plot(x , y1, "-g")
```

```
plt.title('Sine Curve')
```

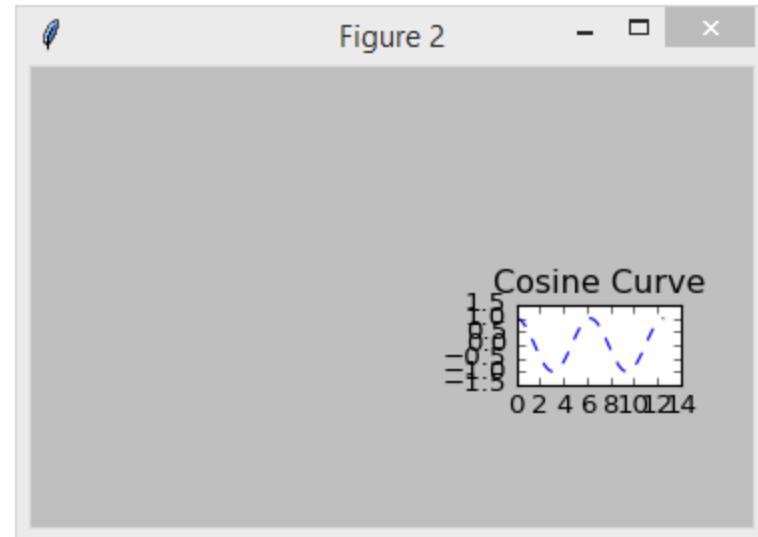
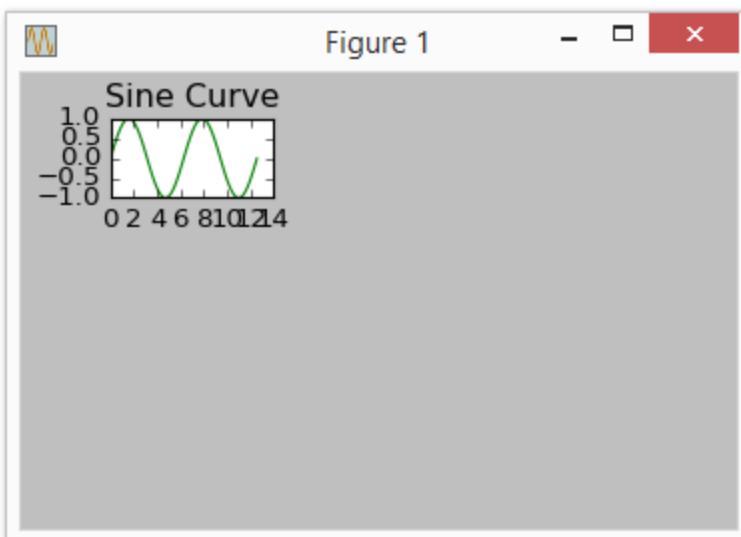
```
plt.figure(2) ←
```

The following plotting will be in Figure 2

```
plt.subplot(439)
```

```
plt.plot(x , y2, "--b")
```

```
plt.title('Cosine Curve')
```



Bar Chart

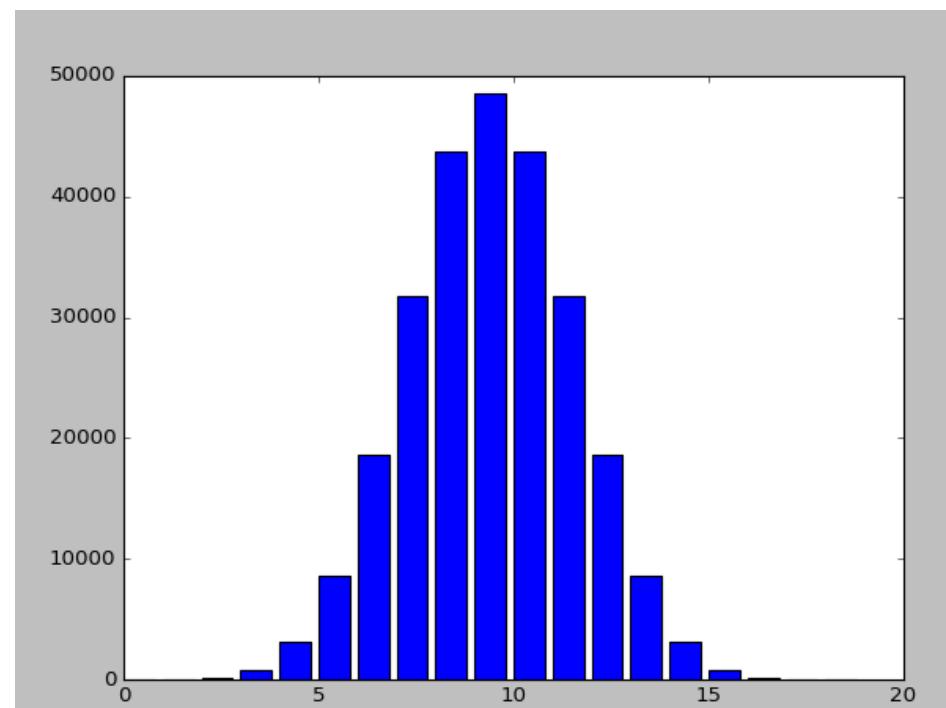
```
from Lab_02_Part_A import nChooseK
```

```
N = 18
```

```
x = [i for i in range(0,N+1)]
```

```
y = [nChooseK(N,i) for i in range(0,N+1)]
```

```
plt.bar(x,y)  
plt.show()
```



Bar Chart

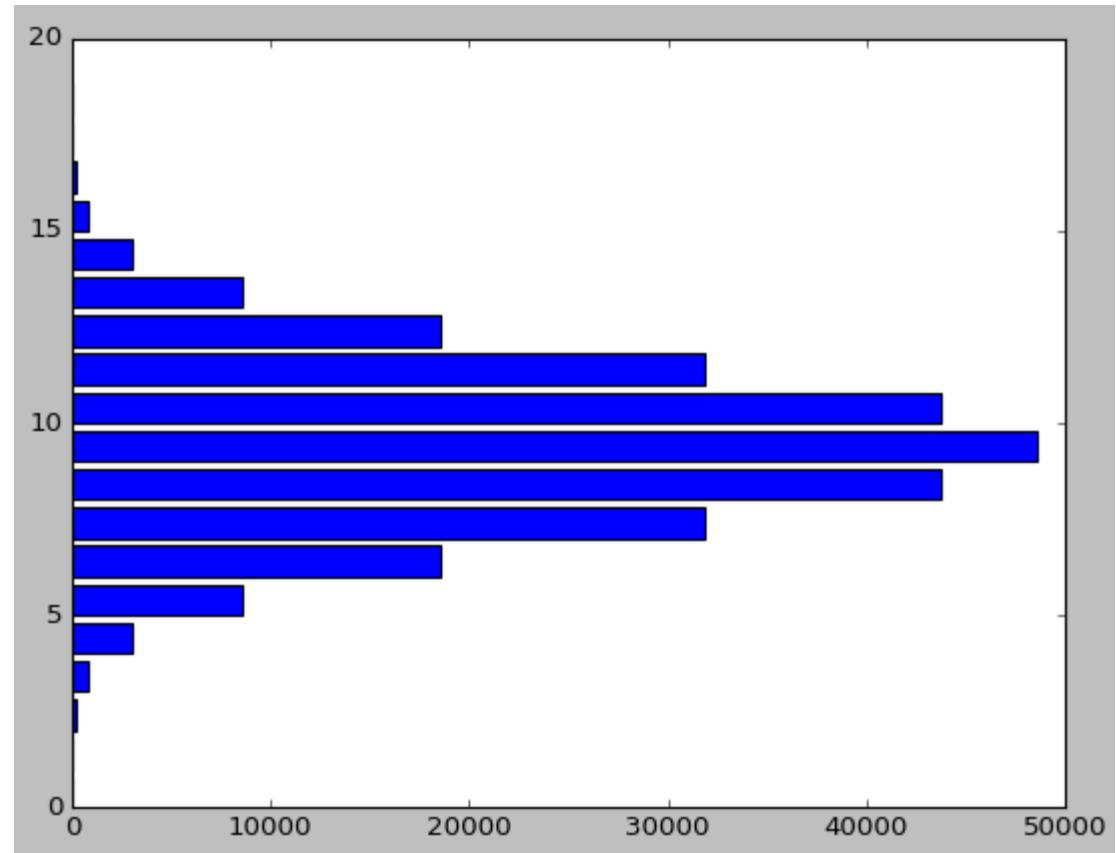
```
N = 18
```

```
x = [i for i in range(0,N+1)]
```

```
y = [nChooseK(N,i) for i in range(0,N+1)]
```

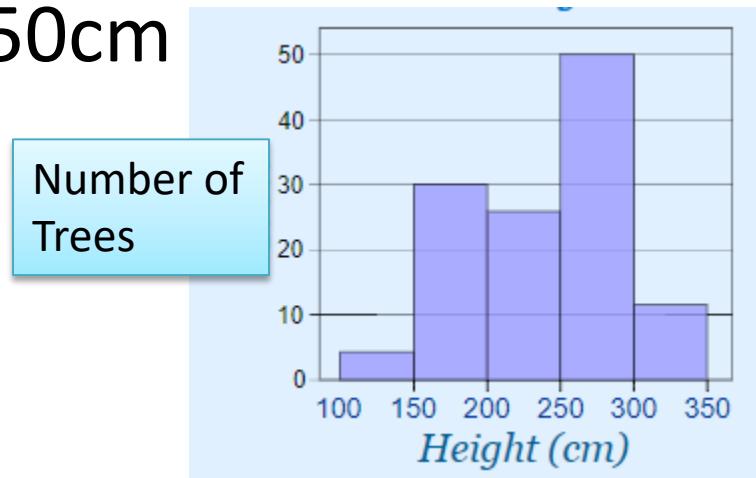
```
plt.barh(x,y)
```

```
plt.show()
```

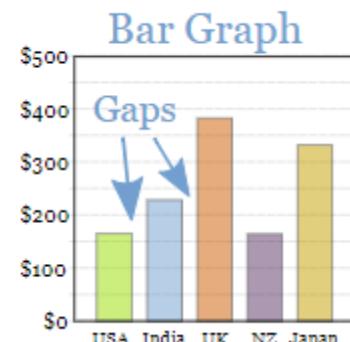


Histogram

- Similar to bar chart, but a histogram groups numbers into ranges
- E.g. Given data: heights of 30 trees from 150cm to 350cm

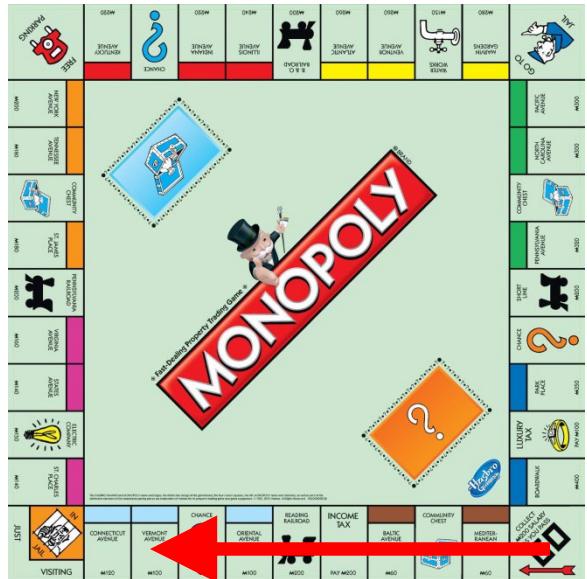


- Usually bar charts have gaps
 - But histograms have no gap



Let's Roll

- When you roll three dices and sum the number, which number has the highest chance?
- Application
 - E.g. in Monopoly, if you start from “GO!”, which place(s) has/have the highest probability to be landed on?



Rolling 3 dices

$$3 = 1 + 1 + 1$$

$$4 = 1 + 1 + 2$$

$$5 = 1 + 1 + 3 = 2 + 2 + 1$$

$$6 = 1 + 1 + 4 = 1 + 2 + 3 = 2 + 2 + 2$$

$$7 = 1 + 1 + 5 = 2 + 2 + 3 = 3 + 3 + 1 = 1 + 2 + 4$$

$$8 = 1 + 1 + 6 = 2 + 3 + 3 = 4 + 3 + 1 = 1 + 2 + 5 = 2 + 2 + 4$$

$$9 = 6 + 2 + 1 = 4 + 3 + 2 = 3 + 3 + 3 = 2 + 2 + 5 = 1 + 3 + 5 = 1 + 4 + 4$$

$$10 = 6 + 3 + 1 = 6 + 2 + 2 = 5 + 3 + 2 = 4 + 4 + 2 = 4 + 3 + 3 = 1 + 4 + 5$$

$$11 = 6 + 4 + 1 = 1 + 5 + 5 = 5 + 4 + 2 = 3 + 3 + 5 = 4 + 3 + 4 = 6 + 3 + 2$$

$$12 = 6 + 5 + 1 = 4 + 3 + 5 = 4 + 4 + 4 = 5 + 2 + 5 = 6 + 4 + 2 = 6 + 3 + 3$$

$$13 = 6 + 6 + 1 = 5 + 4 + 4 = 3 + 4 + 6 = 6 + 5 + 2 = 5 + 5 + 3$$

$$14 = 6 + 6 + 2 = 5 + 5 + 4 = 4 + 4 + 6 = 6 + 5 + 3$$

$$15 = 6 + 6 + 3 = 6 + 5 + 4 = 5 + 5 + 5$$

$$16 = 6 + 6 + 4 = 5 + 5 + 6$$

$$17 = 6 + 6 + 5$$

$$18 = 6 + 6 + 6$$

Study Monopoly



- Rolling three dices, what is the range?
 - 3 to 18
 - Totally 16 different combinations
- All combinations have the same probability?
 - NO!
- Then what are the probabilities?
 - We can do calculations
 - But I forgot all my math already!?!?!

Rolling 3 dices

Probability of a sum of 3: $1/216 = 0.5\%$

Probability of a sum of 4: $3/216 = 1.4\%$

Probability of a sum of 5: $6/216 = 2.8\%$

Probability of a sum of 6: $10/216 = 4.6\%$

Probability of a sum of 7: $15/216 = 7.0\%$

Probability of a sum of 8: $21/216 = 9.7\%$

Probability of a sum of 9: $25/216 = 11.6\%$

Probability of a sum of 10: $27/216 = 12.5\%$

Probability of a sum of 11: $27/216 = 12.5\%$

Probability of a sum of 12: $25/216 = 11.6\%$

Probability of a sum of 13: $21/216 = 9.7\%$

Probability of a sum of 14: $15/216 = 7.0\%$

Probability of a sum of 15: $10/216 = 4.6\%$

Probability of a sum of 16: $6/216 = 2.8\%$

Probability of a sum of 17: $3/216 = 1.4\%$

Probability of a sum of 18: $1/216 = 0.5\%$

Let's run an experiment

```
N = 100000  
  
def roll_dice():  
    return random.randint(1, 6)  
  
dice_stat = []  
for i in range(N):  
    dice_stat.append(roll_dice() + roll_dice() + roll_dice())
```

why not
roll_dice()*3?



- `dics_stat` will contain 100000 numbers of the sum of the dices
- Let's say, how many “18” are there in these 100000 numbers?

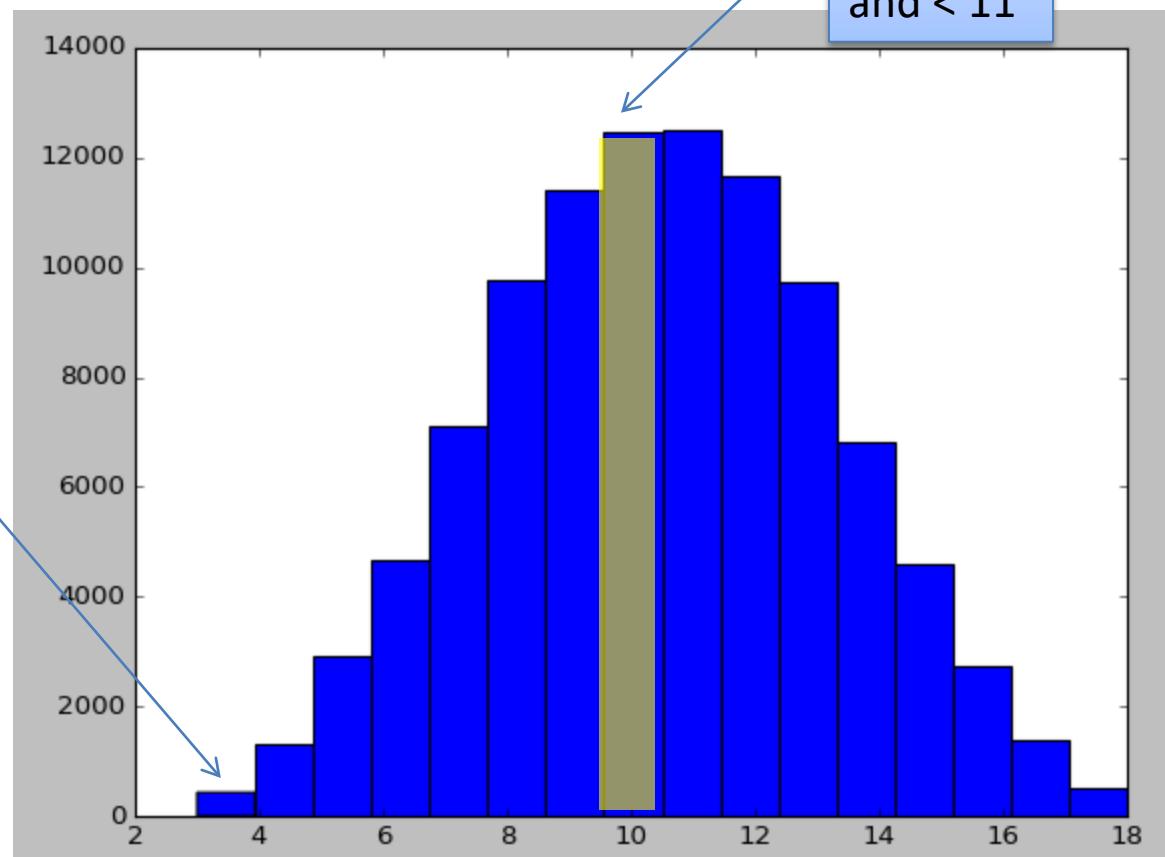
Histogram

```
plt.hist(dice_stat, 16)  
plt.show()
```

Number
of bins

Number
of times
of ≥ 10
and < 11

Number
of times
of ≥ 3
and < 4



Histogram

- Usually, histograms won't have a bin for every single number x
 - $3 \leq x < 4$
- For example, if the data is the salary, every bin will have a larger range
 - $1000 \leq x < 2000$
 - $2000 \leq x < 3000$
 - etc.

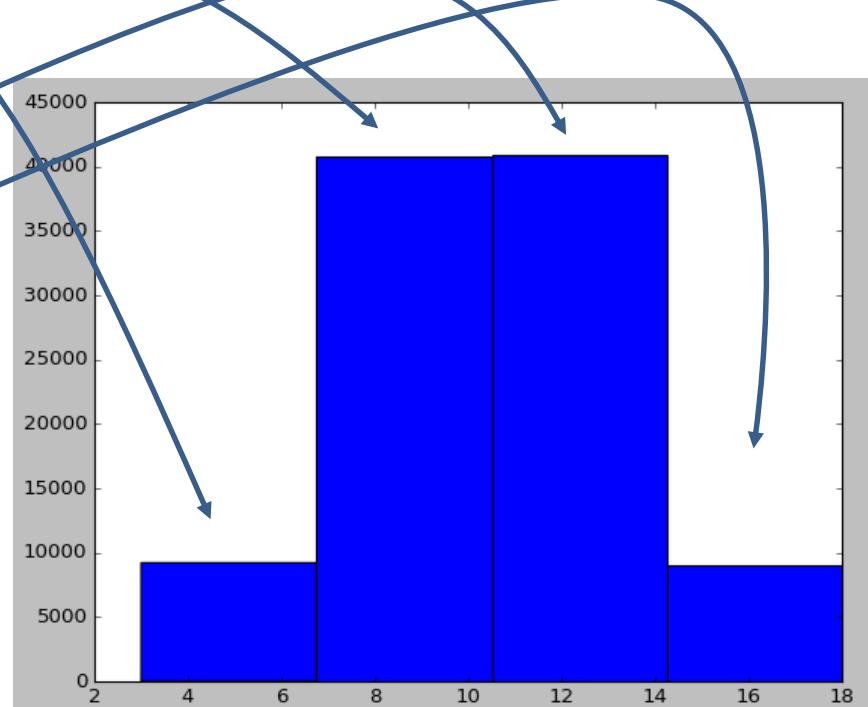
Histogram

- Back to the dice example, say we want the ranges:

- $3 \leq x < 7$
 - $7 \leq x < 11$
 - $11 \leq x < 15$
 - $15 \leq x < 19$

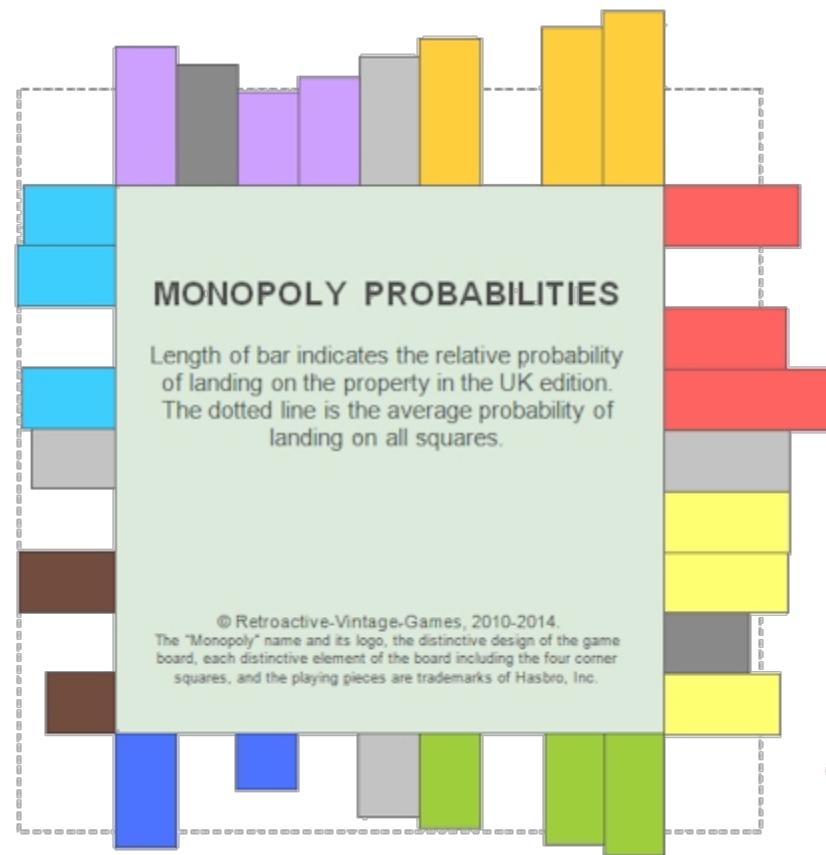
- Then we need **4** bins

```
plt.hist(dice_stat, 4)  
plt.show()
```



Application: Monopoly

- Is every place has a equal chance to be landed on?
 - NO!



Pie Chart

```
import matplotlib.pyplot as plt
```

```
labels = ['Python', 'Java', 'C++', 'C#', 'C', 'JavaScript', 'I  
sizes = [26.7, 22.6, 9.9, 9.4, 7.37, 6.9, 5.9, 11.23]
```

```
plt.pie(sizes, shadow=True, startangle=90)  
plt.legend(labels, loc="best")
```

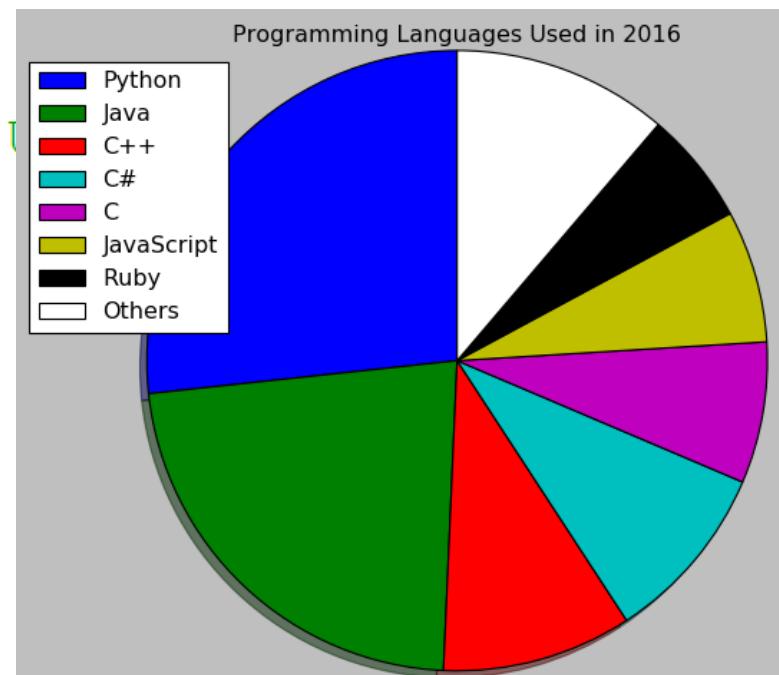
```
plt.axis('equal')
```

```
plt.title('Programming Languages Used in 2016')
```

```
plt.tight_layout()
```

```
plt.show()
```

Otherwise, becomes “oval” chart



Saving a Graph

- If you feel like your graph is cool and want to save it by
 - `fig.savefig('plot.png')`, or
 - `fig.savefig('plot.pdf')`, or
 - Any format that is supported by `matplotlib`

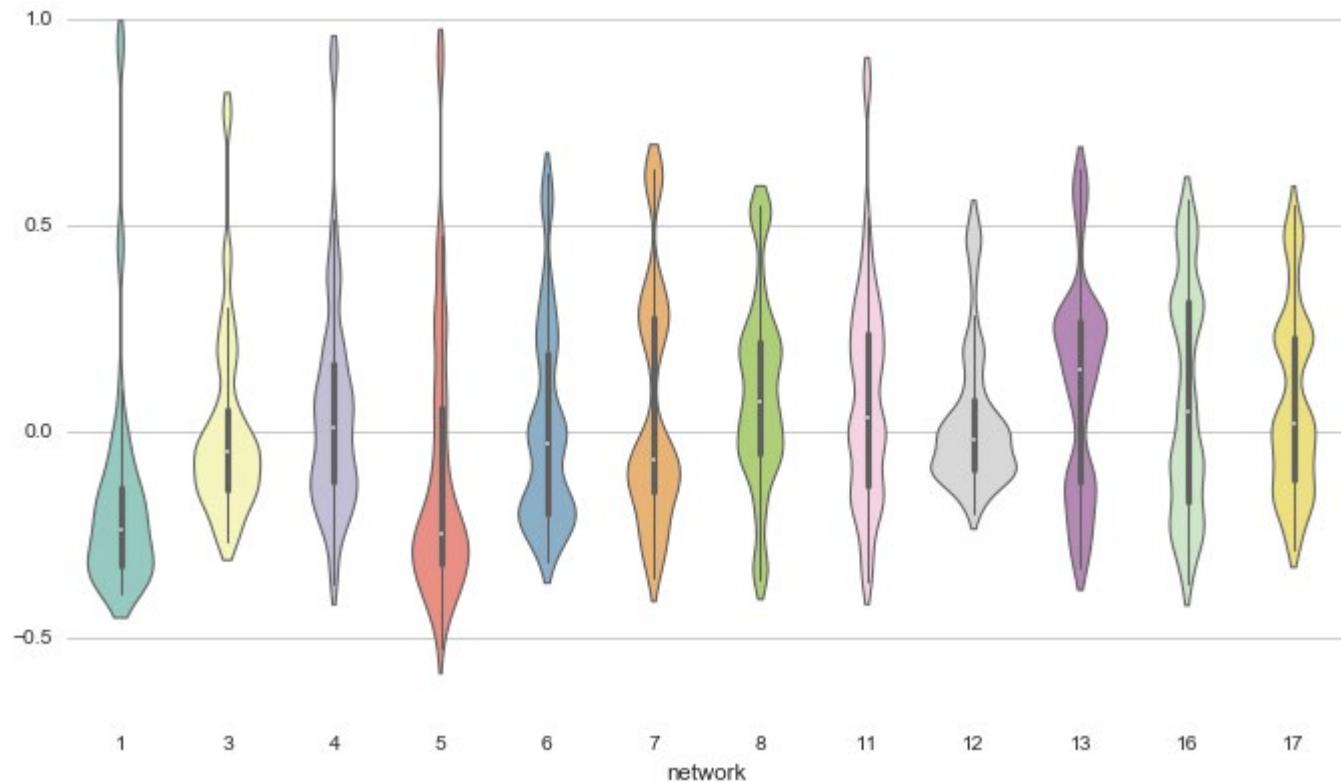
Supported Graphic Format

- You can check the supported file format by
`plt.gcf().canvas.get_supported_filetypes()`

```
{'ps': 'Postscript', 'eps': 'Encapsulated Postscript', 'pdf': 'Portable Document Format', 'pgf': 'PGF code for LaTeX', 'png': 'Portable Network Graphics', 'raw': 'Raw RGBA bitmap', 'rgba': 'Raw RGBA bitmap', 'svg': 'Scalable Vector Graphics', 'svgz': 'Scalable Vector Graphics', 'jpg': 'Joint Photographic Experts Group', 'jpeg': 'Joint Photographic Experts Group', 'tif': 'Tagged Image File Format', 'tiff': 'Tagged Image File Format'}
```

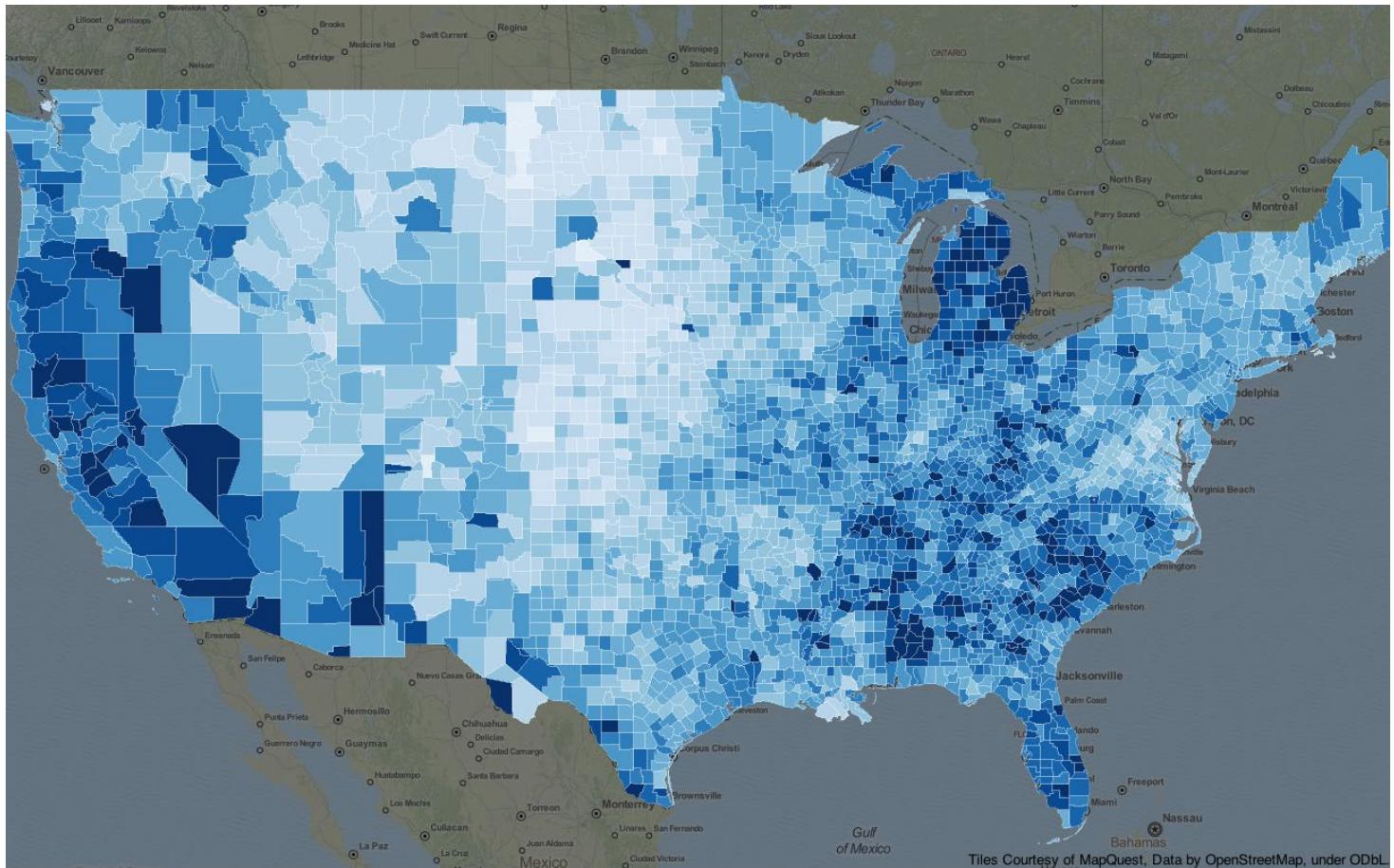
Other Visualization Packages

- Seaborn



Other Visualization Packages

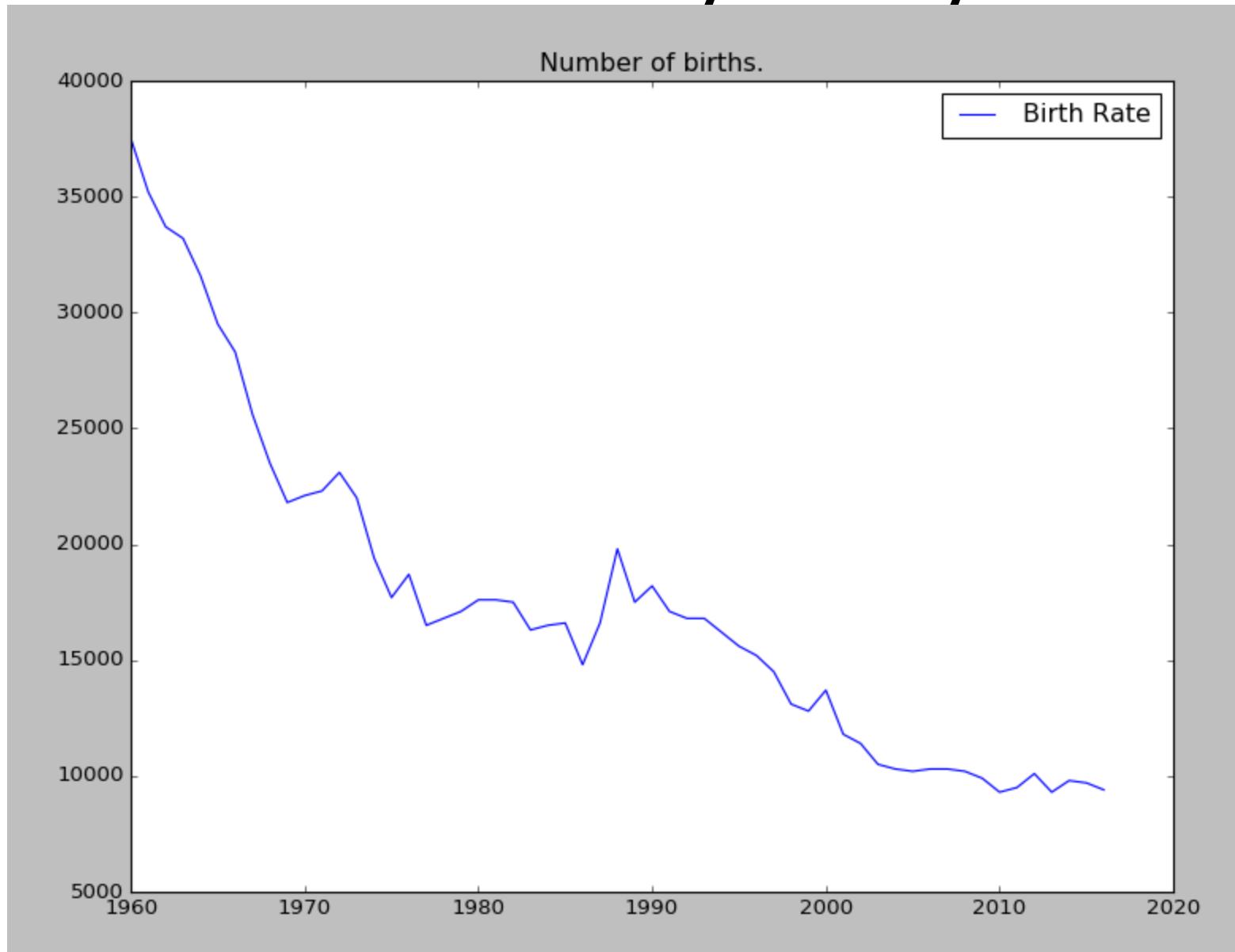
- geoplotlib



Some Tips on GOOD Visualization

- Take a step back and ask yourself, “What is my point?”

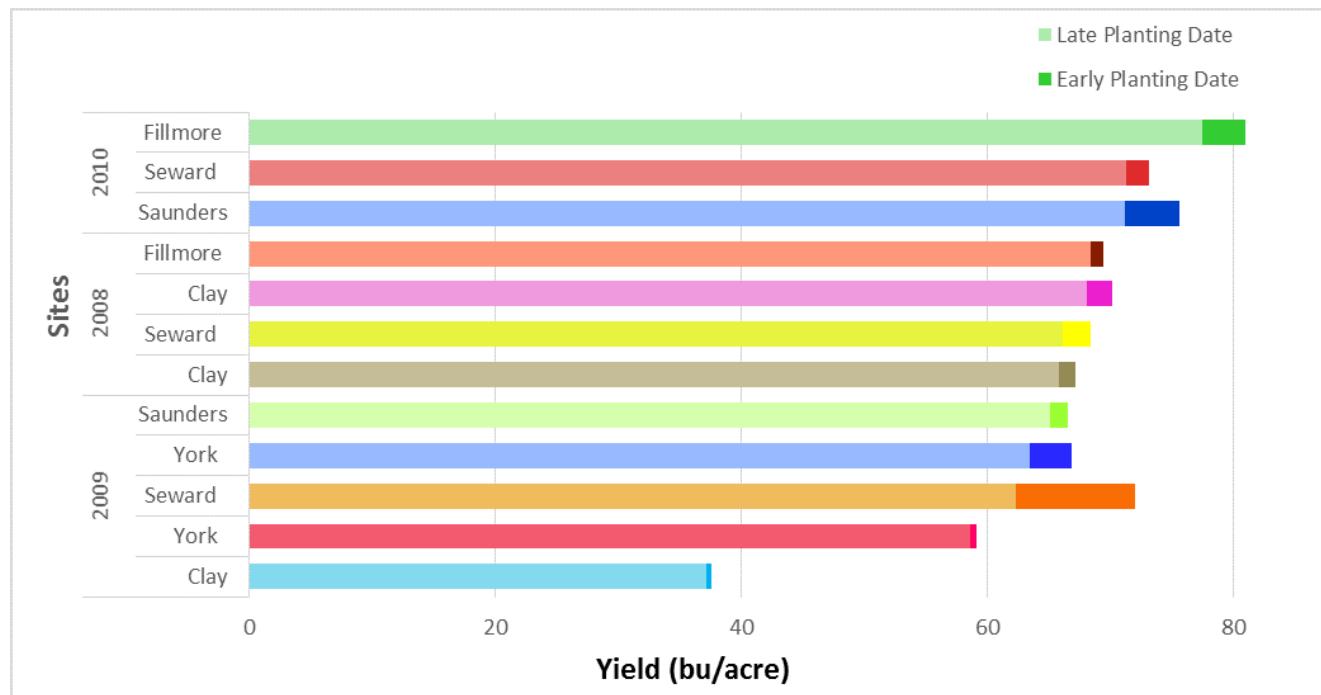
Now You Know Why “Baby Bonus”



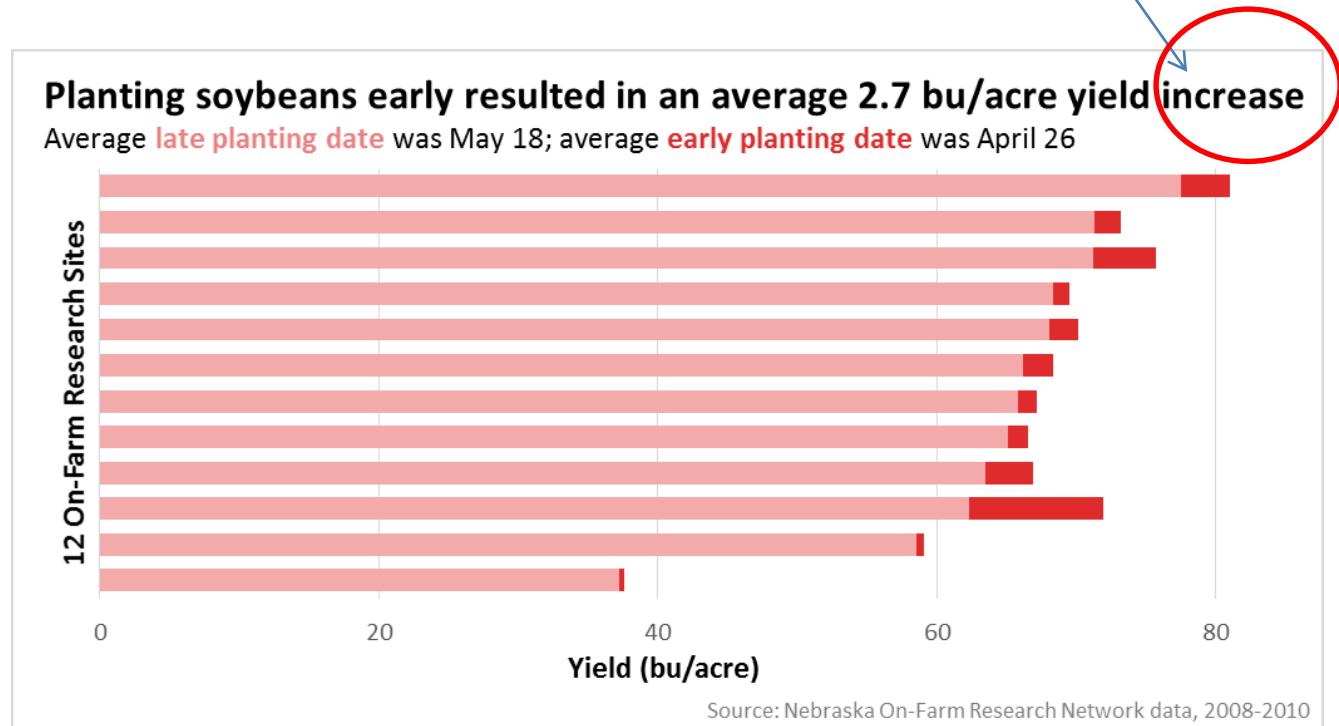
Some Tips on GOOD Visualization

- Take a step back and ask yourself, “What is my point?”
- Choose the right chart
- Use color intentionally

- The graph is bright and eye catching, yet the color is not used in a meaningful way.
- Separating the various sites with different colors is not important and only detracts from the overall point.



- Using color to make the key point stand out. In the following graph
- Use a lighter shade of red for the late planting date, and a darker shade of red for the early planting date.



Some Tips on GOOD Visualization

- Take a step back and ask yourself, “What is my point?”
- Choose the right chart
- Use color intentionally
- Create pointed titles and call out key points with text
- Get feedback and iterate
- Read up and copy other visualizations

More Help on Matplotlib

- https://matplotlib.org/users/pyplot_tutorial.html
- How to FAQ
 - https://matplotlib.org/faq/howto_faq.html





NUS | Computing

National University
of Singapore

IT5001 Software Development Fundamentals

16. Miscellaneous

Agenda

- Decorators for Memoization
- Graph Problems
 - Route Problems
 - Shortest Distance

Decorators

- Decorator is a closure
 - Additionally, outer function accepts a function as input argument
- Modify input function's behaviour with an inner function without explicitly changing input function's code
- Example

```
def deco(func):  
    def wrapper():  
        #statements  
        func()  
        pass  
    return wrapper  
  
def f():  
    pass  
  
f = deco(f)
```

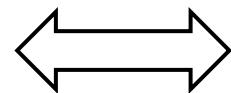
```
def deco(func):  
    def wrapper():  
        #statements  
        func()  
        pass  
    return wrapper  
  
@deco  
def f():  
    pass
```

Decorators - Example

```
def my_decorator(func):  
    def wrapper():  
        print('Hello')  
        func()  
        print('Welcome')  
    return wrapper
```

```
def func():  
    print('IT5001')
```

```
decorated_func = my_decorator(func)  
decorated_func()
```



```
def my_decorator(func):  
    def wrapper():  
        print('Hello')  
        func()  
        print('Welcome')  
    return wrapper
```

```
@my_decorator  
def func():  
    print('IT5001')
```

```
func()
```

Decorators: Example

- Decorating functions that has arguments

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        #statements
        func(*args, **kwargs)
    return wrapper
```

```
@my_decorator
def f():
    pass
```

Decorators: Example

- Decorating functions that has arguments

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print('executing decorated function')
        return func(*args, **kwargs)
    return wrapper
```

```
@my_decorator
def g(x, y = None):
    return x**2+y**2
```

Decorators: Applications

- Profiling
 - For timing functions
- Logging
 - For debugging
- Caching
 - For Memoization

Decorators for Caching

```
def cache(func) :  
    memo = {}  
    def wrap (*args):  
        if args not in memo:  
            memo[args] = func(*args)  
        return memo[args]  
    return wrap
```

Fibonacci using decorators

```
def fibm(n):
    if n in fibans.keys():
        return fibans[n]

    if (n == 0):
        ans = 0
    elif (n == 1):
        ans = 1
    else:
        ans = fibm(n-1)+fibm(n-2)
    return ans
print(fibm(10))
```



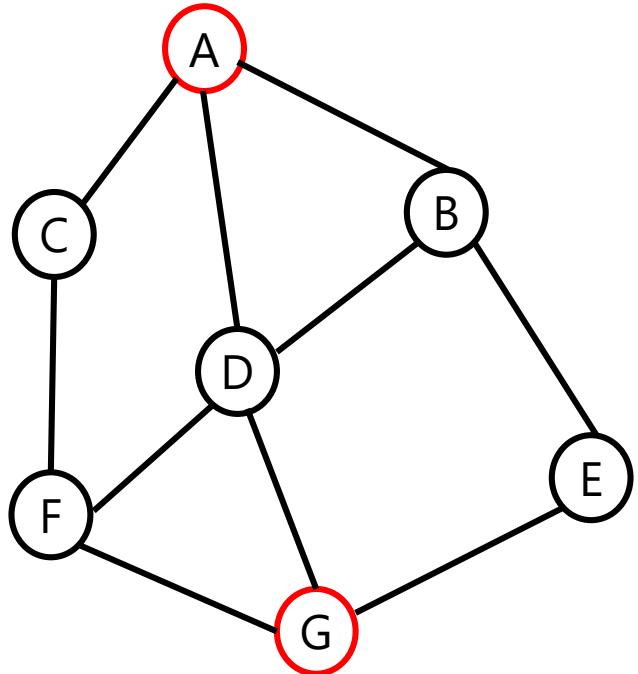
```
@cache
def fib(n):
    if n ==0:
        return 0
    if n <= 2:
        return 1
    return fib(n-1) + fib(n-2)

print(fib(50))
```

Graphs

Path Between Vertices

Breadth-First Search

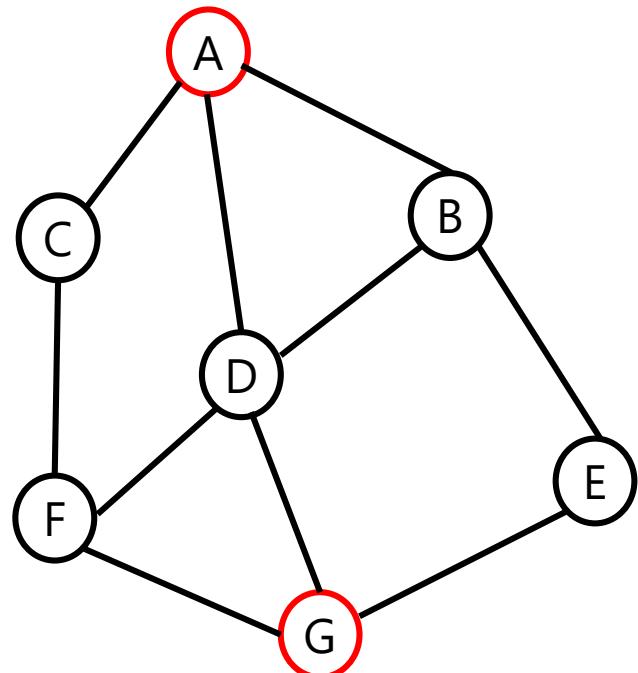


Objective:

Checking if a path exists from vertex A to vertex G

Graph Representation

- Consists of two components
 - Vertices
 - A,B,C,D, E,
 - Edges
 - (A,B), (A,C), (B,D)....
- How to represent it?
 - Edge List
 - Contains list of all edges
 - Adjacency List/Dictionary
 - List of vertices that are adjacent to a given vertex
 - Can use dictionary
 - Provides mapping between each vertex and its neighbors
- Assume we are given list of edges, i.e., edge list



Edge List to Adjacency List

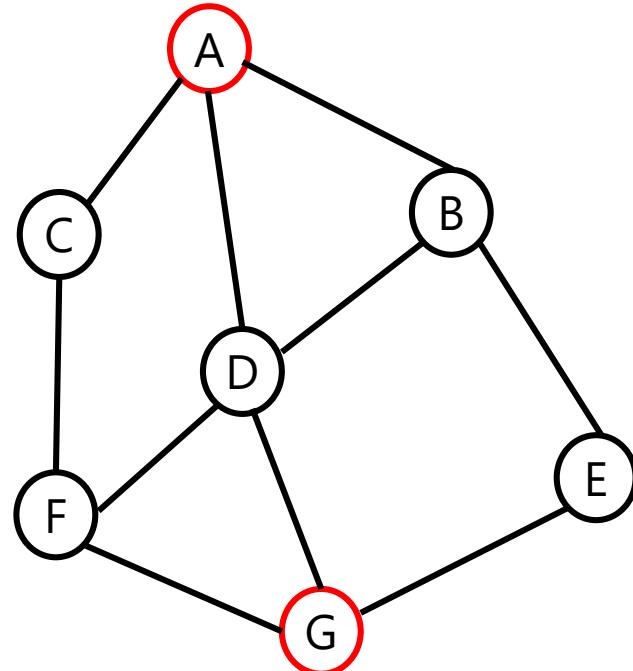
```
def edgeList_to_adjList(edgeList) :  
    adjacencyList = {}  
    for a, b in edgeList:  
        if a not in adjacencyList:  
            adjacencyList[a] = []  
        if b not in adjacencyList:  
            adjacencyList[b] = []  
        adjacencyList[a].append(b)  
        adjacencyList[b].append(a)  
    return adjacencyList
```

Breadth-First Search

```
def can_travel_bfs(edgeList, source, destination):
    adjacencyList = edgeList_to_adjList(edgeList)
    visited = set()
    frontier = [source]
    while frontier:
        current = frontier.pop(0)
        if current == destination:
            return True
        if current not in adjacencyList or current in visited:
            continue
        visited.add(current)
        frontier.extend(adjacencyList[current])
    return False

print(can_travel_bfs(edge_list, 'A', 'C'))
```

Breadth-First Search



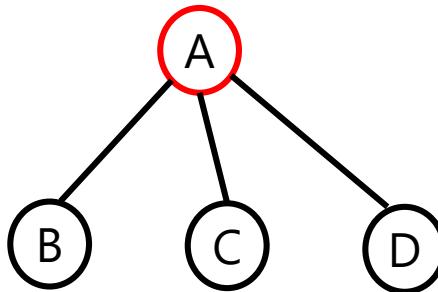
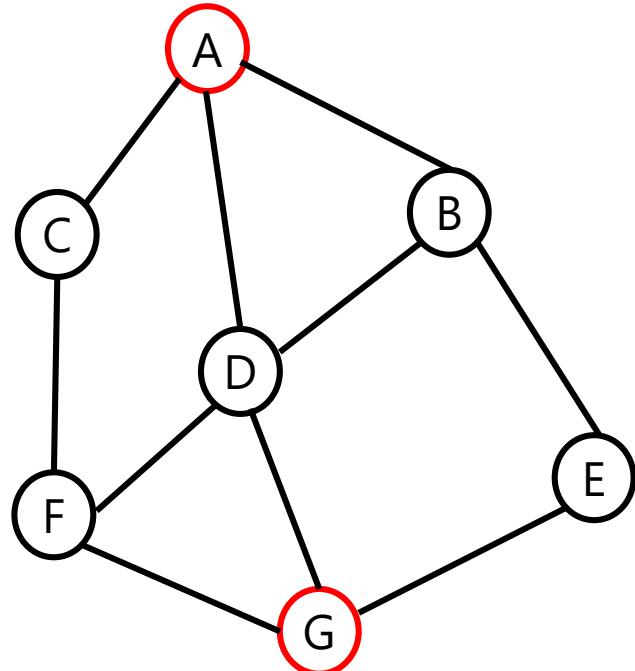
A

current \neq destination
current *not in* visited

```
visited = {}  
frontier = ['A']
```

```
print(can_travel_bfs(edge_list, 'A', 'C'))
```

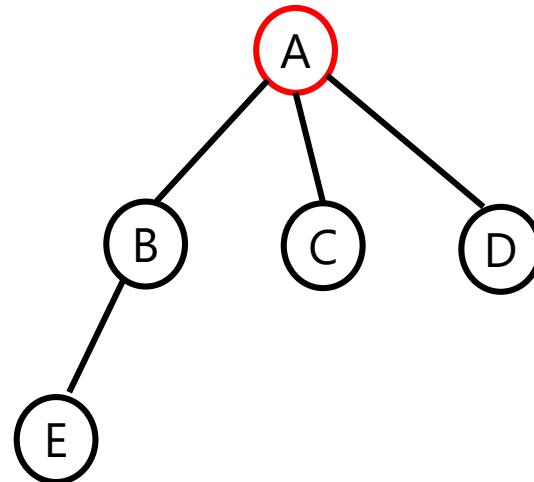
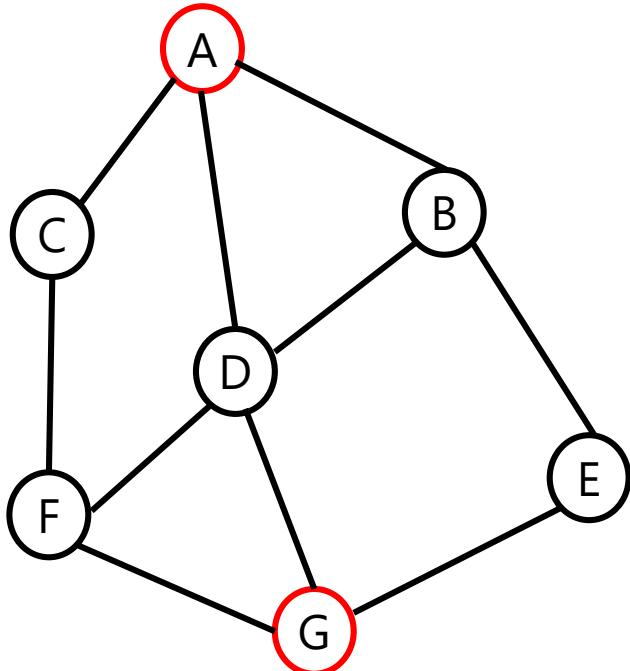
Breadth-First Search



current \neq destination
current *not in* visited

visited = {'A'}
frontier = ['B', 'C', 'D']

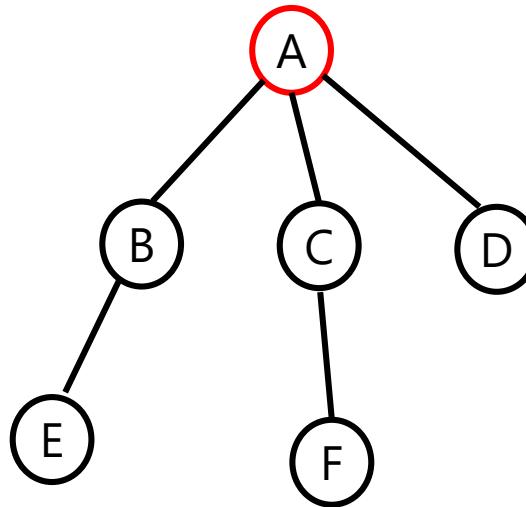
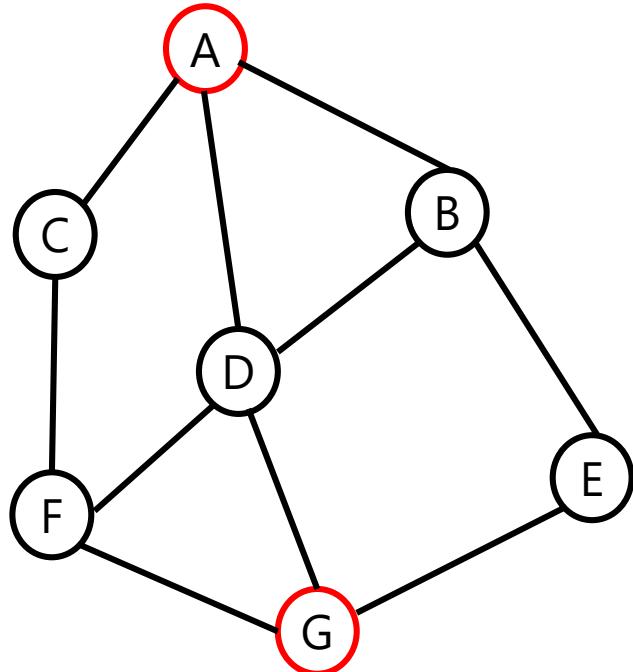
Breadth-First Search



current \neq destination
current *not in* visited

visited = {'A', 'B'}
frontier = ['C', 'D', 'E']

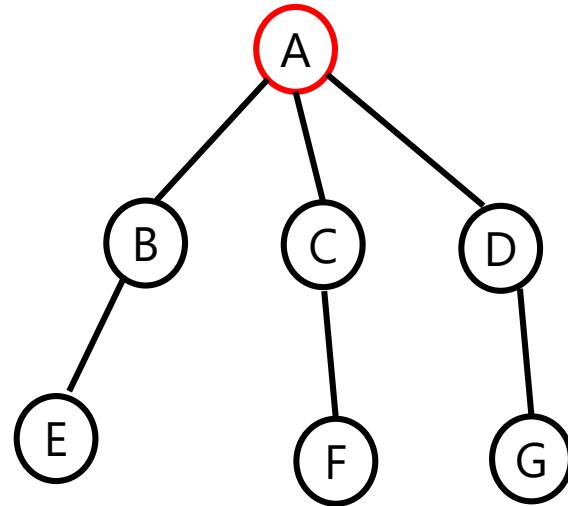
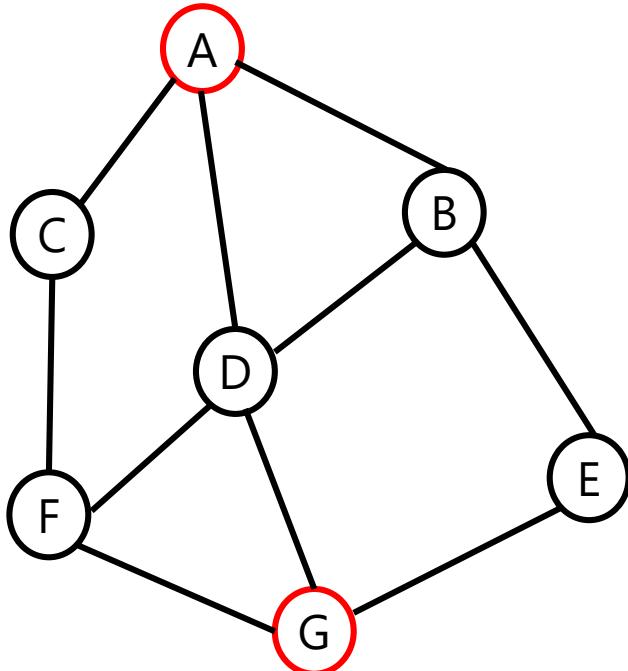
Breadth-First Search



current \neq destination
current *not in* visited

visited = {'A', 'B', 'C'}
frontier = ['D', 'E', 'F']

Breadth-First Search

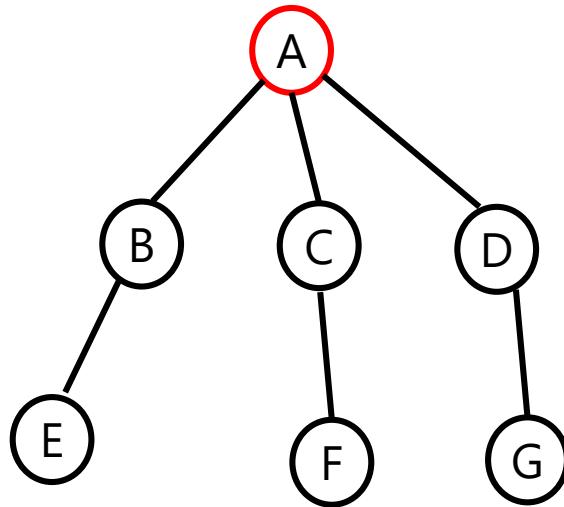
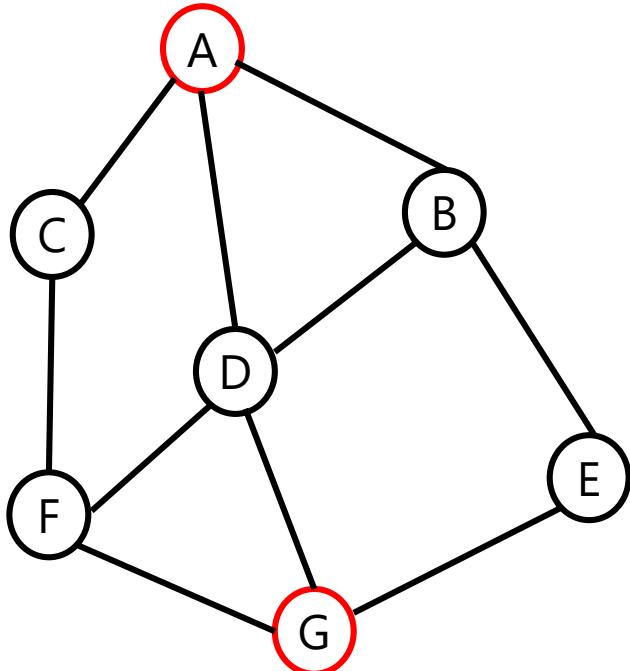


current *not* in visited

current \neq destination

visited = {'A', 'B', 'C', 'D'}
frontier = ['E', 'F', 'G']

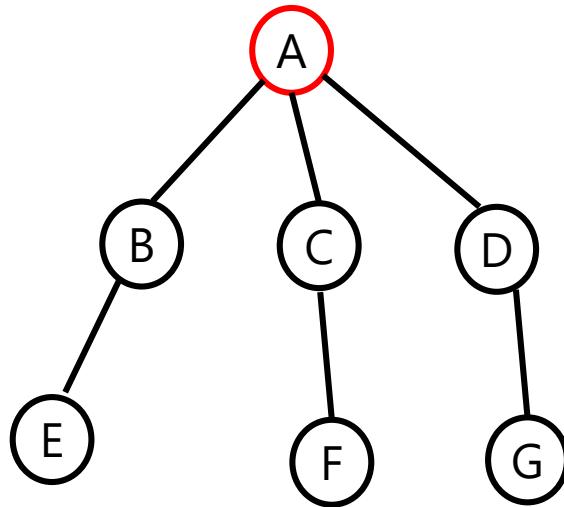
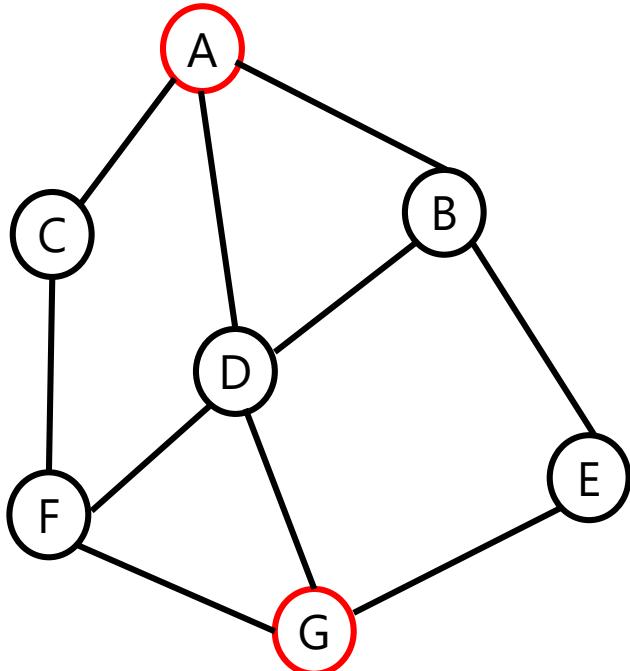
Breadth-First Search



current \neq destination
current *not in* visited

visited = {'A', 'B', 'C', 'D', 'E'}
frontier = ['F', 'G']

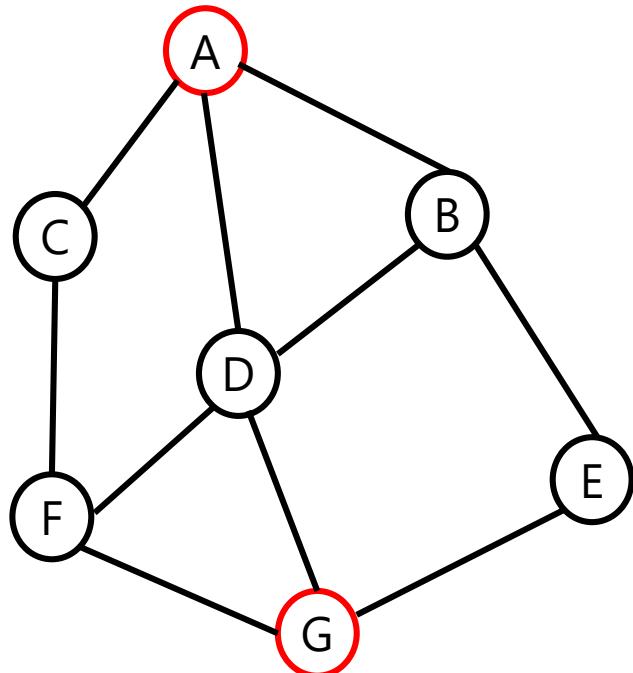
Breadth-First Search



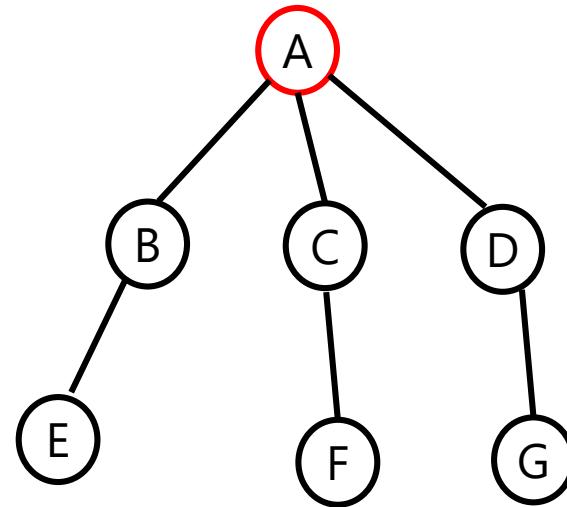
current \neq destination
current *not in* visited

visited = {'A', 'B', 'C', 'D', 'E'}
frontier = ['G']

Breadth-First Search



```
visited = {'A', 'B', 'C', 'D', 'E'}  
frontier = []
```



current == destination

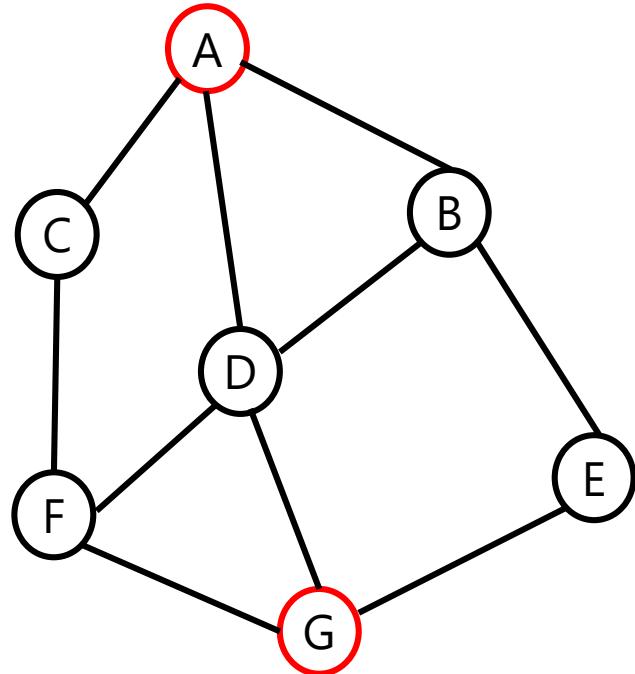
Returns *True*

Depth-First Search

```
def can_travel_dfs(edgeList, source, destination):
    adjacencyList = edgeList_to_adjList(edgeList)
    visited = set()
    frontier = [source]
    while frontier:
        current = frontier.pop()
        if current == destination:
            return True
        if current not in adjacencyList or current in visited:
            continue
        visited.add(current)
        frontier.extend(adjacencyList[current])
    return False

print(can_travel_dfs(edge_list, 'A', 'C'))
```

Depth-First Search

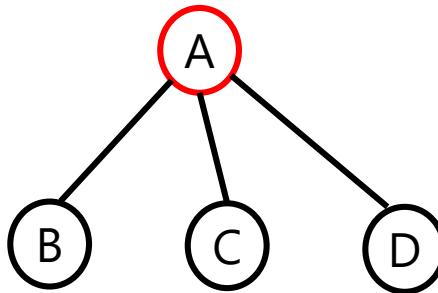
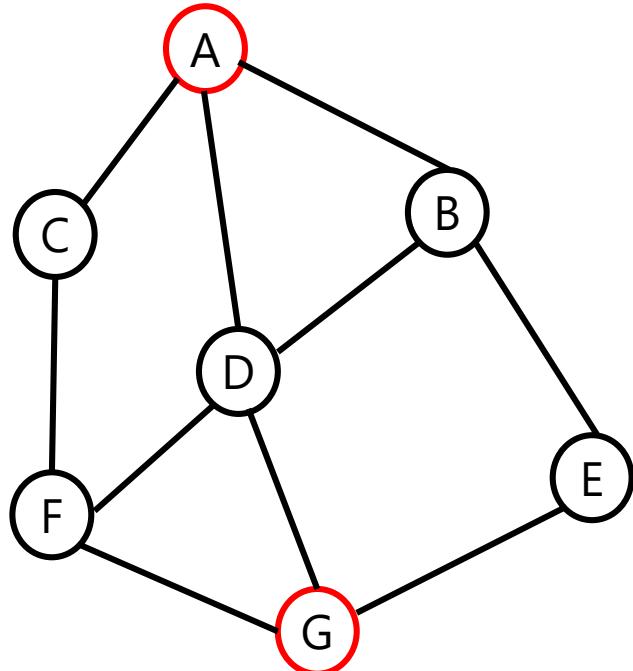


A

current \neq destination
current *not in* visited

visited = {}
frontier = ['A']

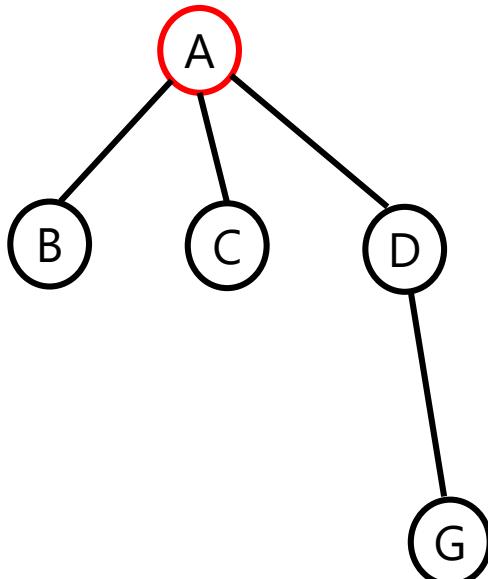
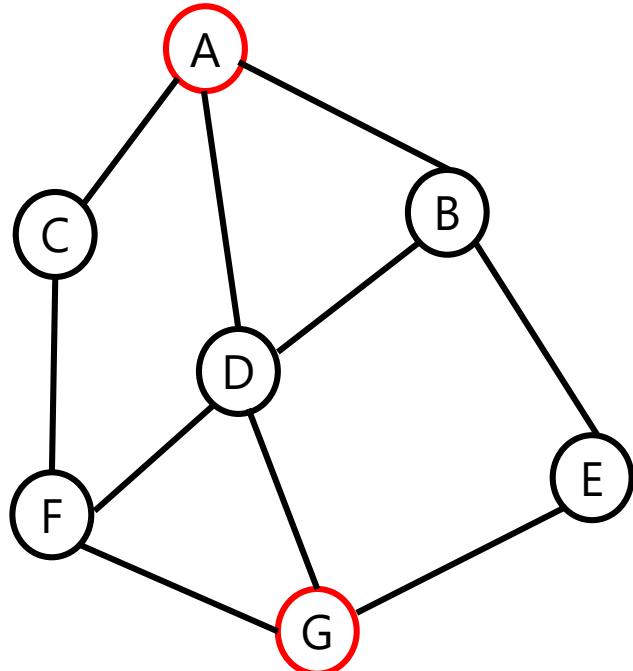
Depth-First Search



current \neq destination
current *not in* visited

visited = {'A'}
frontier = ['B', 'C', 'D']

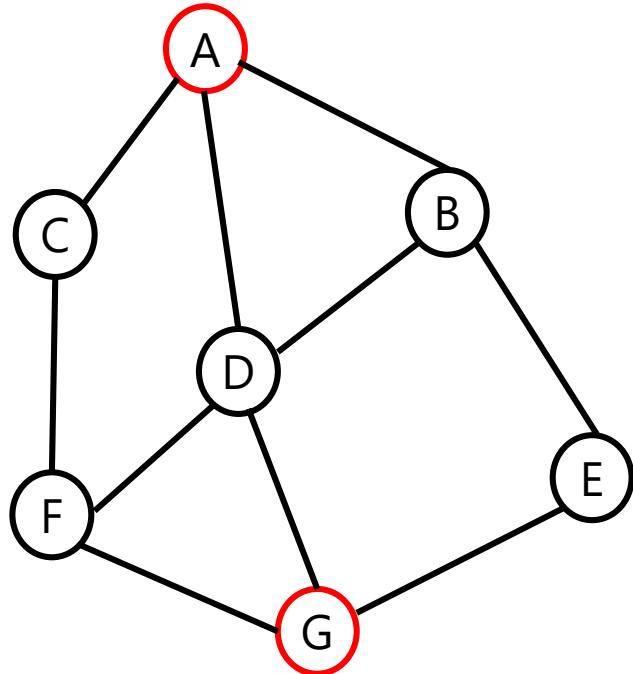
Depth-First Search



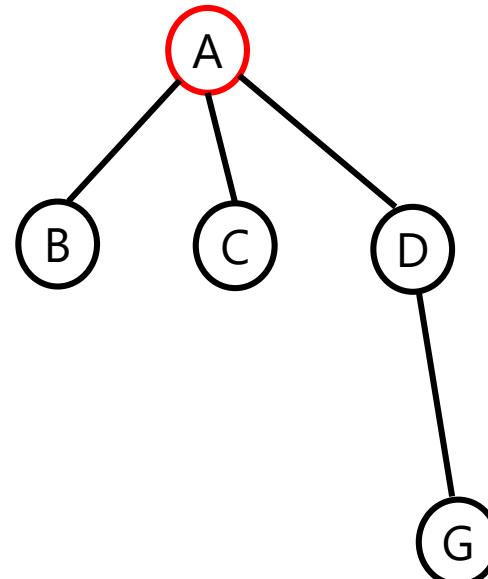
current \neq destination
current *not in* visited

visited = {'A', 'D'}
frontier = ['B', 'C', 'G']

Depth-First Search



```
visited = {'A', 'D'}  
frontier = ['B', 'C', 'G']
```

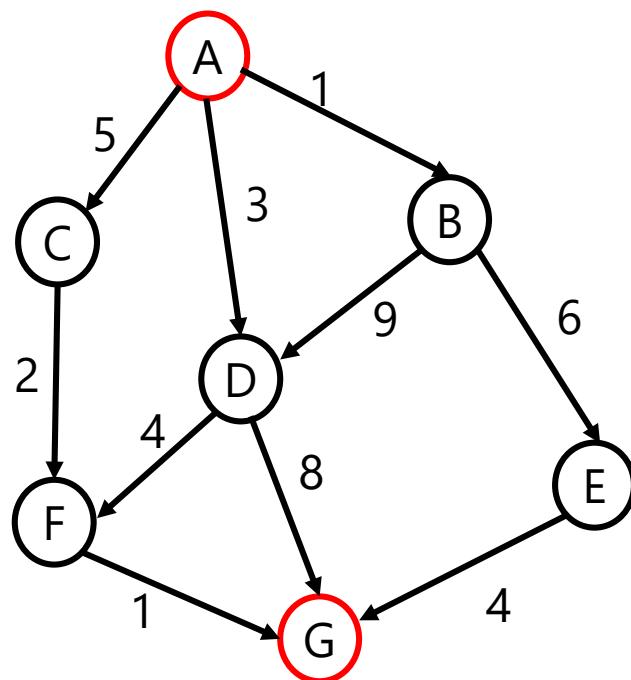


current == destination

Returns *True*

Weighted Graphs

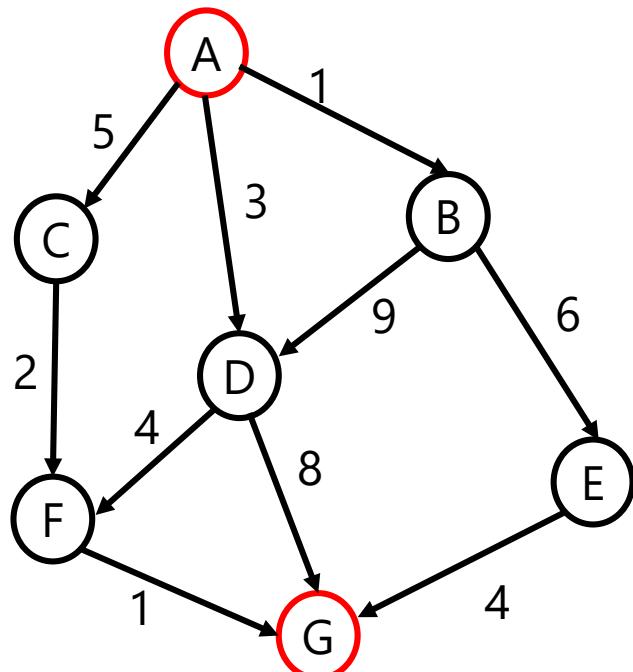
- Objective:
 - Finding distance of shortest path between a source vertex and goal vertex



Directed Acyclic Graph

How to represent the Weighted graph?

- Nested Adjacency Dictionary

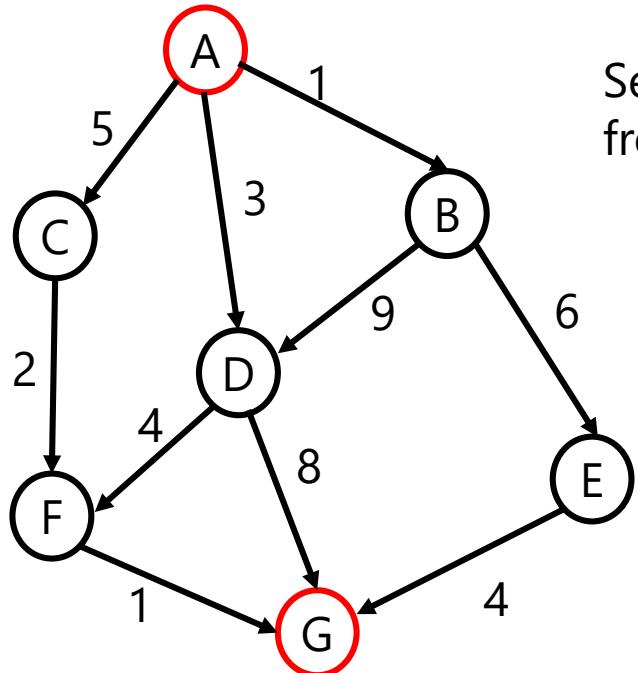


```
adjacencyGraph = { 'A' : { 'B' : 1, 'C' : 5, 'D' : 3 },  
                   'B' : { 'D' : 9, 'E' : 6 },  
                   'C' : { 'F' : 2 },  
                   'D' : { 'F' : 4, 'G' : 8 },  
                   'E' : { 'G' : 4 },  
                   'F' : { 'G' : 1 } }
```

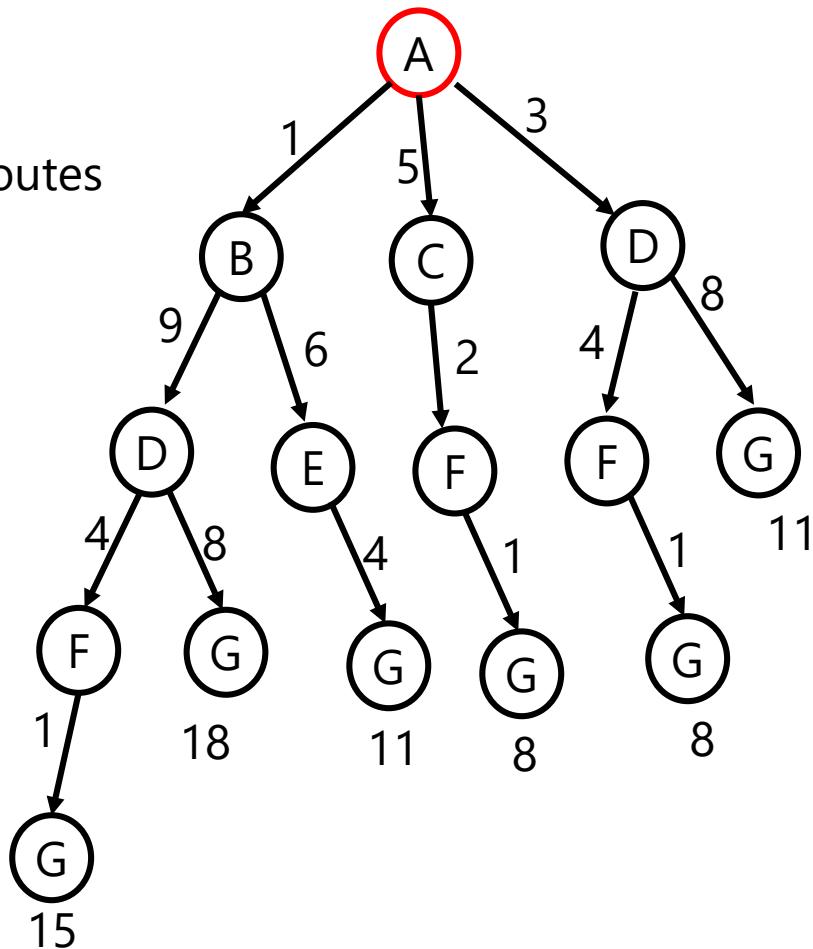
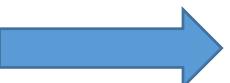
Algorithms

- Exhaustive Search
- Dynamic Programming, etc.
- Dijkstra's Algorithm

Exhaustive Search



Search all possible routes
from A to G

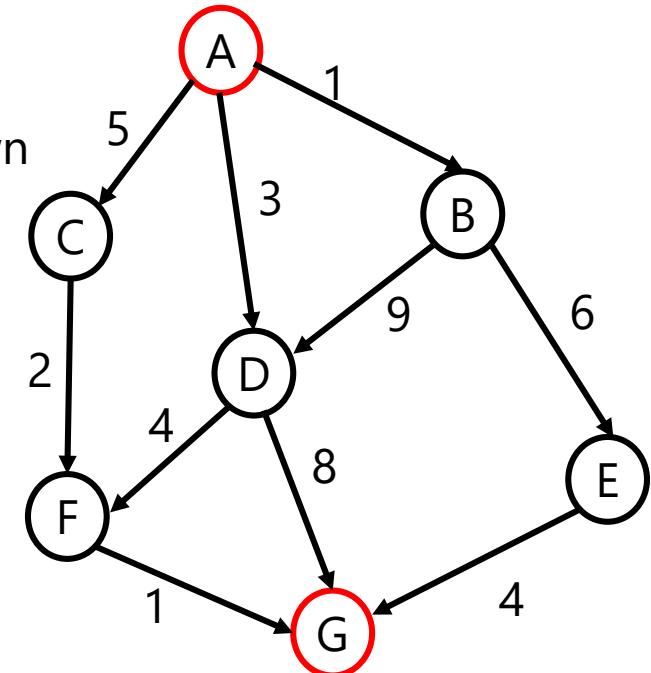


Recursive Algorithm

- Assumption
 - Shortest distance from neighbours of A to G is known
- Shortest distance from A to G is
 - $\min\{d(A, v) + d(v, G)\}, v \in \{B, C, D\}$

Distance from A to its neighbour v

Distance from neighbour v to G



- But how do you find distance from neighbour v to G ?
 - Repeat the above process, look at its neighbours and select shortest distance to G
 - Till G is found

Shortest Distance between Two Nodes

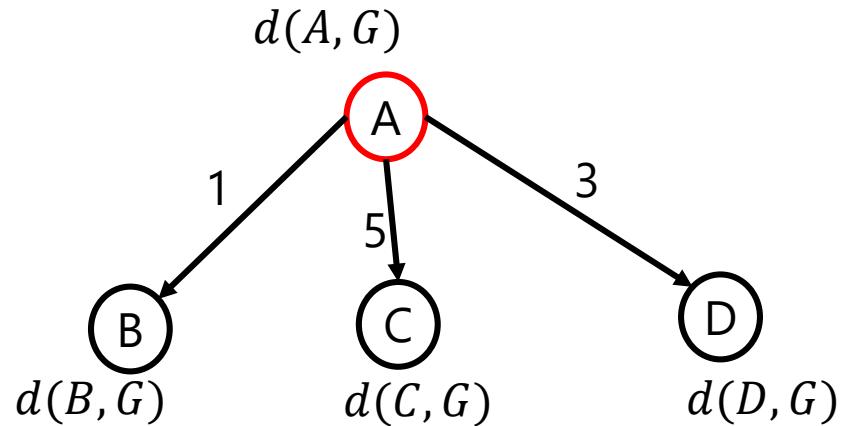
```
import math
def least_cost(adjDict, source, target):
    def d(vertex):
        if vertex == target:
            return 0
        try:
            return min(adjDict[vertex][i]+d(i) for i in adjDict[vertex])
        except:
            return math.inf
    return d(source)
```

$d(A, v)$
obtained from problem

neighbours of vertex

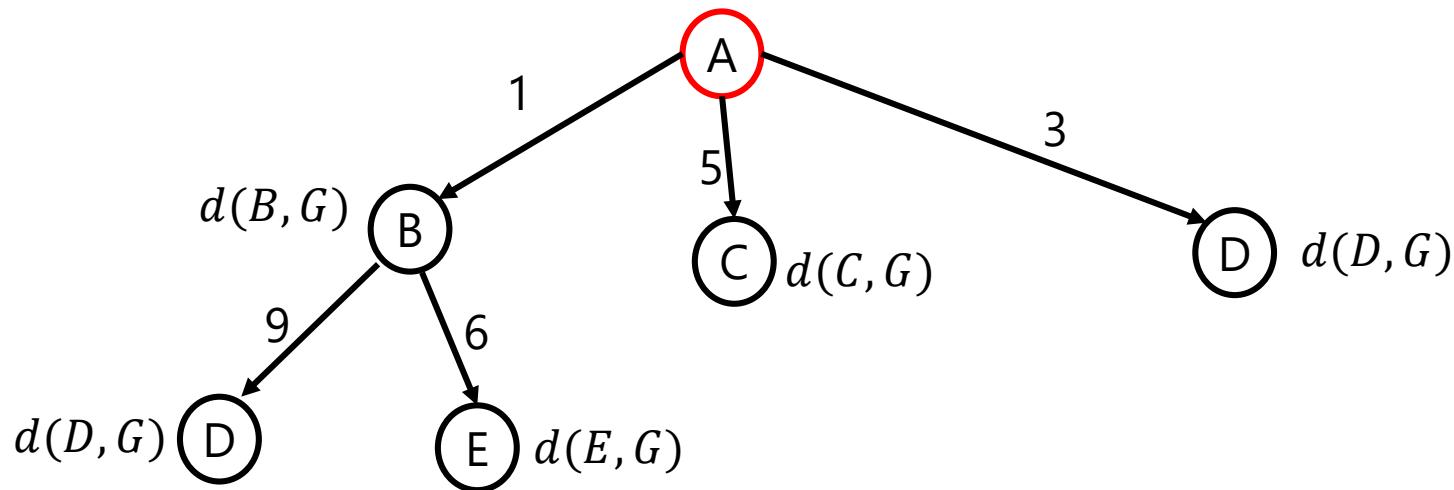
recursive function call $d(v, G)$

Shortest Distance: Recursive Method



$$d(A, G) = \min\{1 + d(B, G), 5 + d(C, G), 3 + d(D, G)\}$$

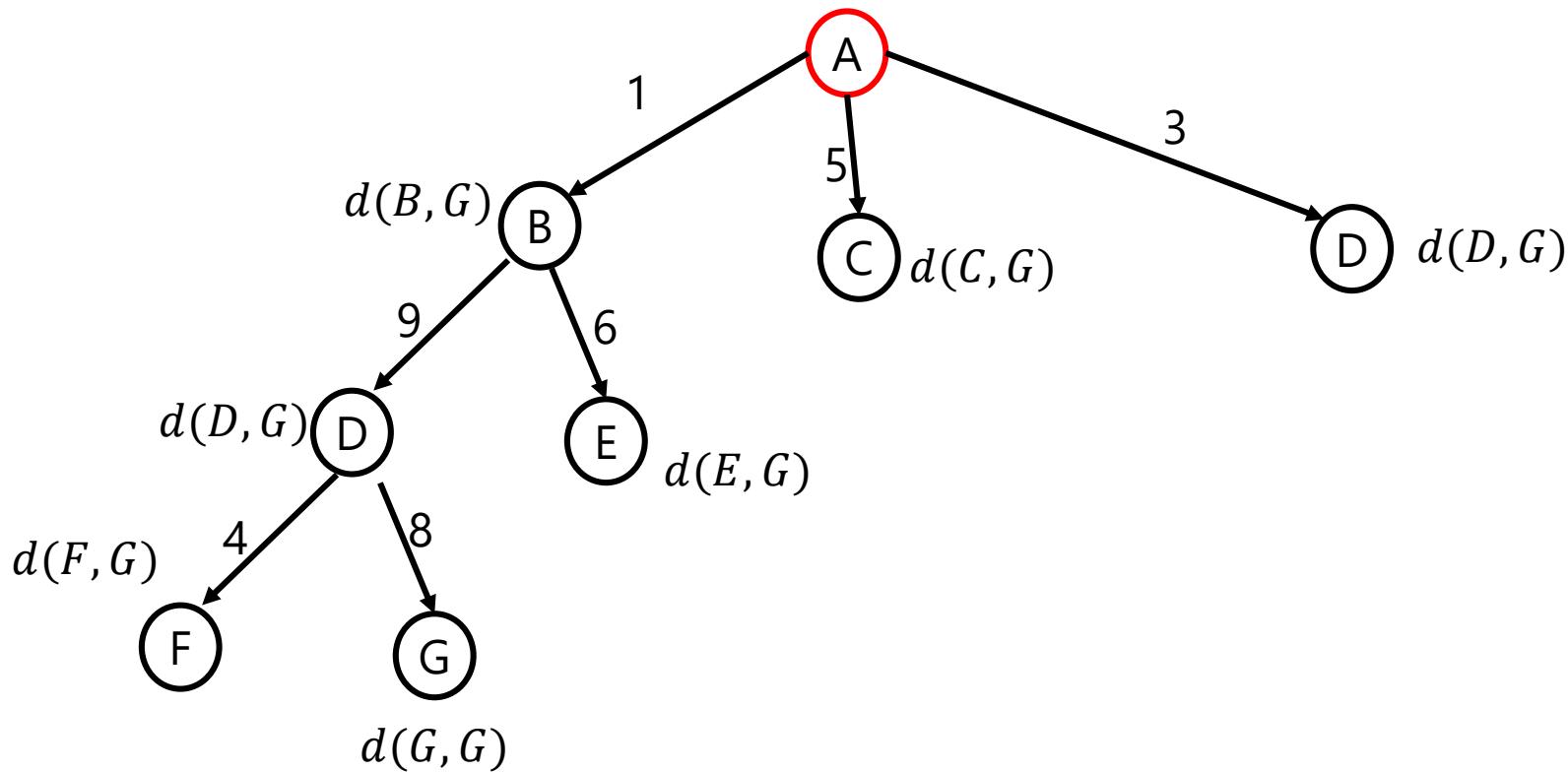
Shortest Distance: Recursive Method



$$d(A, G) = \min\{1 + d(B, G), 5 + d(C, G), 3 + d(D, G)\}$$

$$d(B, G) = \min\{9 + d(D, G), 6 + d(E, G)\}$$

Shortest Distance: Recursive Method

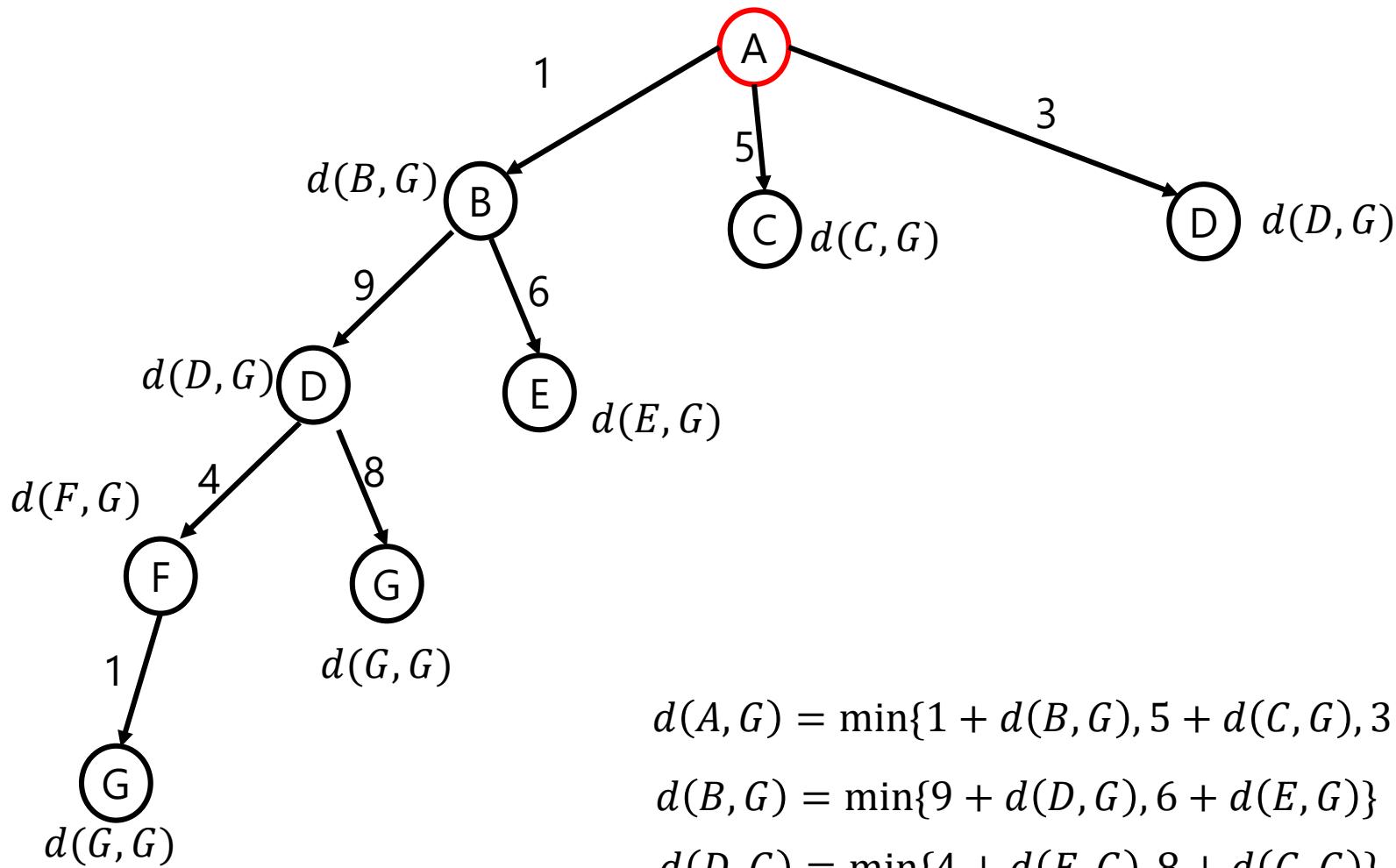


$$d(A, G) = \min\{1 + d(B, G), 5 + d(C, G), 3 + d(D, G)\}$$

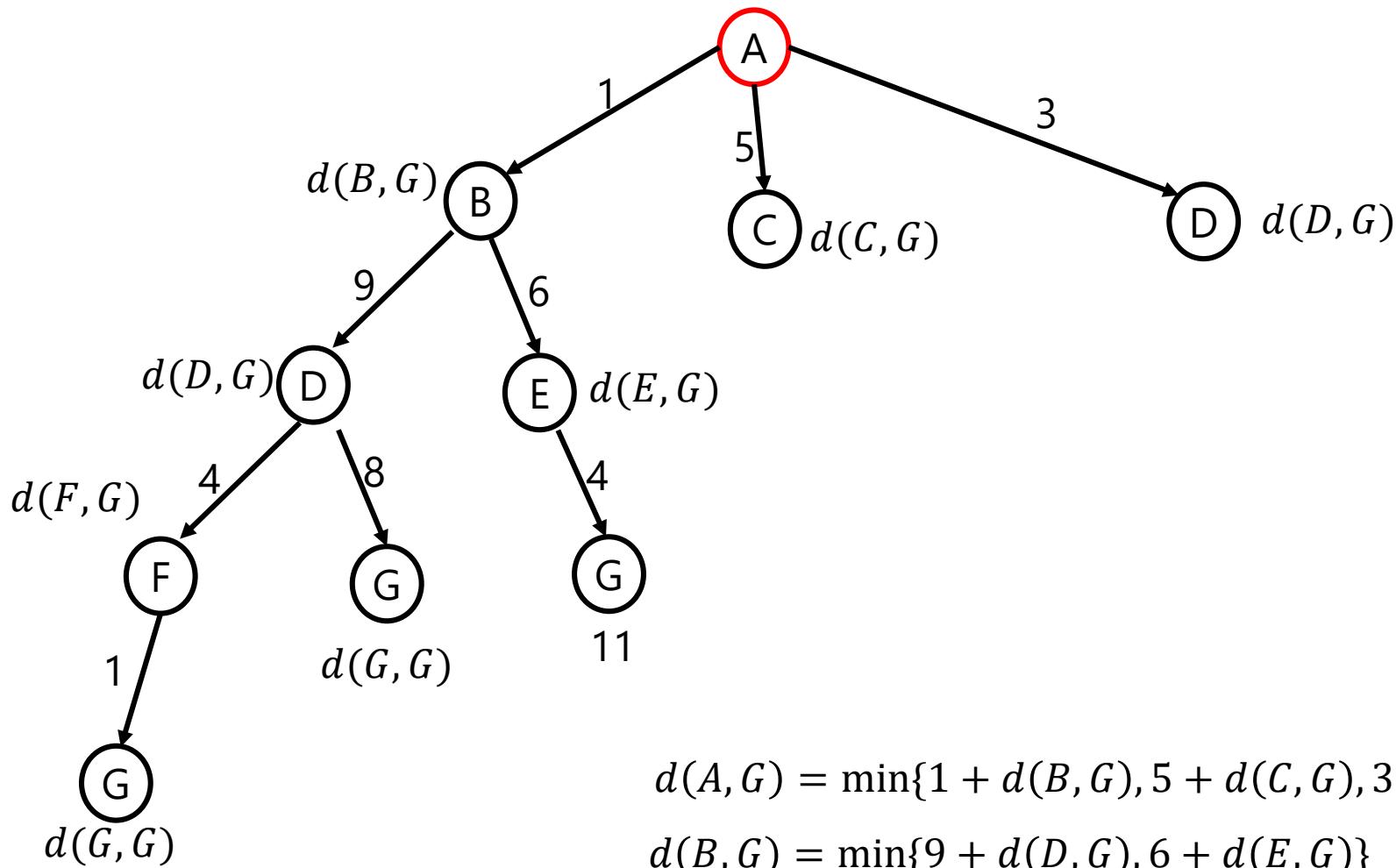
$$d(B, G) = \min\{9 + d(D, G), 6 + d(E, G)\}$$

$$d(D, G) = \min\{4 + d(F, G), 8 + d(G, G)\}$$

Shortest Distance: Recursive Method



Shortest Distance: Recursive Method

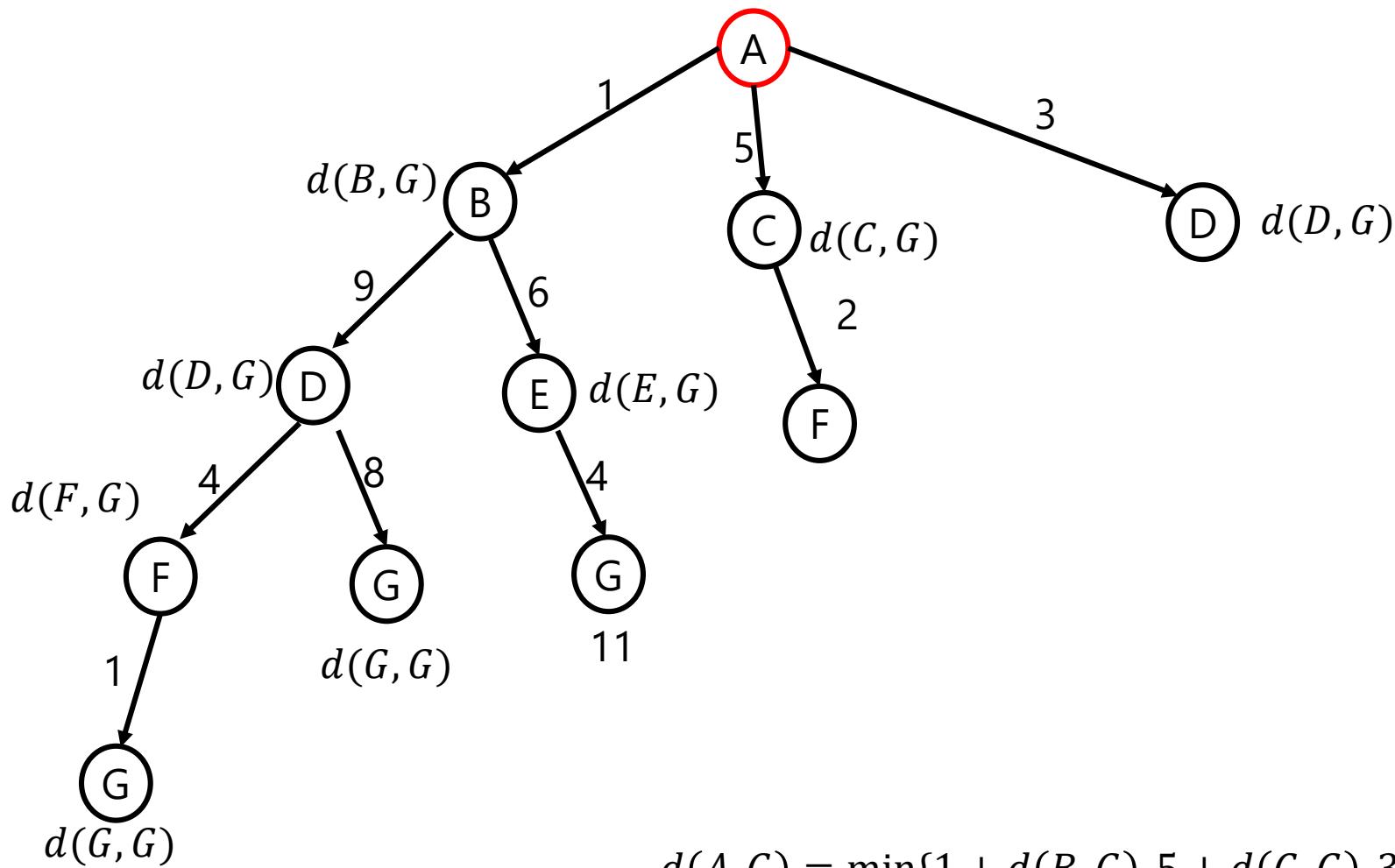


$$d(A, G) = \min\{1 + d(B, G), 5 + d(C, G), 3 + d(D, G)\}$$

$$d(B, G) = \min\{9 + d(D, G), 6 + d(E, G)\}$$

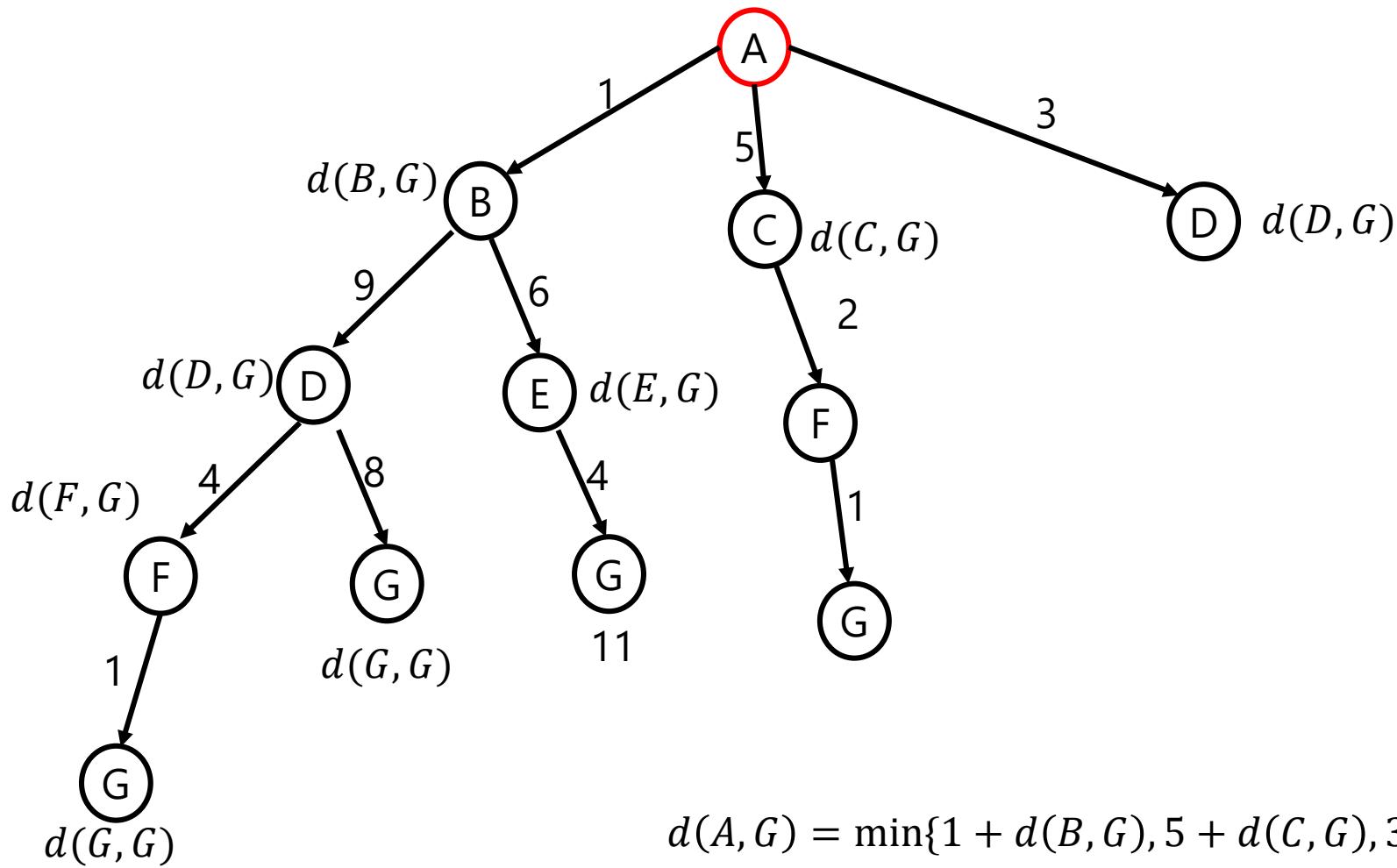
$$d(E, G) = 4 + d(G, G)$$

Shortest Distance: Recursive Method



$$d(A, G) = \min\{1 + d(B, G), 5 + d(C, G), 3 + d(D, G)\}$$
$$d(C, G) = 2 + d(F, G)$$

Shortest Distance: Recursive Method

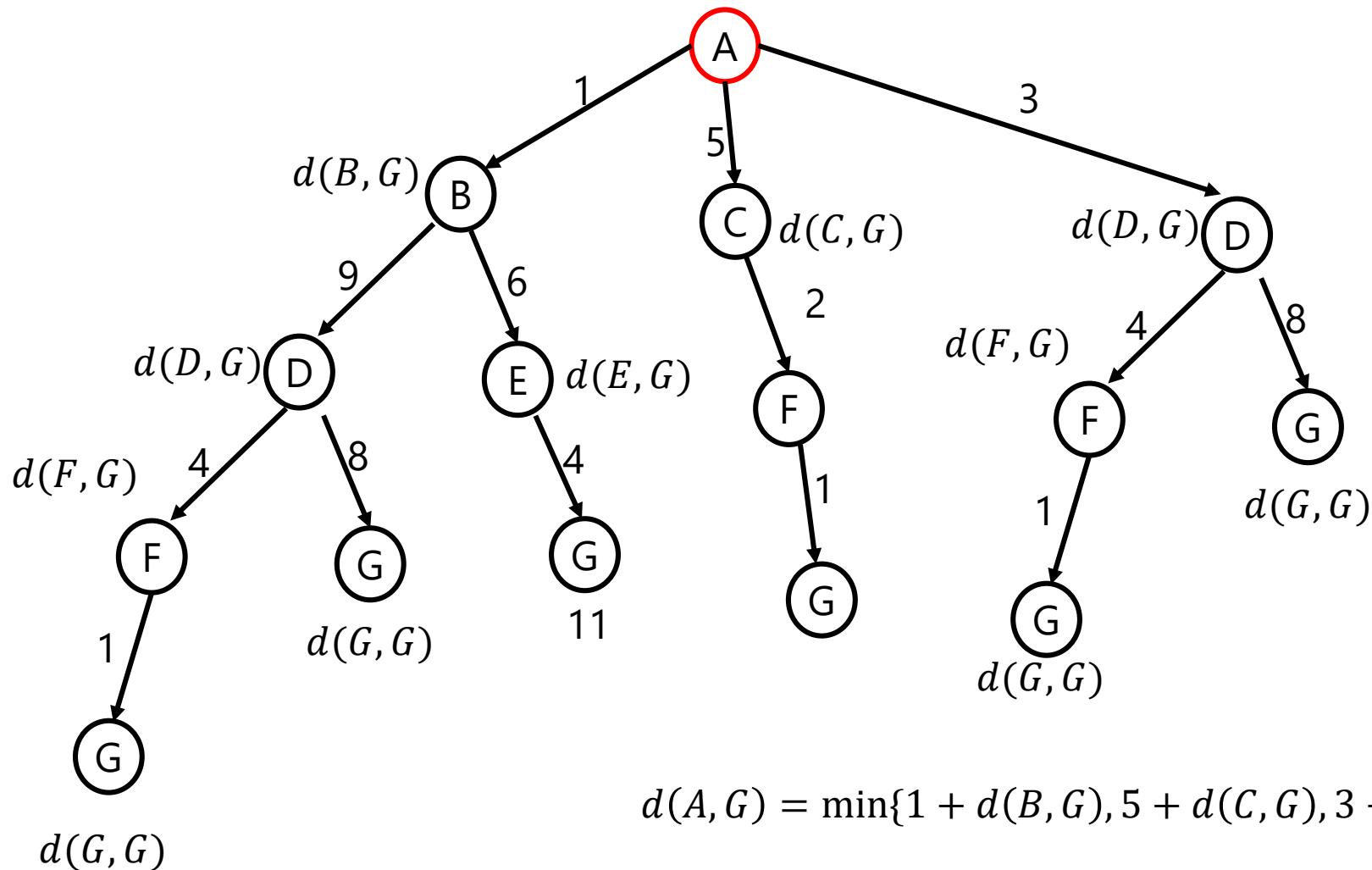


$$d(A, G) = \min\{1 + d(B, G), 5 + d(C, G), 3 + d(D, G)\}$$

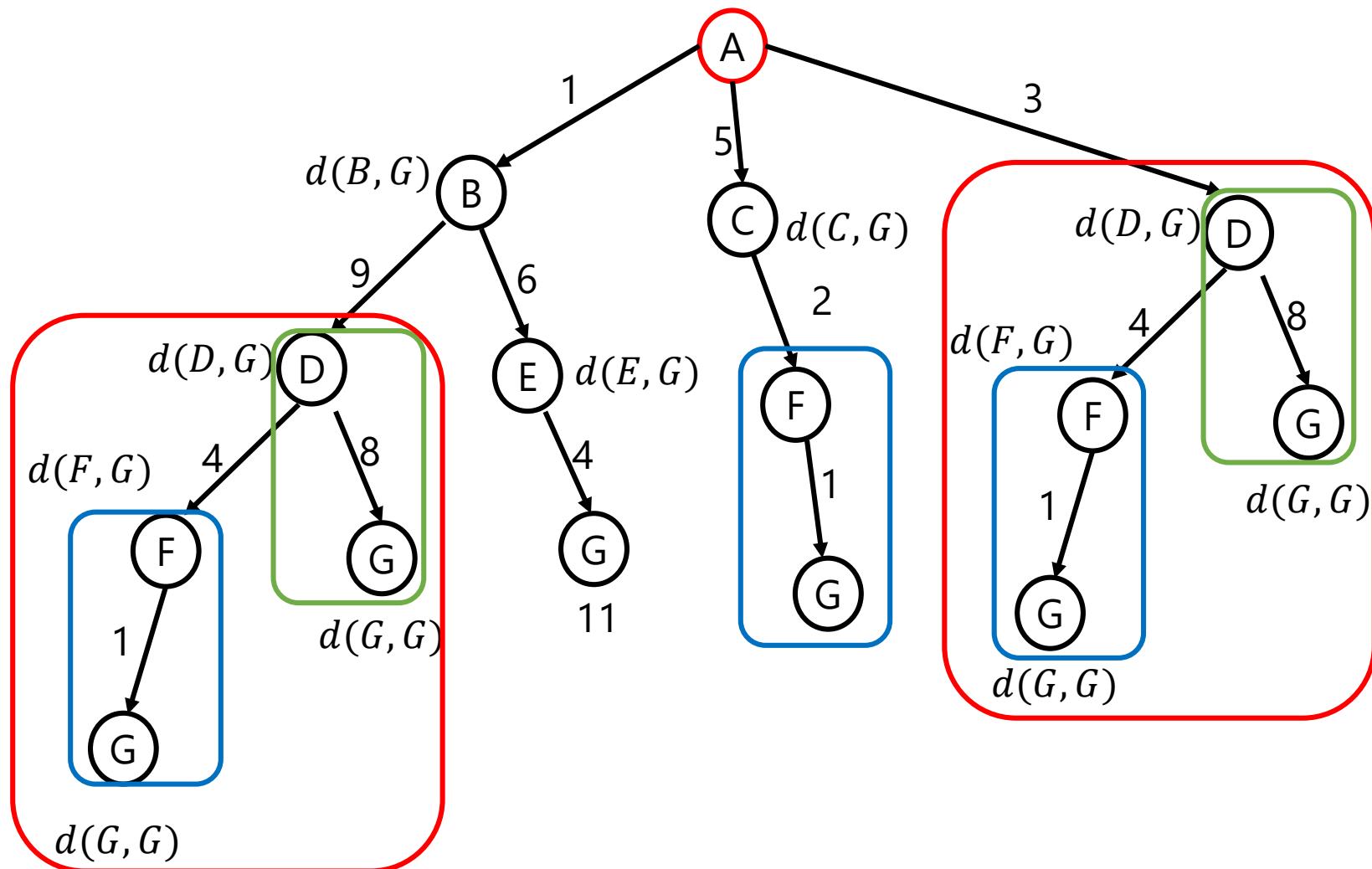
$$d(C, G) = 2 + d(F, G)$$

$$d(F, G) = 1 + d(G, G)$$

Shortest Distance: Recursive Method



Shortest Distance: Recursive Method



Shortest Distance: Memoized Recursive Version

```
import math
def least_cost(adjDict, source, target):
    @cache
    def d(vertex):
        if vertex == target:
            return 0
        try:
            return min(adjDict[vertex][i]+d(i) for i in adjDict[vertex])
        except:
            return math.inf

    return d(source)

print(least_cost(adjDict, 'A', 'G'))
```

Summary

- Dynamic Programming
 - General Problem Solving
 - Divide-and-conquer with redundant or overlapping subproblems
 - Optimization
 - Solving problems that have overlapping subproblems with optimal solutions
 - Subproblem dependency should be acyclic (i.e., only for DAGs)
 - Python's decorators simplifies memoization in DP

Program Design

Modularity and reusability

Complexity

In Engineering and Programming

Project Sizes in University

- Course assignment

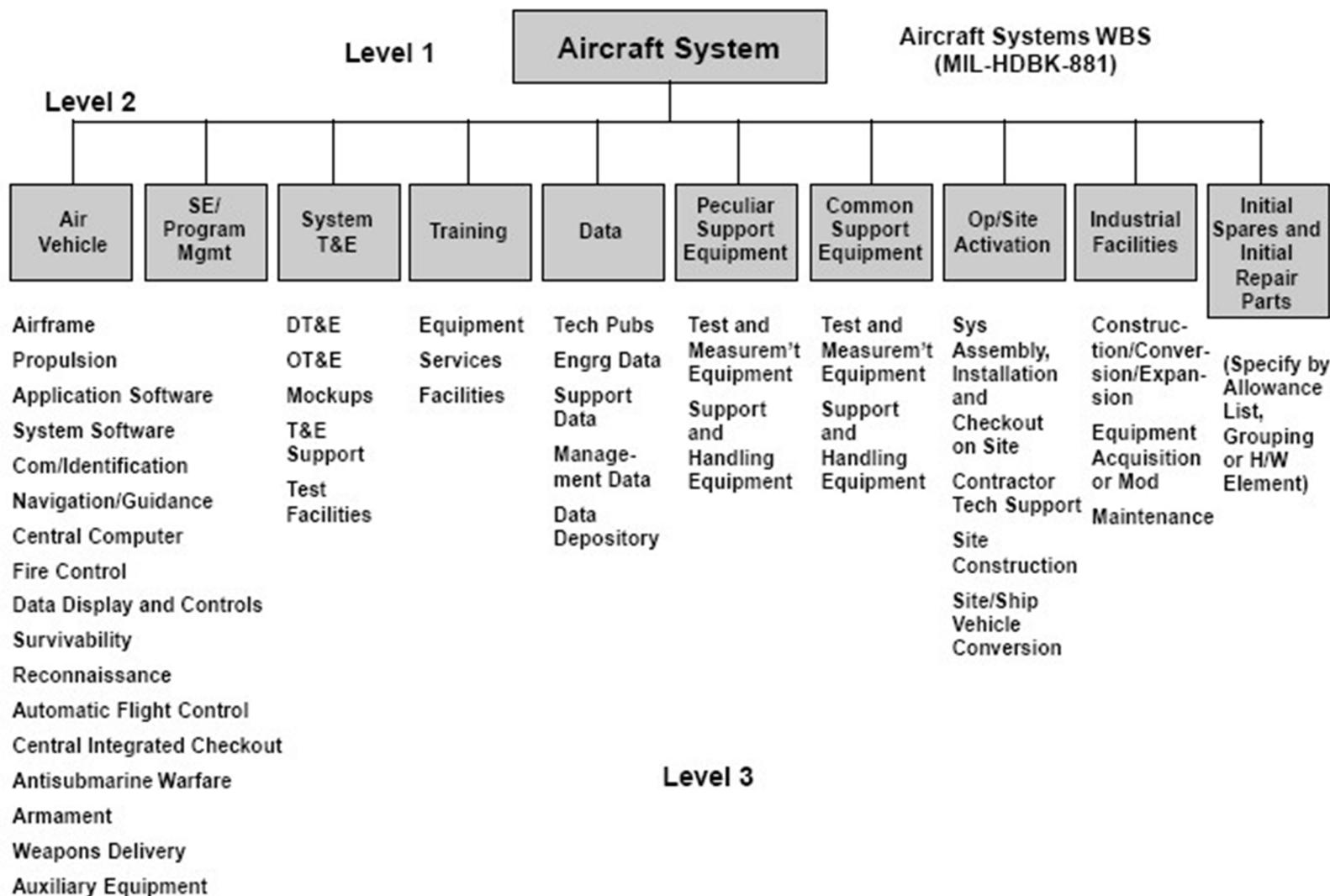


- Course project/FYP



Project Size at Work





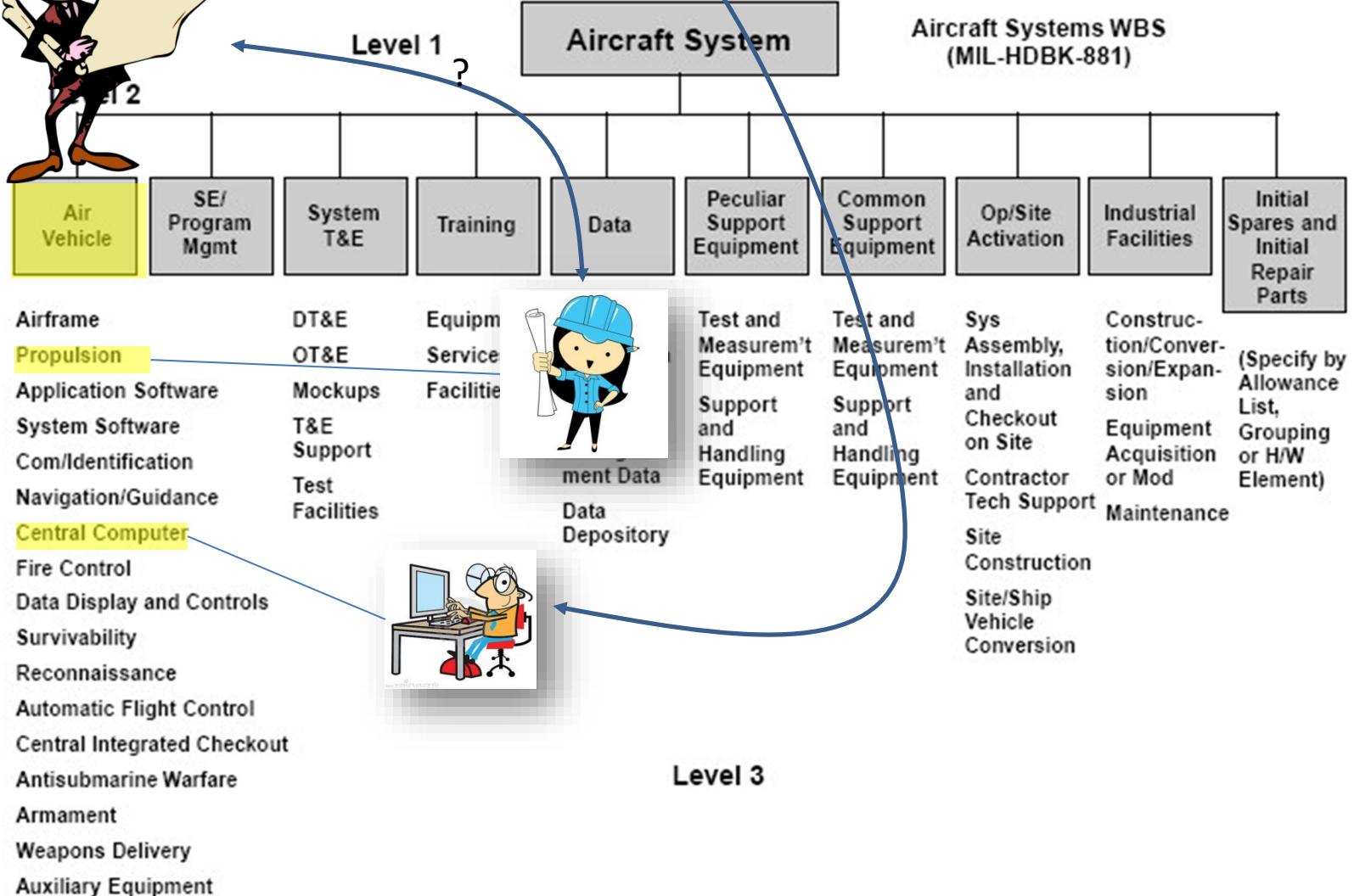
Managing Complexity

Abstraction

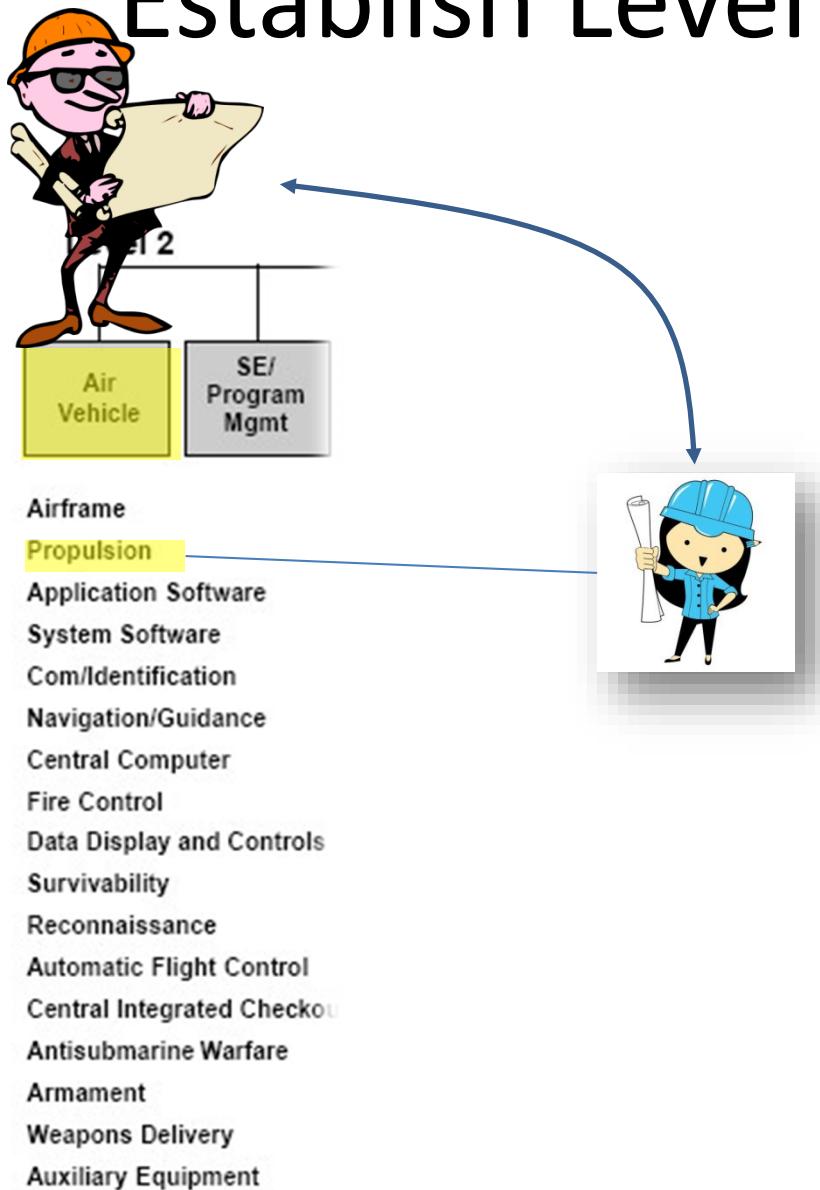
- A technique to manage complexity
- Establishing a level of complexity on which a person interacts with the system
- Suppressing the more complex details below the current level.



How they work together?



Establish Level of Complexity



- The chief engineer want to know

- How much fuel does the propulsion engine uses
- How much force can the engine provide

The level of complexity that the chief engineer wants to know

- But NOT

- How exactly the engine works
- How many parts are there in the engine

The level of complexity suppressed from the chief

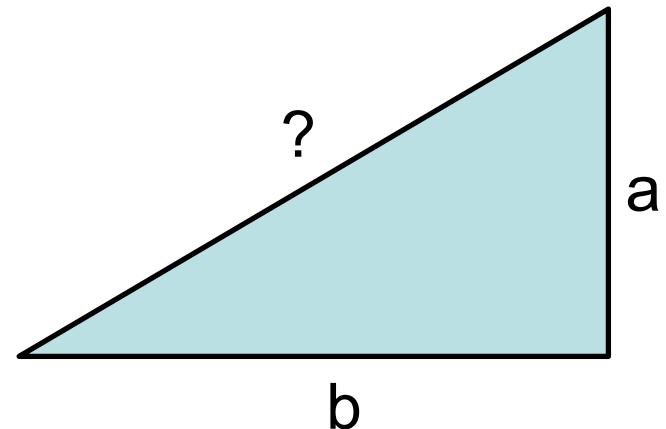


Compare:

```
def hypotenuse(a, b):  
    return sqrt(sum_of_squares(a, b))
```

```
def sum_of_squares(x, y):  
    return square(x) + square(y)
```

```
def square(x):  
    return x * x
```

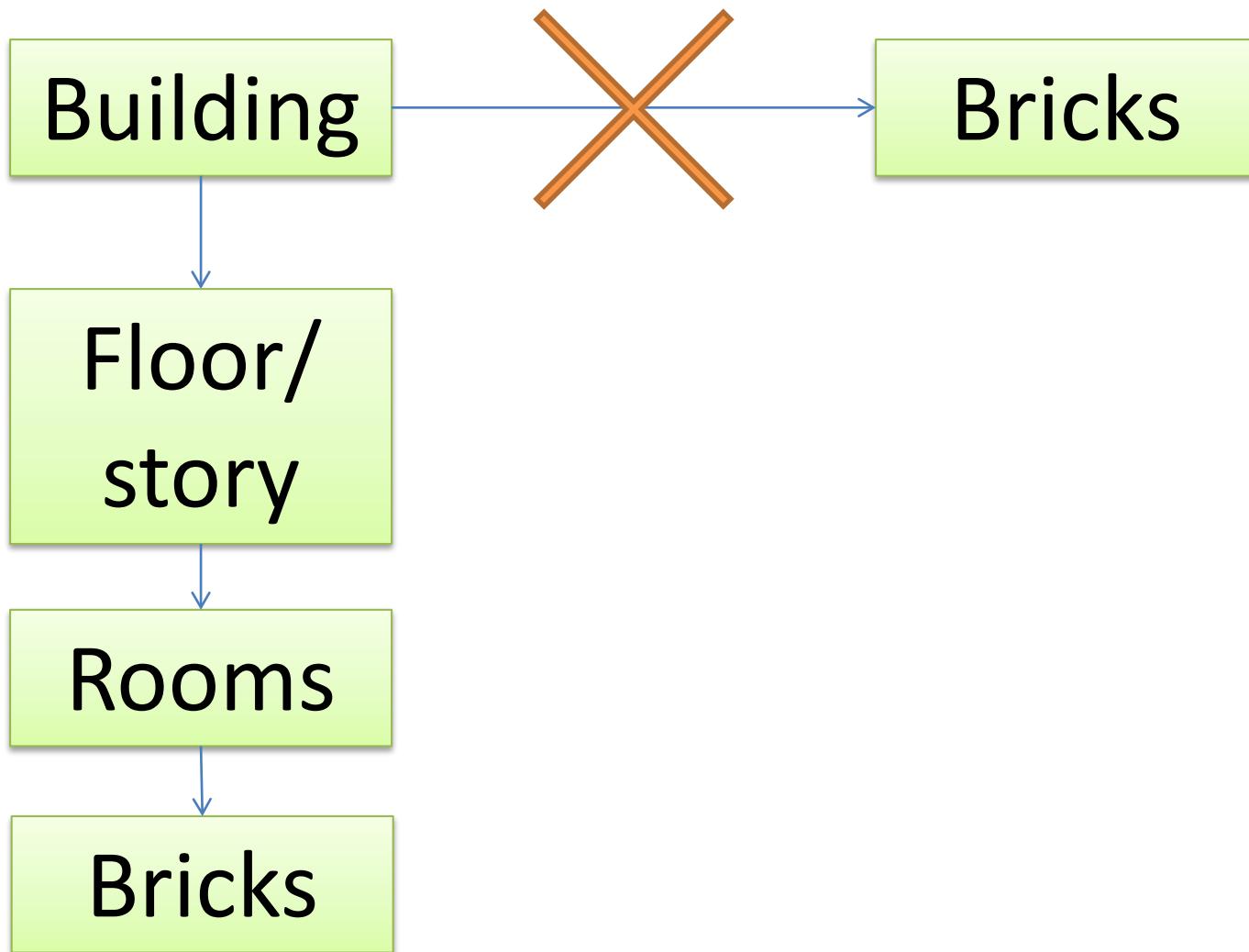


Versus:

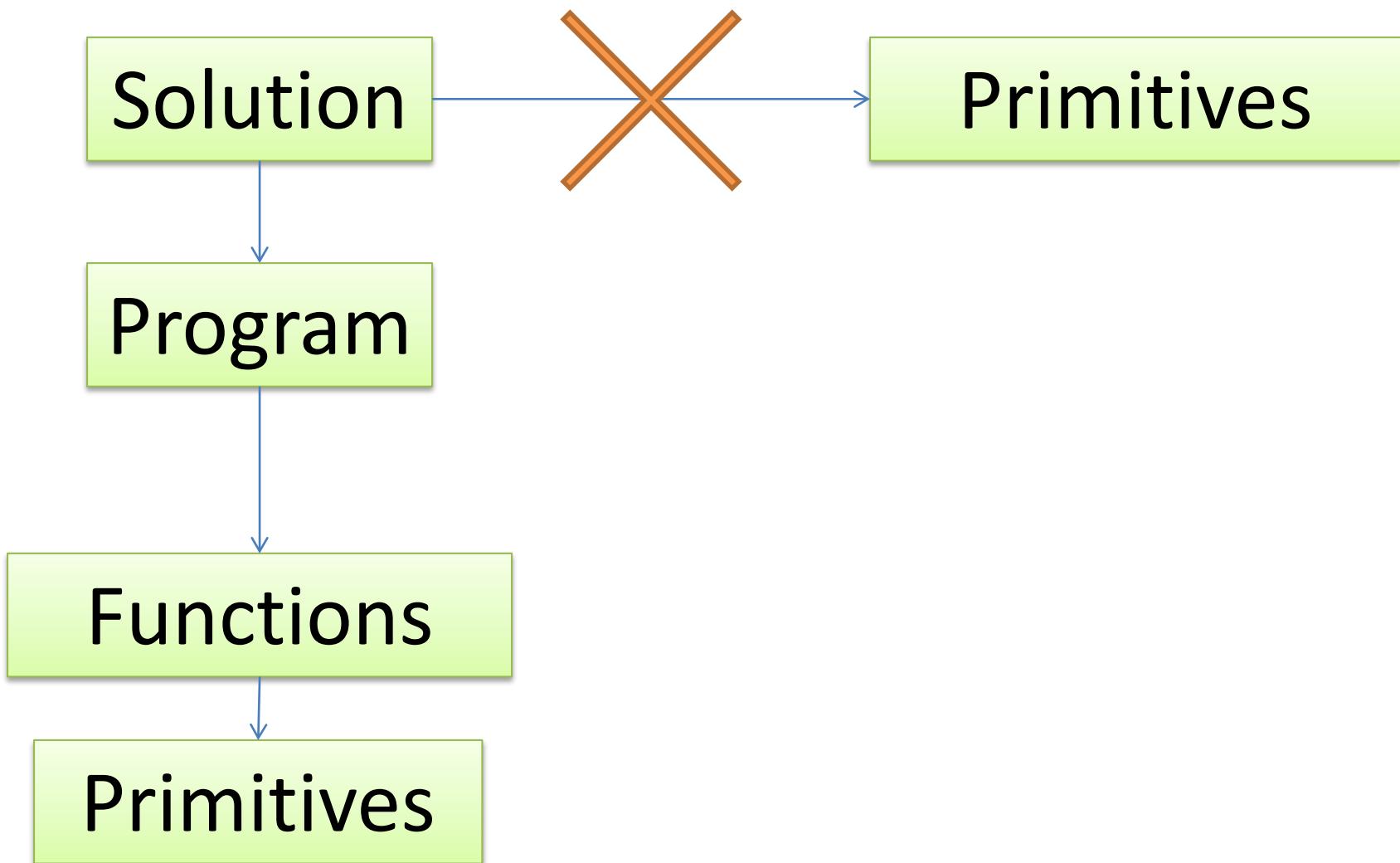
```
def hypotenuse(a, b):  
    return sqrt((a*a) + (b*b))
```

What Makes a Good Abstraction?

1. Makes it more natural to think about tasks and subtasks



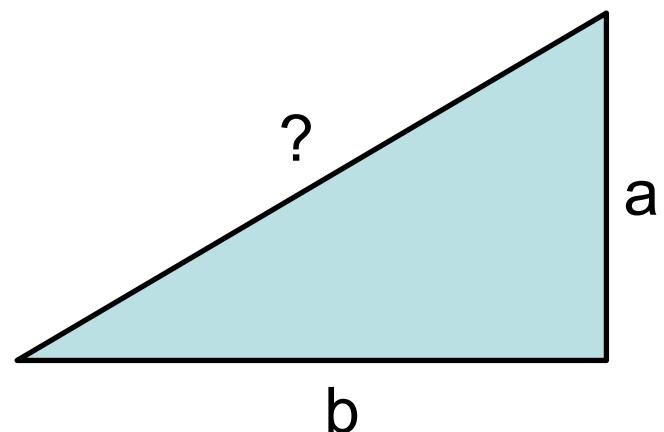
1. Makes it more natural to think about tasks and subtasks



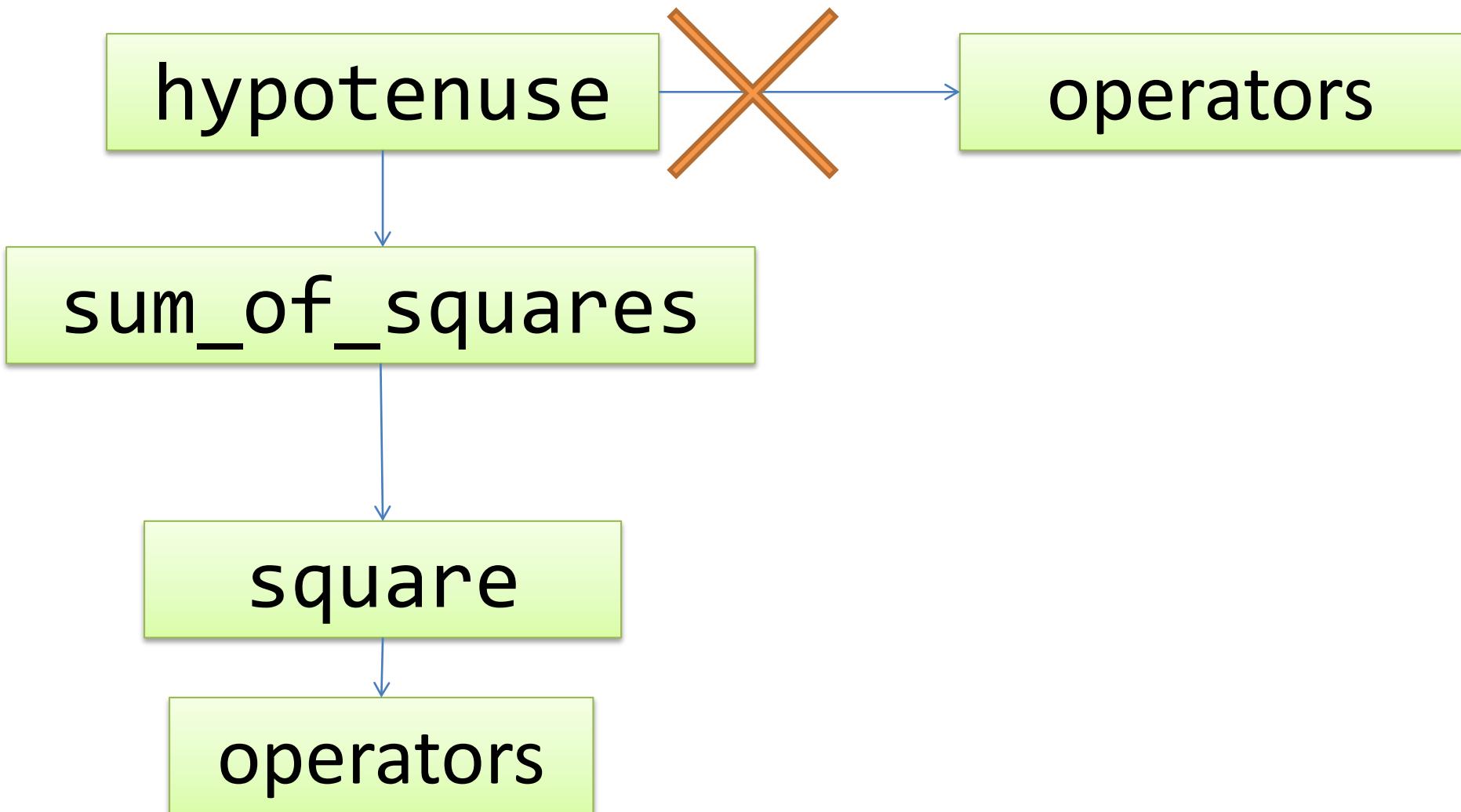
```
def hypotenuse(a, b):  
    return sqrt(sum_of_squares(a, b))
```

```
def sum_of_squares(x, y):  
    return square(x) + square(y)
```

```
def square(x):  
    return x * x
```

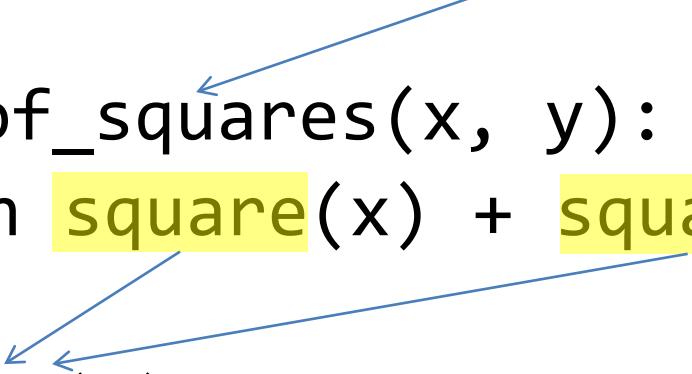


1. Makes it more natural to think about tasks and subtasks



2. Makes programs easier to understand

```
def hypotenuse(a, b):  
    return sqrt(sum_of_squares(a, b))  
  
def sum_of_squares(x, y):  
    return square(x) + square(y)  
  
def square(x):  
    return x * x
```



3. Captures common patterns

- E.g. computing binomial coefficient

$$\binom{n}{k} = \frac{n!}{k! (n - k)!}$$

- You won't write code like

```
def bc(n,k):  
    a = code for computing 1x2x3...x n  
    b = code for computing 1x2x3...x k  
    c = code for computing 1x2x3...x(n-k)  
    return a / (b * c)
```

Common
Patterns

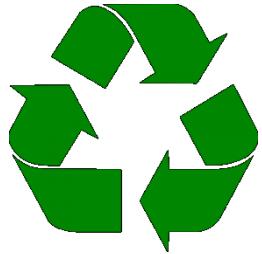
3. Captures common patterns

- E.g. computing binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

- Write a function `factorial(n)` then

```
def bc(n,k):  
    a = factorial(n)  
    b = factorial(k)  
    c = factorial(n-k)  
    return a / (b * c)
```



4. Allows for code reuse

- Function `square()` used in `sum_of_squares()`.
- `square()` can also be used in calculating area of circle.

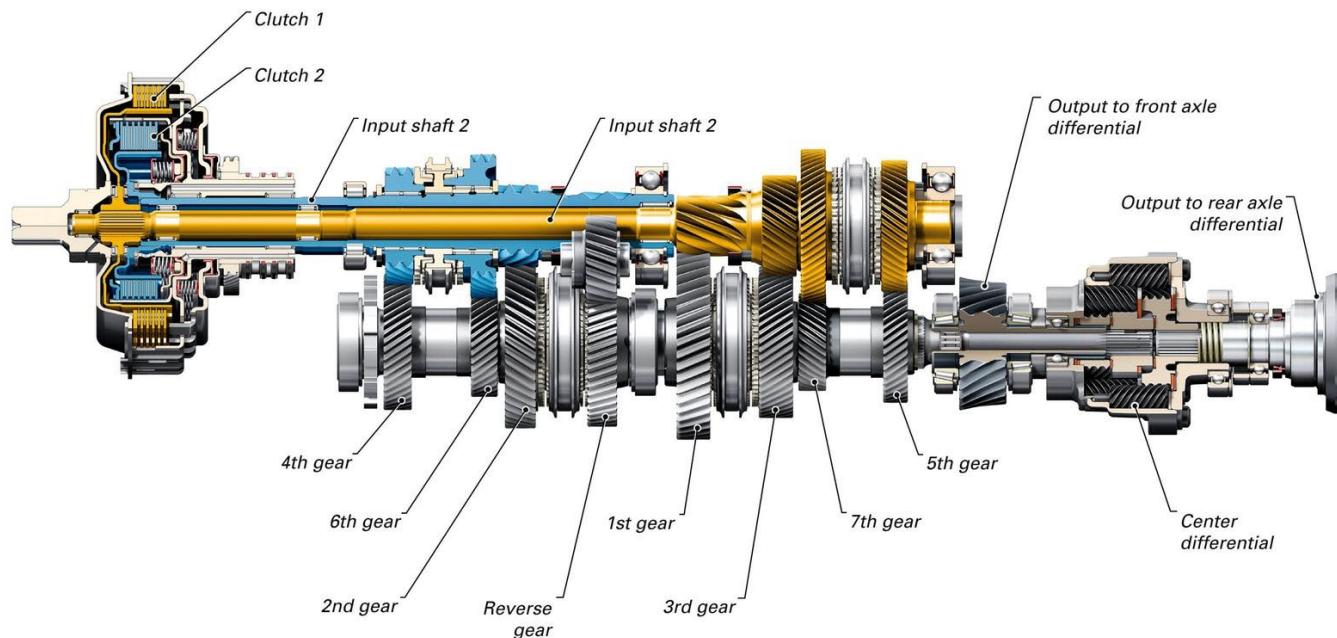
```
pi = 3.14159
```

```
def circle_area_from_radius(r):  
    return pi * square(r)
```

```
def circle_area_from_diameter(d):  
    return circle_area_from_radius(d/2)
```

5. Hides irrelevant details

- The structure of a gear box maybe interesting to some fans but not everyone who drives





sisley
PARIS

BOTANICAL BEAUTY PRODUCTS

Dior

CLARINS



AVEDA

Made of?

How does it work?



CELLEX-C®

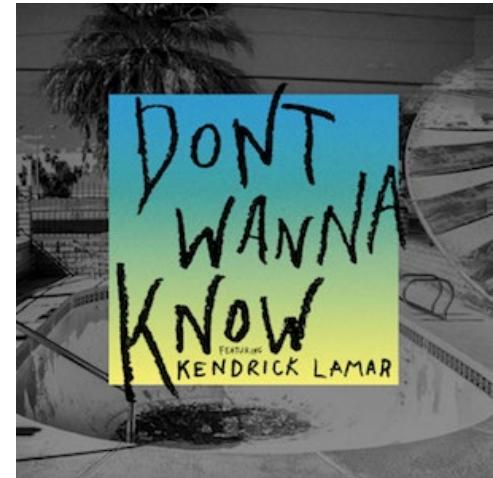


6. Separates specification from implementation

- Specification: WHAT IT DOES
 - E.g the function $\cos(x)$ compute the cosine of x
- Implementation: HOW IT DOES

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}$$

A mathematical formula for the Taylor series expansion of the cosine function. It shows the sum of even powers of x divided by the factorial of the power, starting from $n=0$. A large orange 'X' is drawn through the entire formula.



Different ways of implementing square(x)

```
def square(x):  
    return x * x
```

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

```
def square(x):  
    return exp(double(log(x)))  
def double(x): return x + x
```

7. Makes debugging (fixing errors) easier

```
def hypotenuse(a, b):  
    return sqrt((a + a) * (b + b))
```

Where is/are
the bugs?

```
def hypotenuse(a,b):  
    return sqrt(sum_of_squares(a,b))
```

```
def sum_of_squares(x, y):  
    return square(x) * square(y)
```

```
def square(x):  
    return x + x
```

Good Abstraction?

1. Makes it more natural to think about tasks and subtasks
2. Makes programs easier to understand
3. Captures common patterns
4. Allows for code reuse
5. Hides irrelevant details
6. Separates specification from implementation
7. Makes debugging easier

Program Design

Top-down Approach

pretend you have whatever you need

Sequence of Writing

1

```
def hypotenuse(a, b):  
    return sqrt(sum_of_squares(a, b))
```

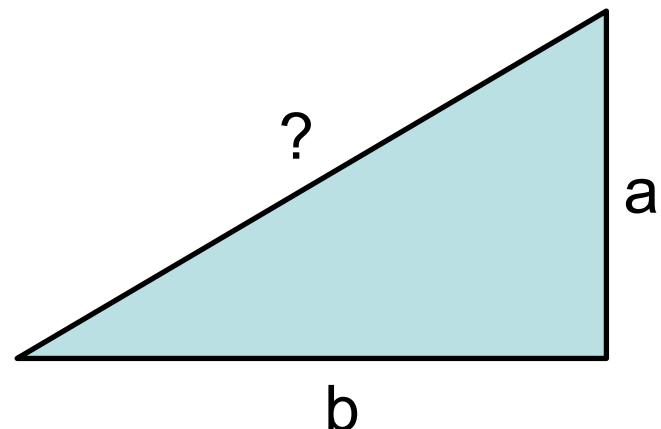
2

```
def sum_of_squares(x, y):  
    return square(x) + square(y)
```

3

```
def square(x):  
    return x * x
```

pretend you have
whatever you need



Another Example

- NTUC Comfort, the largest taxi operator in Singapore, determines the taxi fare based on distance traveled as follows:

Basic fare	Normal
Flag-Down (inclusive of 1st km or less)	\$3.00-\$3.40
Every 400m thereafter or less up to 10km	\$0.22
Every 350 metres thereafter or less after 10 km	\$0.22
Every 45 secs of waiting or less	\$0.22

Problem: Write a Python function
that computes the taxi fare from the
distance traveled.

How do we start?

Formulate the Problem

- We need a name!
 - Pick a meaningful one
 - Definitely not “foo”

Function

Formulate the Problem

- What are the
 - Input?
 - Output?



Formulate the Problem

- How exactly should we design the function?
 1. Try a few simple examples.
 2. Strategize step by step.
 3. Write it down and refine.

Solution

- What to call the function? `taxi_fare`
- What data are required? `distance`
- Where to get data? `function argument`
- What is the result? `fare`

Try a few simple examples

- e.g.#1: distance = 800 m, fare = \$3.00
- e.g.#2: distance = 3300 m,
 - fare = \$3.00 + $\text{roundup}(2300/400) \times \$0.22 = \$4.32$
- e.g.#3: distance = 14500 m,
 - fare = \$3.00 + $\text{roundup}(9000/400) \times \$0.22 +$
 $\text{roundup}(4500/350) \times \$0.22 = \$10.92$

Basic fare	Normal
Flag-Down (inclusive of 1st km or less)	\$3.00-\$3.40
Every 400m thereafter or less up to 10km	\$0.22
Every 350 metres thereafter or less after 10 km	\$0.22
Every 45 secs of waiting or less	\$0.22

Pseudocode

- Case 1: $\text{distance} \leq 1000$
 - fare = \$3.00
- Case 2: $1000 < \text{distance} \leq 10,000$
 - fare = $\$3.00 + \$0.22 * \text{roundup}((\text{distance} - 1000) / 400)$
- Case 3: $\text{distance} > 10,000$
 - fare = $\$3.00 + \text{roundup}(9000 / 400) + \$0.22 * \text{roundup}((\text{distance} - 10,000) / 350)$
- Note: the Python function ceil rounds up its argument.
 $\text{math.ceil}(1.5) = 2$

Pseudocode (Refined)

- Case 1: $\text{distance} \leq 1000$
 - fare = \$3.00
- Case 2: $1000 < \text{distance} \leq 10,000$
 - fare = $\$3.00 + \$0.22 * \text{roundup}((\text{distance} - 1000) / 400)$
- Case 3: $\text{distance} > 10,000$
 - fare = $\$8.06 + \$0.22 * \text{roundup}((\text{distance} - 10,000) / 350)$
- Note: the Python function `ceil` rounds up its argument.
`math.ceil(1.5) = 2`

Solution

```
def taxi_fare(distance): # distance in metres
    if distance <= 1000:
        return 3.0
    elif distance <= 10000:
        return 3.0 +
            (0.22*ceil((distance-1000)/400))
    else:
        return 8.06 +
            (0.22*ceil((distance-10000)/350))

# check: taxi_fare(3300) = 4.32
```

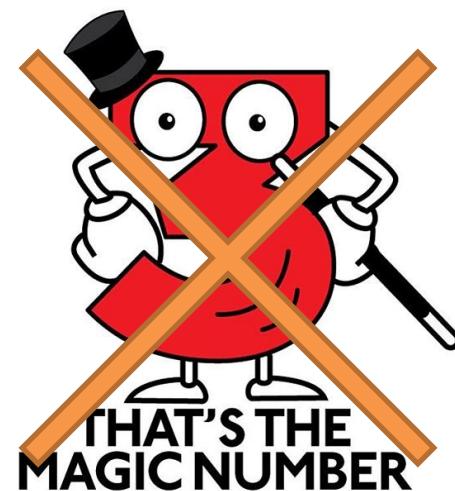
Coping with Changes

- What if starting fare is increased to \$3.20?
- What if 385 m is increased to 400 m?



Avoid Magic Numbers

- It is a terrible idea to hardcode constants (magic numbers):
 - Hard to make changes in future
- Define abstractions to hide them!



```
stage1 = 1000
stage2 = 10000
start_fare = 3.0
increment = 0.22
block1 = 400
block2 = 350
```

Better to make them all
CAPS for “normal”
convention

```
def taxi_fare(distance): # distance in metres
    if distance <= stage1:
        return start_fare
    elif distance <= stage2:
        return start_fare + (increment *
            ceil((distance - stage1) /
block1))
    else:
        return taxi_fare(stage2) +
            (increment * ceil((distance -
stage2) / block2))
```

How to I Manage My Own Code?

Let's say I wrote some cool code

```
def square(x) :  
    return x * x  
  
def singHappyBirthdayTo(name) :  
    print('Happy birthday To You!')  
    print('Happy birthday To You!')  
    print('Happy birthday to ' + name + '~')  
    print('Happy birthday to You!!!')
```

- And I save it to a file called

my_cool_package.py

Then I can use it for another file

- Another file:

Same as the file name but without “.py”

```
import my_cool_package
from math import pi

def circle_area_by_radius(r):
    return pi * my_cool_package.square(r)

print(circle_area_by_radius(10))
```

Or

- Another file:

```
import my_cool_package as mcp
from math import pi

def circle_area_by_radius(r):
    return pi * mcp.square(r)

print(circle_area_by_radius(10))
```

Or

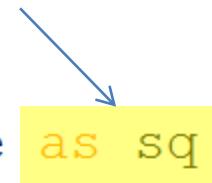
- Another file

```
from my_cool_package import square  
  
def squared_sum(a,b):  
    return square(a) + square(b)  
  
print(squared_sum(3,4))
```

Or

- Another file

But in general, it's not a good habit to name a variable/function so short that you cannot understand what it does



```
from my_cool_package import square as sq
```

```
def squared_sum(a,b):  
    return sq(a) + sq(b)
```

```
print(squared_sum(3,4))
```

|