Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

Jan. 11 – 15, 2021

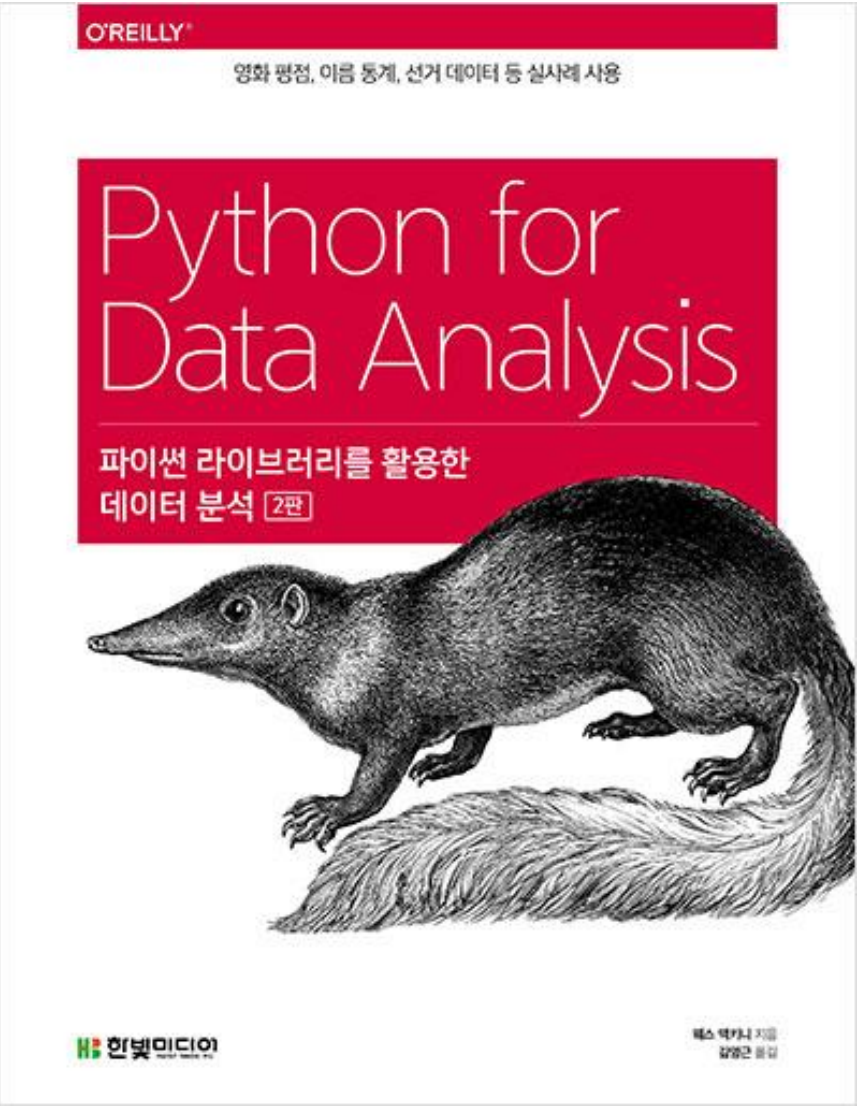
Python for Data Analytics

Python Basics

교재

- 파이썬 라이브러리를 활용한 데이터 분석 (2판)
- 김영근 옮김
- 한빛미디어, 2019.

- Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython (2nd Ed.)
- By Wes McKinney
- O'Reilly Media, 2017



일정

<table>
<tr><th></th><th></th><th>1/11 (Mon)</th><th>1/12 (Tue)</th><th>1/13 (Wed)</th><th>1/14 (Thu)</th><th>1/15 (Fri)</th></tr>
<tr><td rowspan="4">오전</td><td>8:30</td><td rowspan="4">Python Basics</td><td rowspan="4">NumPy</td><td rowspan="3">Pandas I</td><td rowspan="2">Pandas II</td><td rowspan="4">Data Preprocessing</td></tr>
<tr><td>9:30</td></tr>
<tr><td>10:30</td><td rowspan="2">Matplotlib II</td></tr>
<tr><td>11:30</td><td>Matplotlib I</td></tr>
<tr><td></td><td>12:30</td><td colspan="5">점심 시간 (12:30 ~ 1:30)</td></tr>
<tr><td rowspan="5">오후</td><td>1:30</td><td rowspan="5">[실습]</td><td rowspan="5">[실습]</td><td rowspan="5">[실습]</td><td rowspan="5">[실습]</td><td rowspan="5">[실습]</td></tr>
<tr><td>2:30</td></tr>
<tr><td>3:30</td></tr>
<tr><td>4:30</td></tr>
</table>

About Us

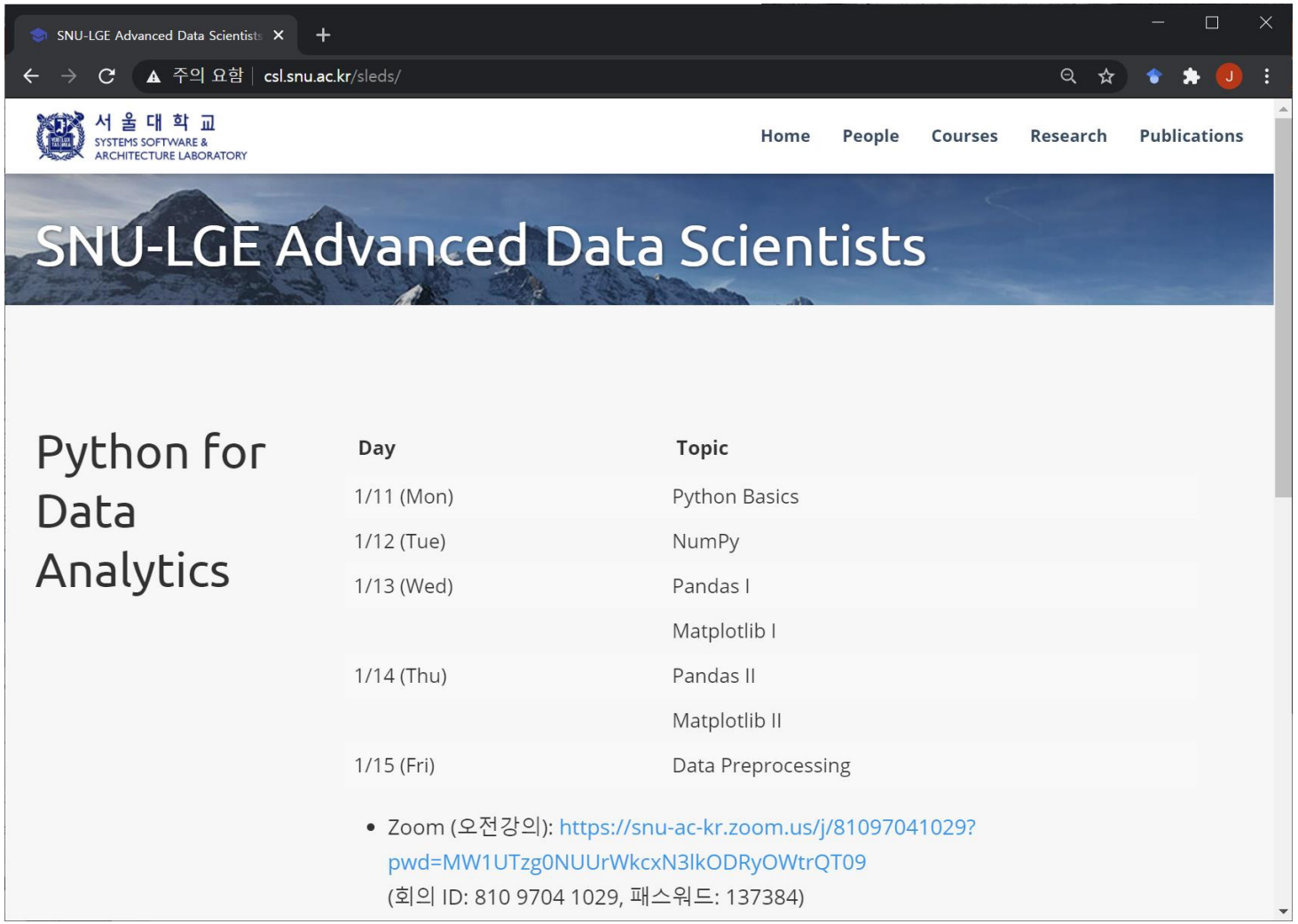
- 김진수 (Jin-Soo Kim)
  - Professor @ Dept. of Computer Science & Engineering, SNU
  - Systems Software & Architecture Laboratory
  - Operating systems, Storage systems, Parallel and Distributed computing, Embedded systems, …
- E-mail: jinsoo.kim@snu.ac.kr
- http://csl.snu.ac.kr
- Tel: 02-880-7302
- Office: SNU Engineering Building #301-520
- TA: 정성엽(seongyeop.jeong@snu.ac.kr), 강인재(abcinje@snu.ac.kr)

Course Homepage

- http://csl.snu.ac.kr/sleds

- Lecture slides
- Lab. materials

- ID: lge
- PW: 6060



Outline

- Introduction to Python
- Basic data types
- Input/Output
- Conditionals
- Loops
- Functions

Introduction to Python

Why Python?

- 1st place among the "Top Programming Languages" (IEEE Spectrum, 2019)
- "Fastest growing major programming language" (stackoverflow.com, 2019)
- The 2nd most popular programming language in GitHub (Nov. 2019)
- "The language of AI"

Rank	Language	Type	Score
1	Python		100.0
2	Java		96.3
3	C		94.4
4	C++		87.5
5	R		81.5
6	JavaScript		79.4
7	C#		74.5
8	Matlab		70.6
9	Swift		69.1
10	Go		68.0

The Birth of Python

- Developed by Guido van Rossum in 1990

Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office … would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus).

Python Goals

- "Computer programming for Everybody"
  - DARPA funding proposal

- An easy and intuitive language just as powerful as major competitors
- Open source, so anyone can contribute to its development
- Code that is as understandable as plain English
- Suitability for everyday tasks, allowing for short development times

Program like Plain English?

```
filename = input('Enter file: ')
f = open(filename)

counts = dict()
for line in f:
    words = line.strip().lower().split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1

lst = sorted([(v,k) for k,v in counts.items()], reverse=True)

for v, k in lst[:10]:
    print(v, k)
```

Compare with this:

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>

# define MAX_CHARS 26
# define MAX_WORD_SIZE 30

struct TrieNode
{
    bool isEnd;
    unsigned frequency;
    int indexMinHeap;
    TrieNode* child[MAX_CHARS];
};

struct MinHeapNode
{
    TrieNode* root;
    unsigned frequency;
    char* word;
};

struct MinHeap
{
    unsigned capacity;
    int count;
    MinHeapNode* array;
};

TrieNode* newTrieNode()
{
    TrieNode* trieNode = new TrieNode;

    trieNode->isEnd = 0;
    trieNode->frequency = 0;
    trieNode->indexMinHeap = -1;
    for( int i = 0; i < MAX_CHARS; ++i )
        trieNode->child[i] = NULL;

    return trieNode;
}

MinHeap* createMinHeap( int capacity )
{
    MinHeap* minHeap = new MinHeap;

    minHeap->capacity = capacity;
    minHeap->count = 0;

    minHeap->array = new MinHeapNode [ minHeap->capacity ];

    return minHeap;
}

void swapMinHeapNodes ( MinHeapNode* a, MinHeapNode* b )
{
    MinHeapNode temp = *a;
    *a = *b;
    *b = temp;
}

void minHeapify( MinHeap* minHeap, int idx )
{
    int left, right, smallest;

    left = 2 * idx + 1;
    right = 2 * idx + 2;
    smallest = idx;
                                                    68,1-4        Top
```

```c
    if ( left < minHeap->count &&
        minHeap->array[ left ]. frequency <
        minHeap->array[ smallest ]. frequency
    )
        smallest = left;

    if ( right < minHeap->count &&
        minHeap->array[ right ]. frequency <
        minHeap->array[ smallest ]. frequency
    )
        smallest = right;

    if( smallest != idx )
    {
        minHeap->array[ smallest ]. root->indexMinHeap = idx;
        minHeap->array[ idx ]. root->indexMinHeap = smallest;

        swapMinHeapNodes (&minHeap->array[ smallest ], &minHeap->array[ idx ]);

        minHeapify( minHeap, smallest );
    }
}

void buildMinHeap( MinHeap* minHeap )
{
    int n, i;
    n = minHeap->count - 1;

    for( i = ( n - 1 ) / 2; i >= 0; --i )
        minHeapify( minHeap, i );
}

void insertInMinHeap( MinHeap* minHeap, TrieNode** root, const char* word )
{
    if( (*root)->indexMinHeap != -1 )
    {
        ++( minHeap->array[ (*root)->indexMinHeap ]. frequency );

        minHeapify( minHeap, (*root)->indexMinHeap );
    }

    else if( minHeap->count < minHeap->capacity )
    {
        int count = minHeap->count;
        minHeap->array[ count ]. frequency = (*root)->frequency;
        minHeap->array[ count ]. word = new char [strlen( word ) + 1];
        strcpy( minHeap->array[ count ]. word, word );

        minHeap->array[ count ]. root = *root;
        (*root)->indexMinHeap = minHeap->count;

        ++( minHeap->count );
        buildMinHeap( minHeap );
    }

    else if ( (*root)->frequency > minHeap->array[0]. frequency )
    {

        minHeap->array[ 0 ]. root->indexMinHeap = -1;
        minHeap->array[ 0 ]. root = *root;
        minHeap->array[ 0 ]. root->indexMinHeap = 0;
        minHeap->array[ 0 ]. frequency = (*root)->frequency;

        delete [] minHeap->array[ 0 ]. word;
        minHeap->array[ 0 ]. word = new char [strlen( word ) + 1];
        strcpy( minHeap->array[ 0 ]. word, word );

        minHeapify ( minHeap, 0 );
                                                    136,2-8       50%
```

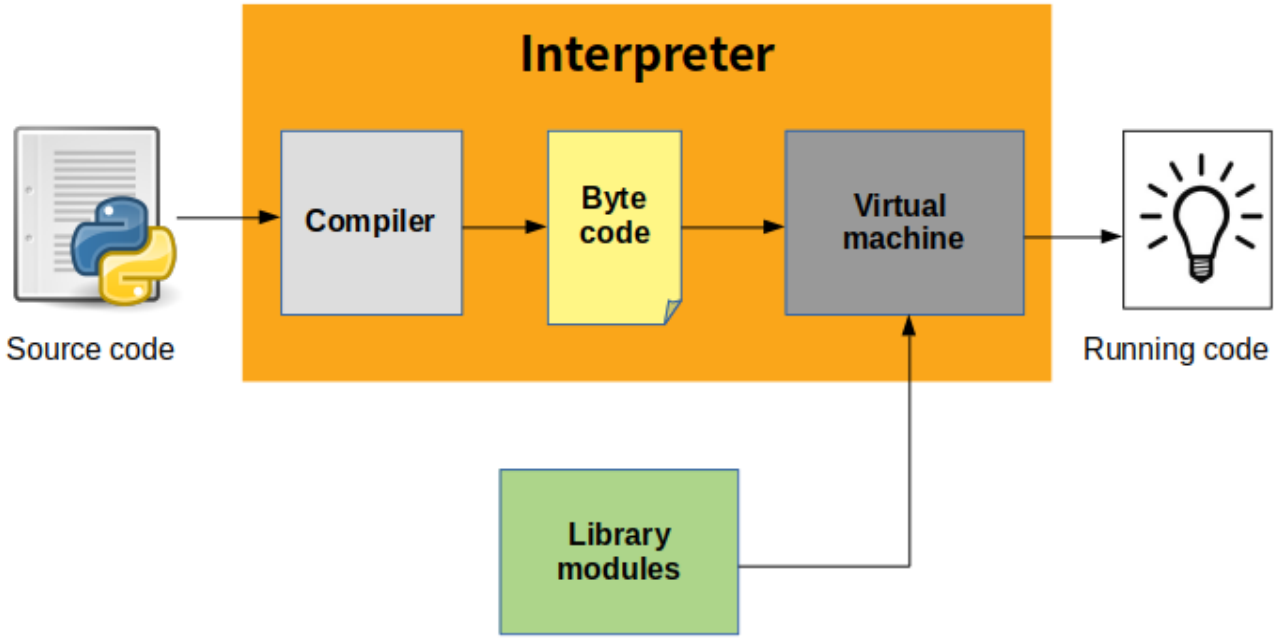
```c
    }
}

void insertUtil ( TrieNode** root, MinHeap* minHeap,
                        const char* word, const char* dupWord )
{
    if ( *root == NULL )
        *root = newTrieNode();

    if ( *word != '\0' )
        insertUtil ( &((*root)->child[ tolower( *word ) - 97 ]),
                        minHeap, word + 1, dupWord );
    else
    {
        if ( (*root)->isEnd )
            ++( (*root)->frequency );
        else
        {
            (*root)->isEnd = 1;
            (*root)->frequency = 1;
        }

        insertInMinHeap( minHeap, root, dupWord );
    }
}


void insertTrieAndHeap(const char *word, TrieNode** root, MinHeap* minHeap)
{
    insertUtil( root, minHeap, word, word );
}

void displayMinHeap( MinHeap* minHeap )
{
    int i;

    for( i = 0; i < minHeap->count; ++i )
    {
        printf( "%s : %d\n", minHeap->array[i].word,
                        minHeap->array[i].frequency );
    }
}

void printKMostFreq( FILE* fp, int k )
{
    MinHeap* minHeap = createMinHeap( k );

    TrieNode* root = NULL;

    char buffer[MAX_WORD_SIZE];

    while( fscanf( fp, "%s", buffer ) != EOF )
        insertTrieAndHeap(buffer, &root, minHeap);

    displayMinHeap( minHeap );
}

int main()
{
    int k = 5;
    FILE *fp = fopen ("test.txt", "r");
    if (fp == NULL)
        printf ("File doesn't exist ");
    else
        printKMostFreq (fp, k);
    return 0;
}
                                                    204,0-1       Bot
```

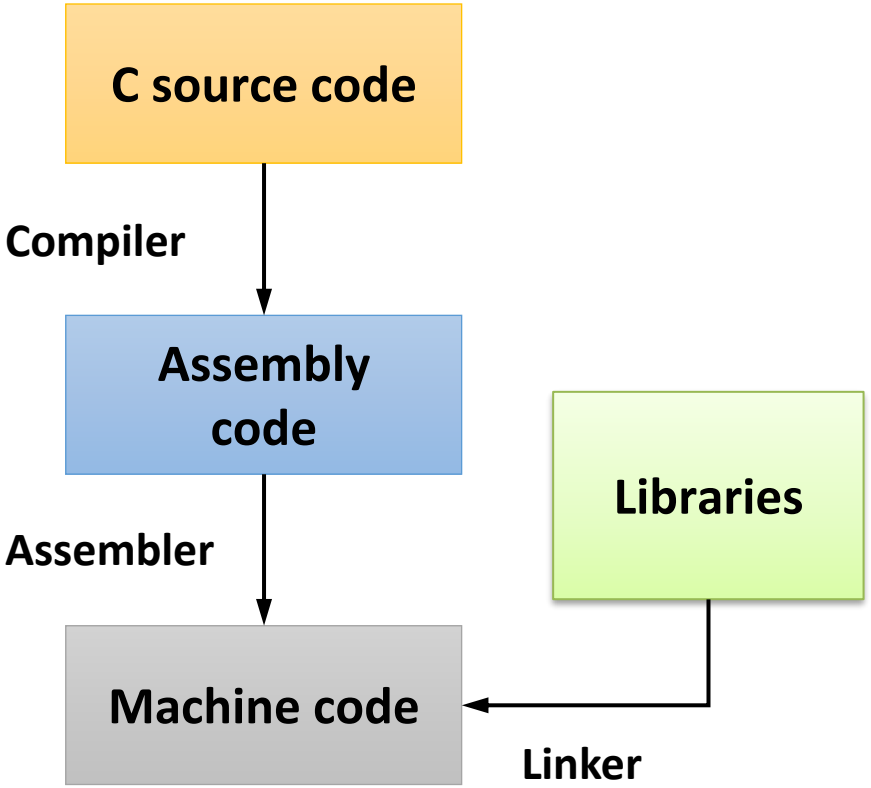
Source: https://www.geeksforgeeks.org/find-the-k-most-frequent-words-from-a-file/

Python Features

- Multi-paradigm platform-independent programming language
  - Structured
  - Object-oriented
  - Functional
  - …
- Interpreted
- Dynamically typed
- Highly extensible
  - Modules can be written in other languages such as C, C++, …
- "Pythonic"

Interpreted vs. Compiled





Python Versions

- Python 1.0 (1990)
- Python 2.0 (2000)
- Python 3.0 (2008) – Not backward compatible to 2.0

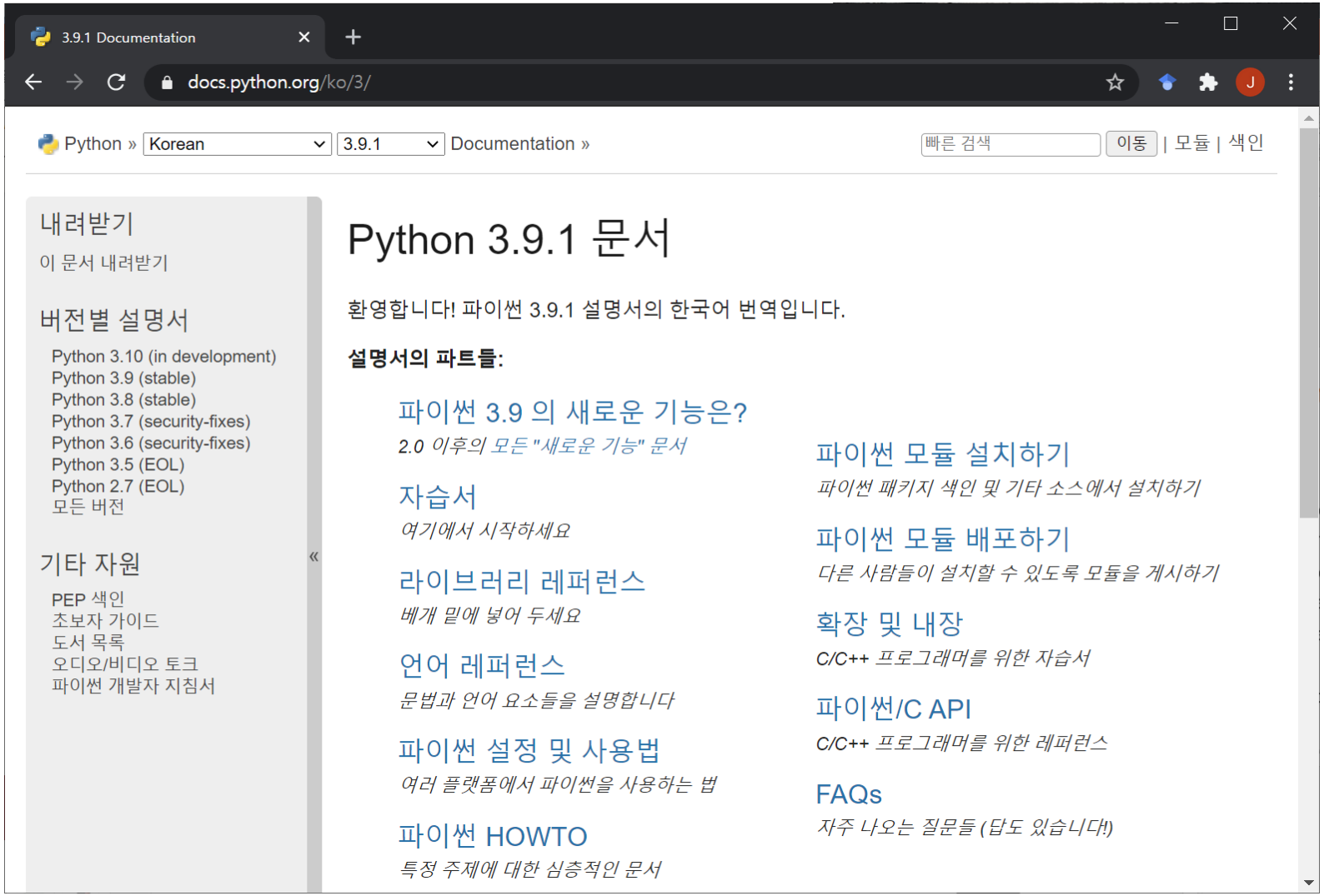
- The latest version: 3.9.1

- Official homepage: https://www.python.org
- Tutorial: https://docs.python.org/ko/3/tutorial

Python Applications

- Machine Learning (TensorFlow, PyTorch, etc.)
- GUI Applications (Kivy, Tkinter, PyQt, etc.)
- Web frameworks (Django used by YouTube, Instagram, Dropbox)
- Image processing (OpenCV, Pillow, etc.)
- Web scraping (Scrapy, BeautifulSoup, etc.)
- Text processing (NLTK, KoNLPy, Word2vec, etc.)
- Test frameworks
- Multimedia
- Scientific computing and many more …

https://docs.python.org



https://stackoverflow.com

how to copy list in pyth

https://www.google.co.kr/search?q=how+to+copy+list+in+python&ie=&oe=

Google

how to copy list in python

All | Images | Videos | News | Maps | More | Settings | Tools

About 154,000,000 results (0.48 seconds)

To actually copy the list, you have various possibilities:

1. You can use the builtin **list.copy**() method (available since **Python** 3.3): new_list = old_list. ...
2. You can slice it: new_list = old_list[:] ...
3. You can use the built in **list**() function: new_list = **list**(old_list)

More items...

How to clone or copy a list? - Stack Overflow
https://stackoverflow.com › questions › how-to-clone-or-copy-a-list

About Featured Snippets | Feedback

Python List copy() - Programiz
https://www.programiz.com › python-programming › methods › list › copy
The **copy**() method returns a shallow **copy** of the **list**.

How to clone or copy a list? - Stack Overflow
https://stackoverflow.com › questions › how-to-clone-or-copy-a-list
16 answers
Oct 18, 2016 - To actually **copy** the **list**, you have various possibilities: You can use the builtin **list.copy**() method (available since **Python** 3.3): new_list = old_list. You can slice it: new_list = old_list[:] You can use the built in **list**() function: new_list = **list**(old_list)

How to deep **copy** a **list**?	7 answers	Jul 29, 2018
Python: Make **Copy** of **List**	2 answers	Feb 14, 2017
python list by value not by reference	11 answers	Jan 5, 2012
Python copy a **list** of **lists**	3 answers	Feb 23, 2015

More results from stackoverflow.com

You've visited this page 3 times. Last visit: 12/5/19

python - How to clone

https://stackoverflow.com/questions/2612802/how-to-clone-or-copy-a-list

stackoverflow | Products | Search... | Log in | Sign up

Home
PUBLIC
Stack Overflow
Tags
Users
Jobs
TEAMS | What's this?
First 25 Users Free

How to clone or copy a list?

Ask Question

Asked 9 years, 8 months ago | Active 12 days ago | Viewed 1.5m times

2301

What are the options to clone or copy a list in Python?

While using `new_list = my_list`, any modifications to `new_list` changes `my_list` everytime. Why is this?

python | list | copy | clone

558

share | improve this question

edited Feb 22 at 12:43
Satya
513 • 2 • 18

asked Apr 10 '10 at 8:49
aF.
50.8k • 39 • 117 • 180

add a comment

16 Answers | active | oldest | votes

3057

With `new_list = my_list`, you don't actually have two lists. The assignment just copies the reference to the list, not the actual list, so both `new_list` and `my_list` refer to the same list after the assignment.

To actually copy the list, you have various possibilities:

Welcome to the World of Spam!

Source: https://www.youtube.com/watch?v=Gxtsa-OvQLA

Basic Data Types

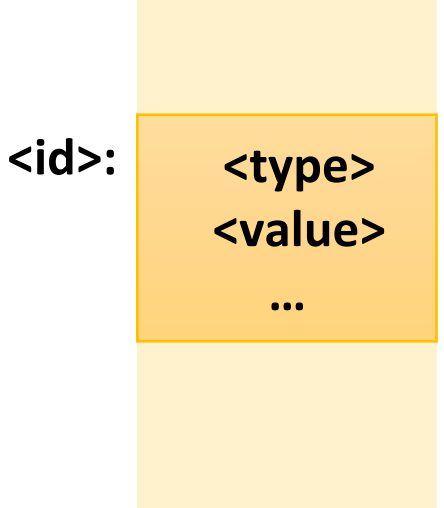
Data Types

- Basic data types
  - Boolean
  - Integer
  - Floating point
  - String
- Container data types
  - List
  - Dictionary
  - Tuple
  - Set
- Libraries
  - array
  - math
  - random
  - urllib
  - …
- User-defined data types (classes)
  - Automobile
  - Monster
  - Pixel
  - …

Object-oriented Data Model

- Objects are Python's abstraction for data

- Each object has:
  - An identity (e.g., memory address) – `id(x)`
  - A type (or class) – `type(x)`
  - A value
  - A reference count – `sys.getrefcount(x)`



- The '`is`' operator compares the identity of two objects
- Objects can be immutable (e.g., numbers, strings, tuples, …)
- Different variables can refer to the same object

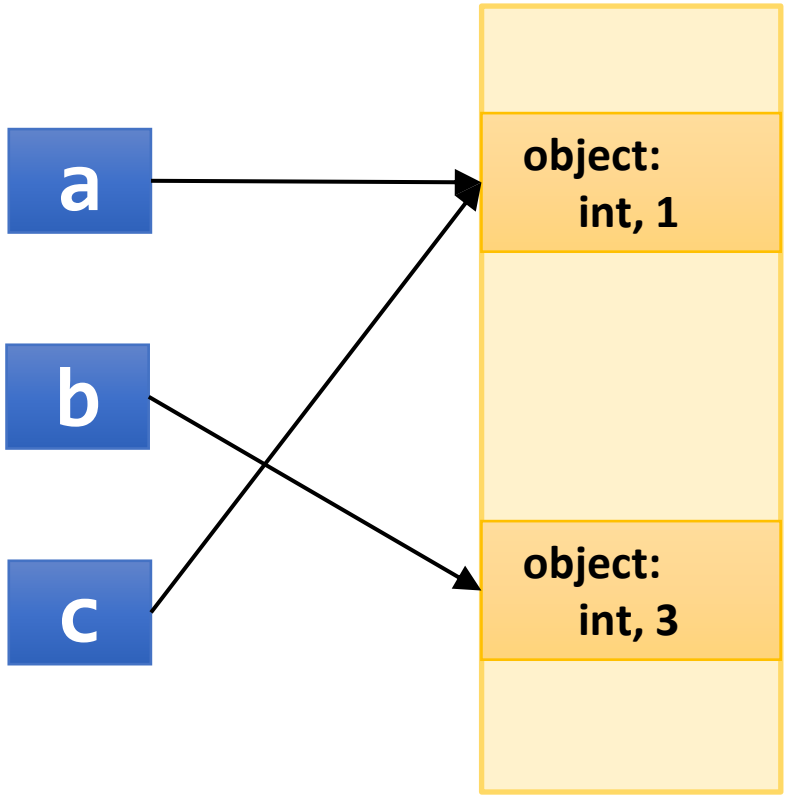
Constants and Variables

- Constant
  - An immutable object with a fixed value (its value cannot be changed)
  - Boolean constants: `True`, `False`
  - Numeric constants: `0, 12, 3.14159`
  - String constants: `'this is a string', "hello"`
- Variable
  - A "name" for an object
  - A variable refers to an object (mutable/immutable)
- Python is a dynamically-typed language
  - Variable names can be bound to different values, possibly of varying types (or classes)

Assignments

- Assignment operator (=) assigns a value (or an object) to a variable

```
>>> a = 1

>>> b = 3

>>> c = a

>>> print(id(a), id(b), id(c))
```



Assignments: Examples

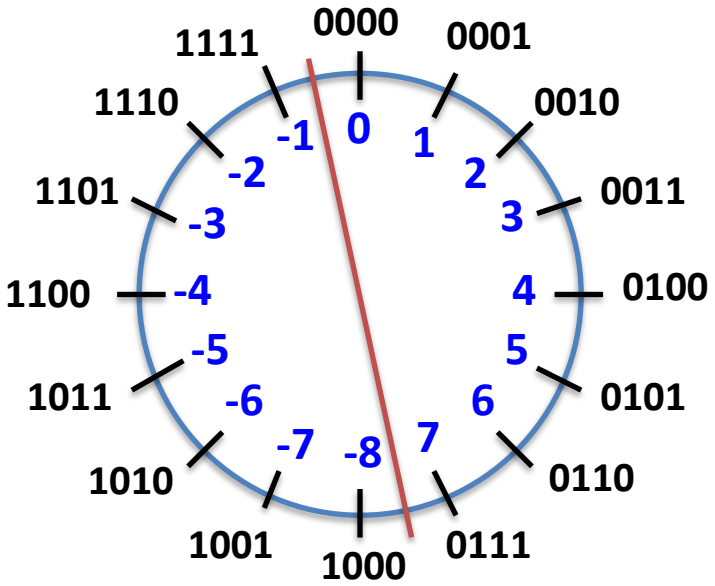
- Variables are used to store temporary values
- In python, a variable actually points to an object
- The same variable can point objects with different types

```
>>> x = 1              # x points to an integer (constant) object
>>> y = 2              # y also points to an integer (constant) object
>>> x + 1              # read the value x and add 1
>>> a = x + 1          # a points to an integer object that has 2
>>> x = x + 1          # x points to an integer object that has 2
>>> b = y              # a points to the same integer object 2
>>> y = "2"            # y now points to a string object
>>> z = x + y          # throws a type mismatch exception (TypeError)
>>> z = x + b          # z now points to an integer object 4
>>> a, b = b, a        # ???
```

Integer

- `int`
  - Integer numbers in an unlimited range
  - Negative numbers are represented in two's complement format

$$O(B) = -b_{w-1} \cdot 2^{w-1} + \left(\sum_{i=0}^{w-2} b_i \cdot 2^i \right)$$



- Some small integers are shared (implementation-specific)

Representing Integer Constants

- An integer constant should start with a non-zero digit (except zero)
- Use prefixes (0b, 0o, 0x) to denote binary/octal/hexadecimal values
- A single underscore('_') can be placed between digits

```
>>> print(1011)
>>> print(0b1011)
>>> print(0o1011)
>>> print(0x1011)
>>> print(10_11)
>>> print(0100)
>>> print(10__11)
```

```
>>> print(2_7_8_9_0)
>>> print(27_890)
>>> print(2_7890)
>>> print(0b0110_1100_1111_0010)
>>> print(0x6cf2)
>>> print(0o66362)
```

Integer: Arithmetic Operations

- Addition: `a + b`
- Subtraction: `a - b`
- Multiplication: `a * b`
- Division: `a / b` → floating-point
- Floor division: `a // b` → integer
- Modulo: `a % b`
- Power: `a ** b`

Integer: Bitwise Operations

- Invert: ~a
- Shift left: a << b
- Shift right: a >> b
- Bitwise AND: a & b
- Bitwise OR: a | b
- Bitwise XOR: a ^ b

Boolean

- `bool`
  - `False` or `True`
  - A subtype of the integer type
  - Boolean values behave like the values `0` and `1`, respectively
  - When converted to a string, `'False'` or `'True'` are returned, respectively

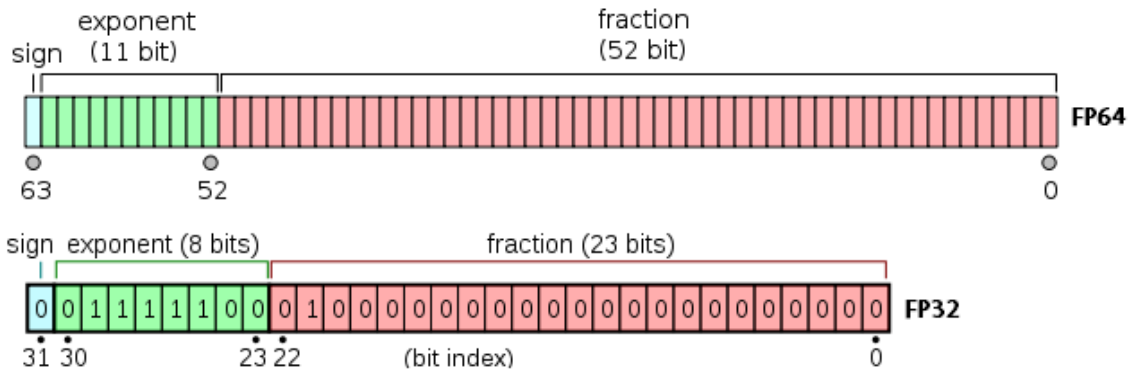
```
>>> t = False
>>> print(t)

>>> a = 100
>>> b = bool(a)
>>> print(a, b)
```

```
>>> t = True
>>> f = False
>>> x = 10
>>> print(x + t)
>>> print(t * f)
>>> print(True == 1)
```

Floating Point

- `float`
  - Python only supports double-precision floating point numbers
  - The benefit of supporting single-precision is not that great due to the overhead of using objects in Python

- IEEE754 floating point representation standard
  - Double precision: 64-bit ($4.9 \times 10^{-324} \sim 1.8 \times 10^{308}$)
  - Single precision: 32-bit ($1.4 \times 10^{-45} \sim 3.4 \times 10^{38}$)



Representing FP Constants

- Represented with or without exponent
- Integer and exponent parts are always interpreted using radix 10
- A single underscore ('_') can be placed between digits

```
>>> print(3.14)
>>> print(10.)
>>> print(0.001)
>>> print(1e100)
>>> print(3.14e-10)
>>> print(0e0)
```

```
>>> print(3.14_15_92)
>>> print(1_234.005_694)
>>> print(e100)
>>> print(0b1000.0011)
>>> print(0o1234.56)
>>> print(0xdead.beef)
```

Floating Point Operations

- Addition: a + b
- Subtraction: a – b
- Multiplication: a * b
- Division: a / b

Floating Point Example

```
>>> pi = 3.14159
>>> print(pi)
>>> print(2*pi)

>>> d = 0.1
>>> print(d+d+d+d+d+d+d+d+d+d)

>>> VeryLarge = 1e20
>>> x = (pi + VeryLarge) - VeryLarge
>>> y = pi + (VeryLarge – VeryLarge)
>>> print(x, y)
```

math: Mathematical Functions

- `import math`
- `math.exp(x)`: e^x
- `math.log(x)`: $\log_e x$
- `math.log10(x)`: $\log_{10} x$
- `math.log(x,b)`: $\log_b x$
- `math.pow(x,y)`: x^y
- `math.sqrt(x)`: $\sqrt{x}$
- `math.sin(x)`: $\sin x$
- `math.pi`: π
- …

random: Random Number Generators

- `import random`
- `random.random()`: generate a number between [0, 1)
- `random.randint(a,b)`: generate a number (integer) between [a, b]
- `random.seed(a=None)`: seed the random number generator
- `random.choice(seq)`: choose an element from the sequence

```
>>> import random
>>> random.random()
0.6145831050305076
>>> random.randint(0, 10)
4
>>> random.choice(['가위', '바위', '보'])
'보'
```

String

- `str`
  - A sequence of characters
  - Python 3 natively supports Unicode characters (even in identifiers)
  - No difference in single (e.g., `'hello'`) or double-quoted strings (e.g., `"hello"`)
  - You can use raw strings by adding an `r` before the first quote

```
>>> print('I\'m your father')
>>> print("Where is 'spam'?")
>>> s1 = "What is the"
>>> s2 = 'spam'
>>> print(s1 + s2)
>>> print(len(s1))
```

```
>>> 이름 = '홍길동'
>>> print("안녕" , 이름)
>>> print("안녕" + 이름)
>>> print("안녕\n"+이름)

>>> print(r'C:\abc\name')
```

Concatenating/Replicating Strings

- *str1* + *str2* : create a new string by adding two existing strings together
- Two or more string literals are automatically concatenated
- *str* * *n*: create a new string by replicating the original string *n* times

```
>>> s1 = 'hello'
>>> s2 = 'world'
>>> s = s1 + s2
>>> print(s)
helloworld
>>> print('hello''world''!')
helloworld!
```

```
>>> s1 = 'hello'
>>> s2 = 'world'
helloworld
>>> print(s1*3 + s2)
hellohellohelloworld
>>> print('-'*30)
------------------------------
```

Type Conversion

- int(), float(), str()
- hex(), oct(), bin()
- ord(), chr()

```
>>> int(3.14)
>>> int('3.14')
>>> int(True)
>>> int('0xcafe')
>>> int('cafe', 16)
>>> int('0xcafe', 0)
```

```
>>> float(3)
>>> float('  -3.14\n')
>>> float('1e10')

>>> str(2020)
>>> str(0xcafe)
>>> str(3.141592)

>>> hex(2020)
>>> oct(2020)
>>> bin(2020)
>>> ord('a')
>>> chr(100)
```

Input/Output

Printing Output

- String formatting
  - %d: integer
  - %f: floating-point
  - %s: string

```
>>> m = 10; g = 3.14; t = 'hello'
>>> print('m = ', m, ', g = ', g)
m = 10 , g = 3.14

>>> print('%d: this is an integer' % m)
10: this is an integer

>>> print('m = %d, g = %f' % (m, g))
m = 10, g = 3.140000

>>> print('%s\t\tm: %d\n\t\tg: %f' % (t, m, g))
hello           m: 10
                g: 3.140000
```

Getting User Input

- `input(prompt)`
  - If a prompt argument is present, it is written to standard output
  - Then, reads a line from input, converting it to a string (stripping a trailing newline)

```
>>> name = input('Your name: ')
Your name: Spam
>>> age = input('Your age: ')
Your age: 20
>>> print('Hello,', name)
Hello, Spam
>>> print('You will be', int(age)+1, 'next year!')
You will be 21 next year!
```

Comments

- Single-line comments

```
# This is a comment
```

- Multi-line comments

```
# These
# are all
# comments
```

```
"""
These are
all
comments
"""
```

```
'''
These are
all
comments
'''
```

- A newline terminates each statement. However, you can use a semicolon to separate statements on the same line.

```
a = 3; print(a)
```

Getting Help

- help()
- help(class)
- help(class.func)
- help(object.func)
- …

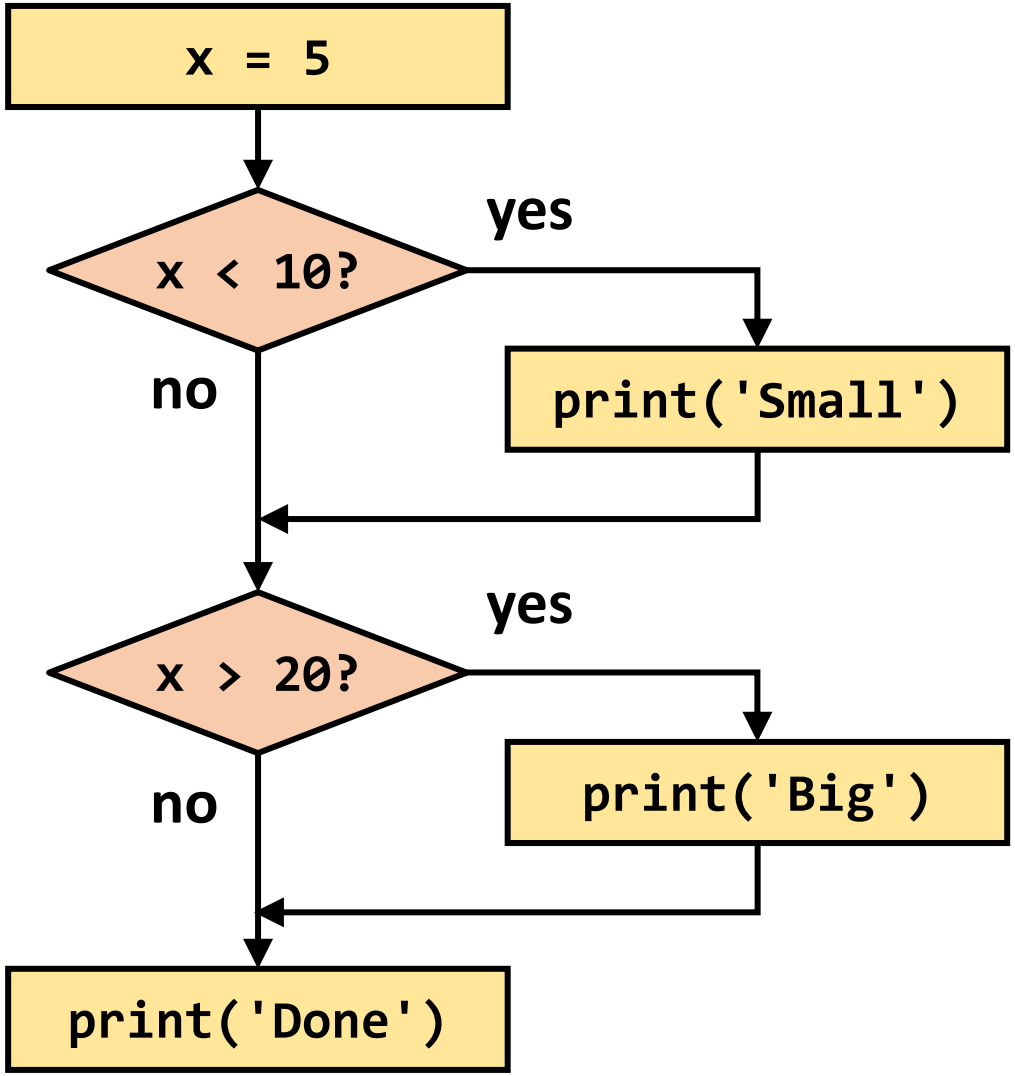
- dir(class)
- dir(object)

```
Help on class int in module builtins:

class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __and__(self, value, /)
 |      Return self&value.
 |
```

Conditionals

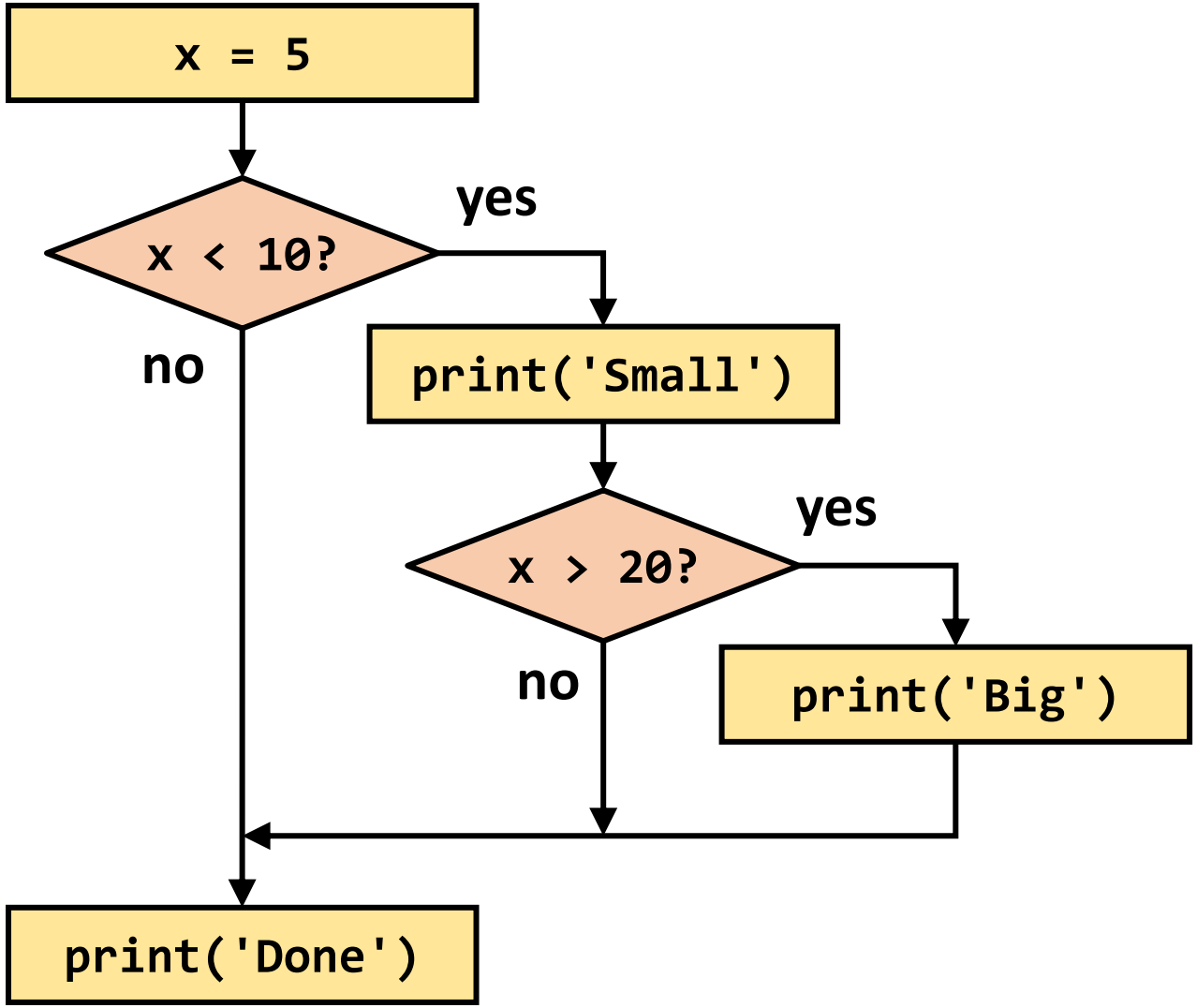
Conditional Steps

```
x = 5
if x < 10:
    print('Small')
if x > 20:
    print('Big')
print('Done')
```



Indentation Matters!

```
x = 5
if x < 10:
    print('Small')
    if x > 20:
        print('Big')
print('Done')
```



Indentation

- Python does not use curly braces { } to indicate a block of statements
- Increase indent after an `if`, `elif`, `else`, `for`, `while` etc. statement
- Maintain indent to indicate the scope of the block
- Reduce indent back to indicate the end of the block
- Blank / comment lines are ignored
- Turn off tabs! (turn tabs into spaces)

```
if a < b:
    a = a + b
    print('here')
else:
    a = a - b
    print('there')
print('done')
```

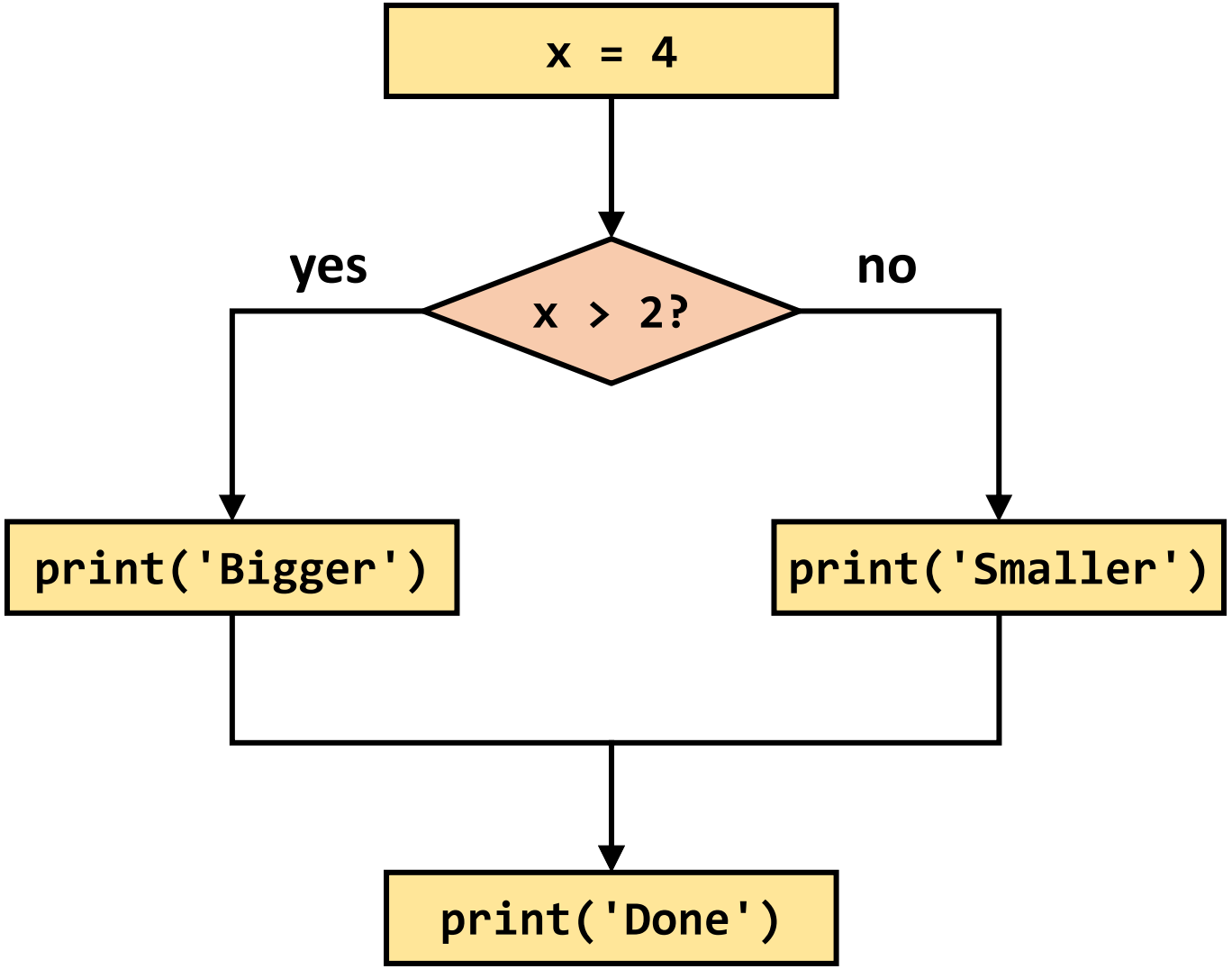
Evaluating Conditions

- Boolean expressions using comparison operators evaluate to `True` or `False`
- Several Boolean expressions can be combined using logical `and` / `or` / `not` operators
- Comparison operators do not change the variables

Notation	Meaning
`a < b`	True if **a** is less than **b**
`a <= b`	True if **a** is less than or equal to **b**
`a == b`	True if **a** is equal to **b**
`a != b`	True if **a** is not equal to **b**
`a >= b`	True if **a** is greater than or equal to **b**
`a > b`	True if **a** is greater than **b**
`A and B`	True if both **A** and **B** are True
`A or B`	True if either **A** or **B** (or both) is True
`not A`	True if **A** is False
`a is b`	True if **a** and **b** point to the same object
`a is not b`	True if **a** and **b** point to the different object
`a in b`	True if **a** is in the sequence **b**
`a not in b`	True if **a** is not in the sequence **b**

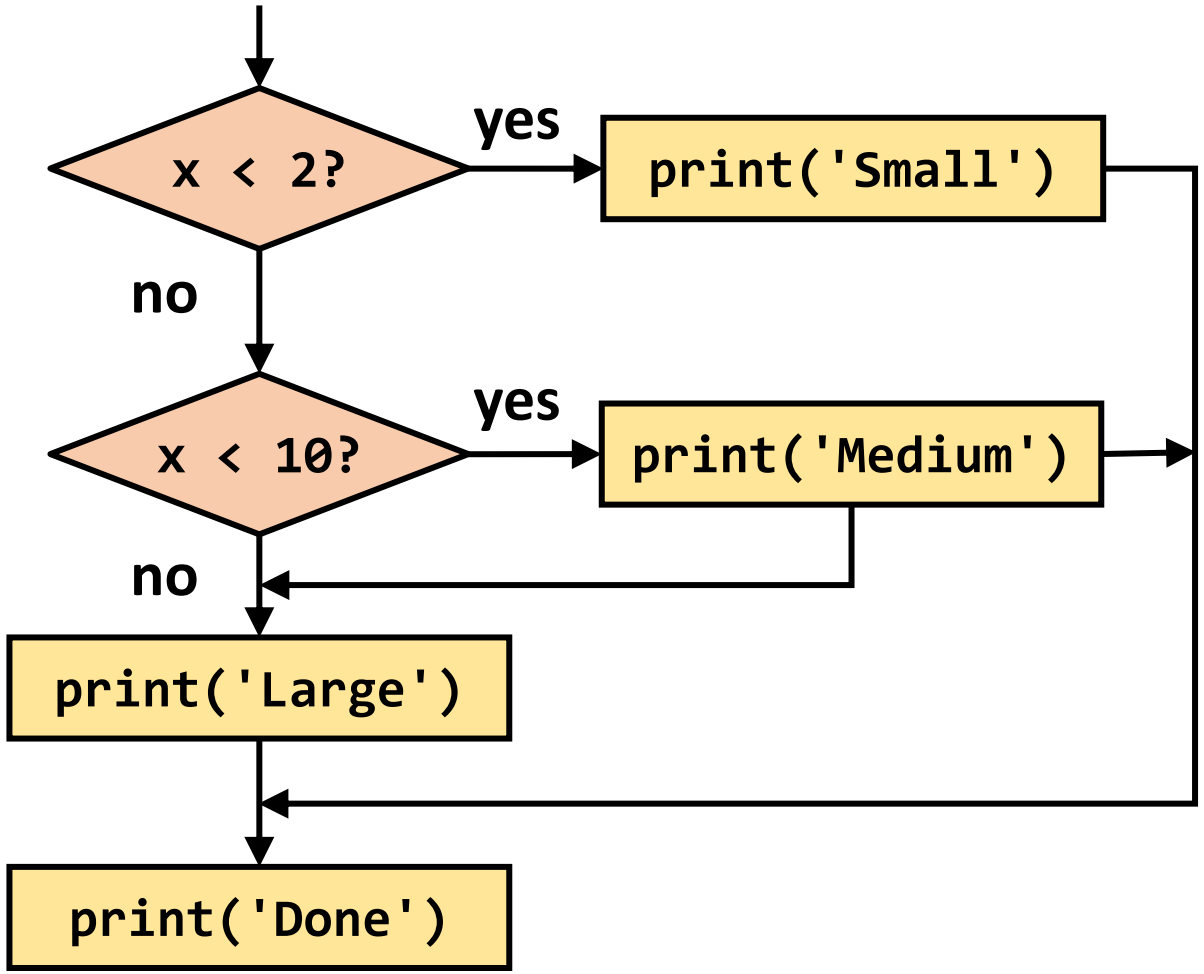
Two-way Decisions

```
x = 4
if x > 2:
    print('Bigger')
else:
    print('Smaller')
print('Done')
```



Multi-way Decisions

```
if x < 2:
    print('Small')
elif x < 10:
    print('Medium')
else:
    print('Large')
print('Done')
```



Multi-way Puzzles

- What's wrong with these programs?

```
if x < 2:
    print('Below 2')
elif x >= 2:
    print('Two or more')
else:
    print('Something else')
```

```
if x < 2:
    print('Below 2')
elif x < 20:
    print('Below 20')
elif x < 10:
    print('Below 10')
else:
    print('Something else')
```

Conditional Expression

```
if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
elif score >= 70:
    grade = 'C'
elif score >= 60:
    grade = 'D'
else:
    grade = 'F'
```

```
grade = 'A' if score >= 90 else \
        'B' if score >= 80 else \
        'C' if score >= 70 else \
        'D' if score >= 60 else \
        'F'
```

Exceptions

- Errors detected during execution even if a statement or expression is syntactically correct
  - `ZeroDivisionError`
  - `NameError`
  - `TypeError`
  - `ValueError`
  - `IndexError`…

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
>>> int('what')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with
base 10: 'what'
```

Handling Exceptions

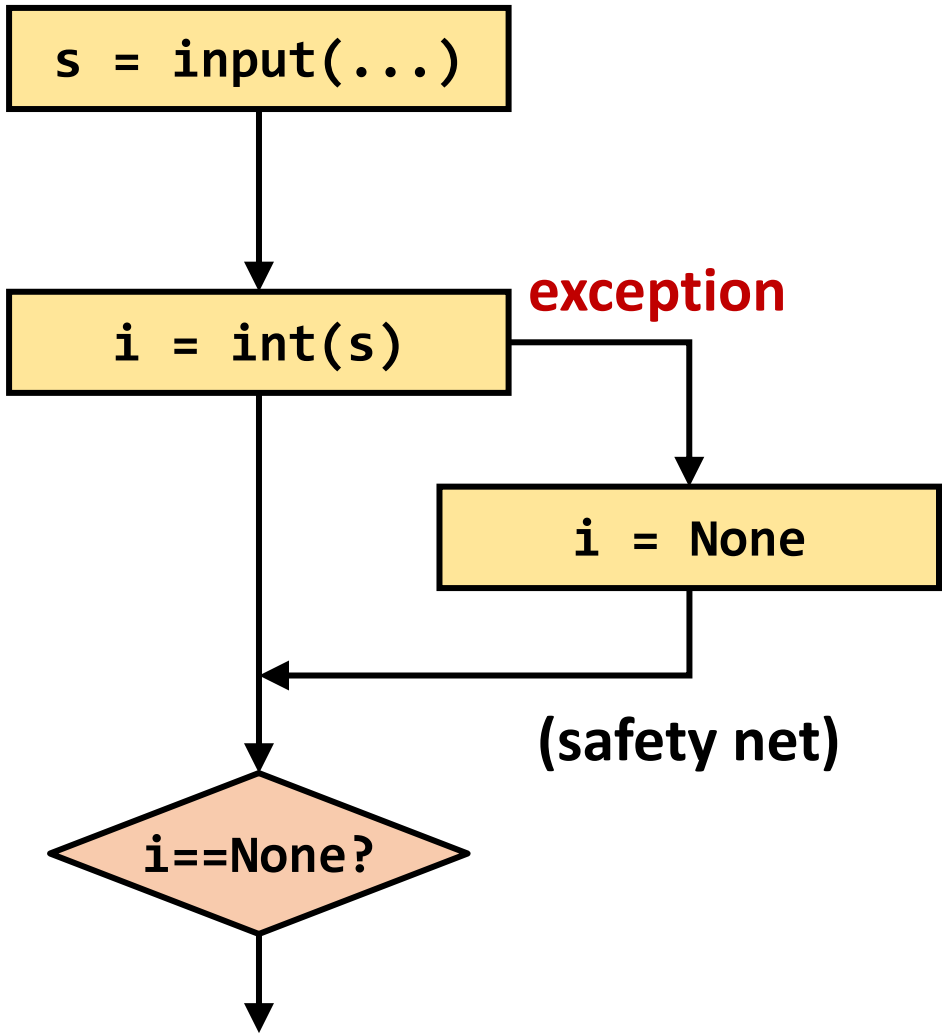
- Surround a dangerous section of code with `try` and `except`
- If the code in the `try` works – the `except` is skipped
- If the code in the `try` fails – it jumps to the `except` code block

```
x = int(input('Enter a number: '))
x1 = x + 1
print(x, '+ 1 =', x1)
```

```
while True:
    try:
        x = int(input('Enter a number: '))
        break
    except:    # catch all exceptions
        print('Oops, try again...')
x1 = x + 1
print(x, '+ 1 =', x1)
```

Example

```
s = input('Enter a number: ')
try:
    i = int(s)
except:
    i = None

if i is None:
    print('Not a number')
else:
    print('Nice work')
```



Loops

Loops

- `while` loop
  - Keep running the loop body while expression is `True`

```
while (expression):
    <statement_1>
    <statement_2>
        ...
    <statement_n>
```

- `for` loop
  - Run the loop body for the specified range

```
for <element> in <object>:
    <statement_1>
    <statement_2>
        ...
    <statement_n>
```

Loops Example

- `while` loop
  - Keep running the loop body while expression is `True`
- `for` loop
  - Run the loop body for the specified range

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

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

While vs. For

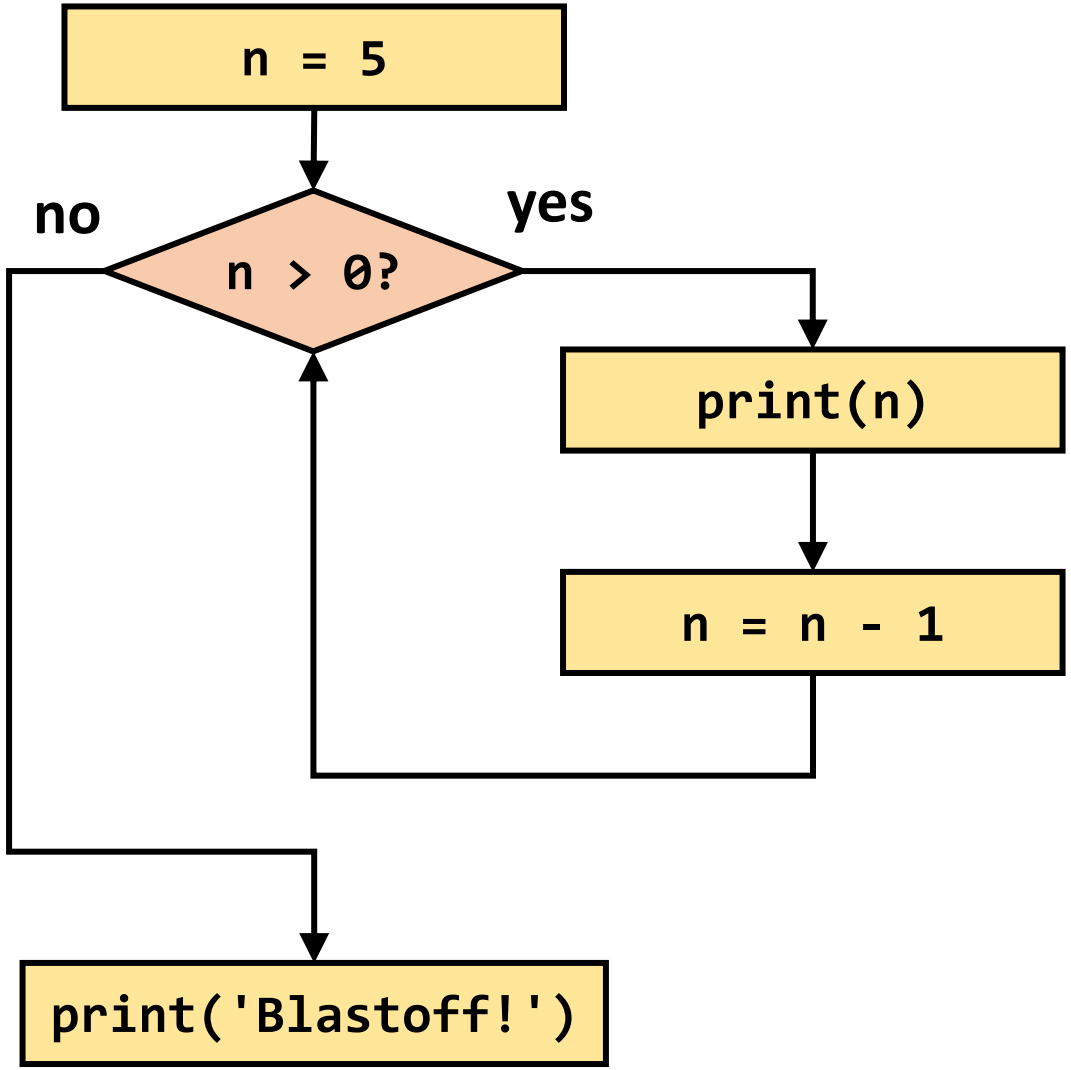
- Indefinite loop – `while`
  - `while` is natural to loop an indeterminate number of times until a logical condition becomes False

- Definite loop – `for`
  - `for` is natural to loop through a list, characters in a string, etc. (anything of determinate size)
  - Run the loop once for each of the items

Indefinite Loop with while

- Indefinite loops have iteration variables that change each time through a loop

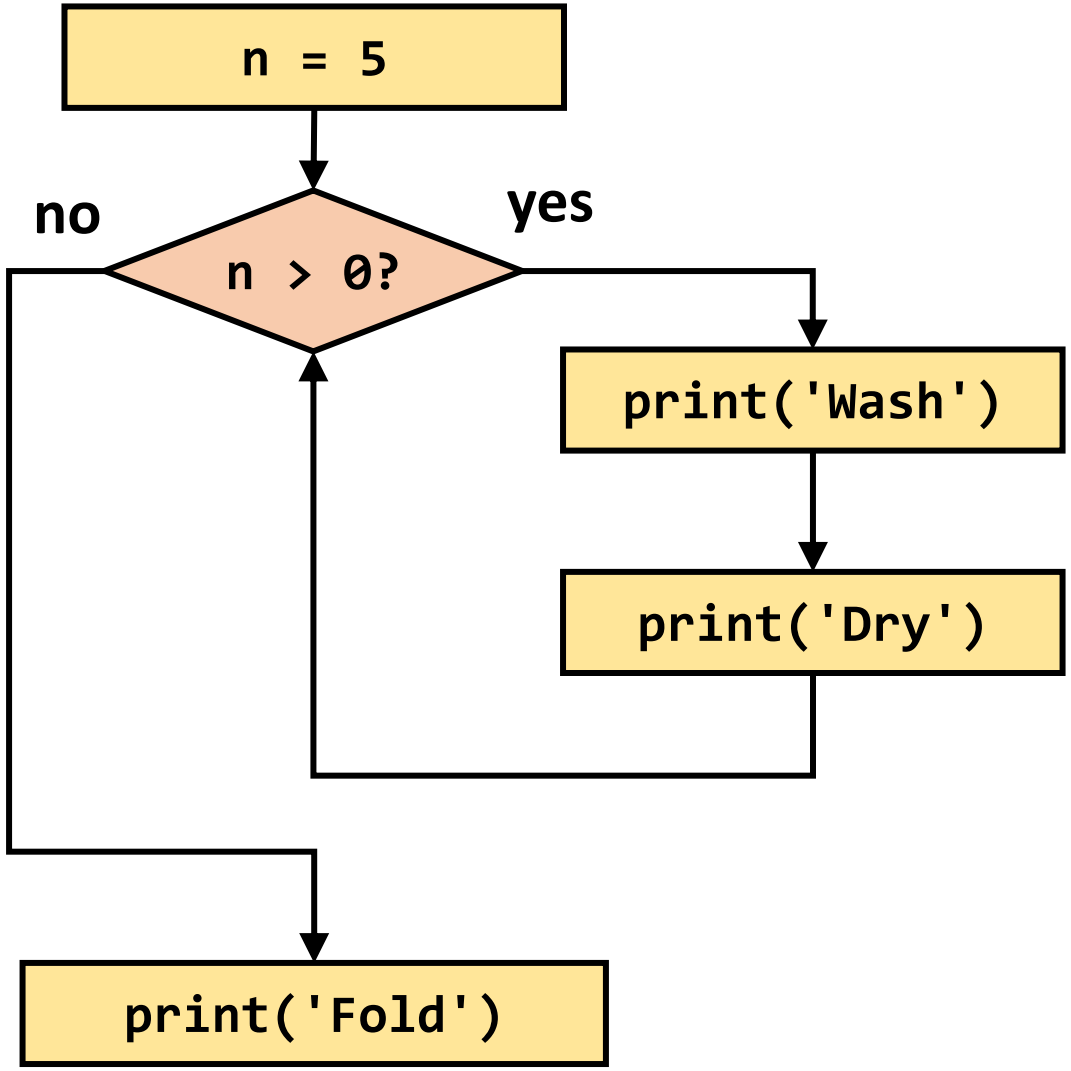
```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('Blastoff!')
```



An Infinite Loop

- What's wrong with this loop?

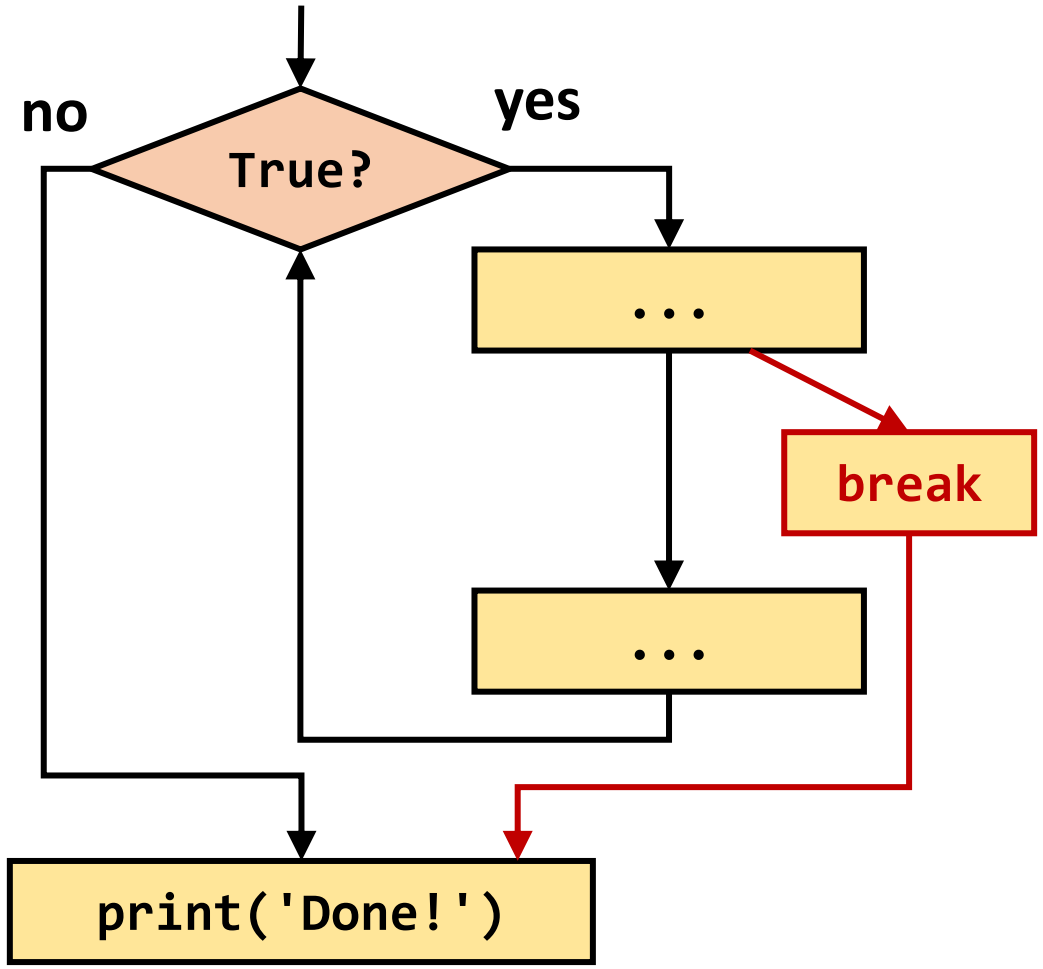
```
n = 5
while n > 0:
    print('Wash')
    print('Dry')
print('Fold')
```



break: Breaking Out of a Loop

- The `break` statement ends the current loop and jumps to the statement immediately following the loop

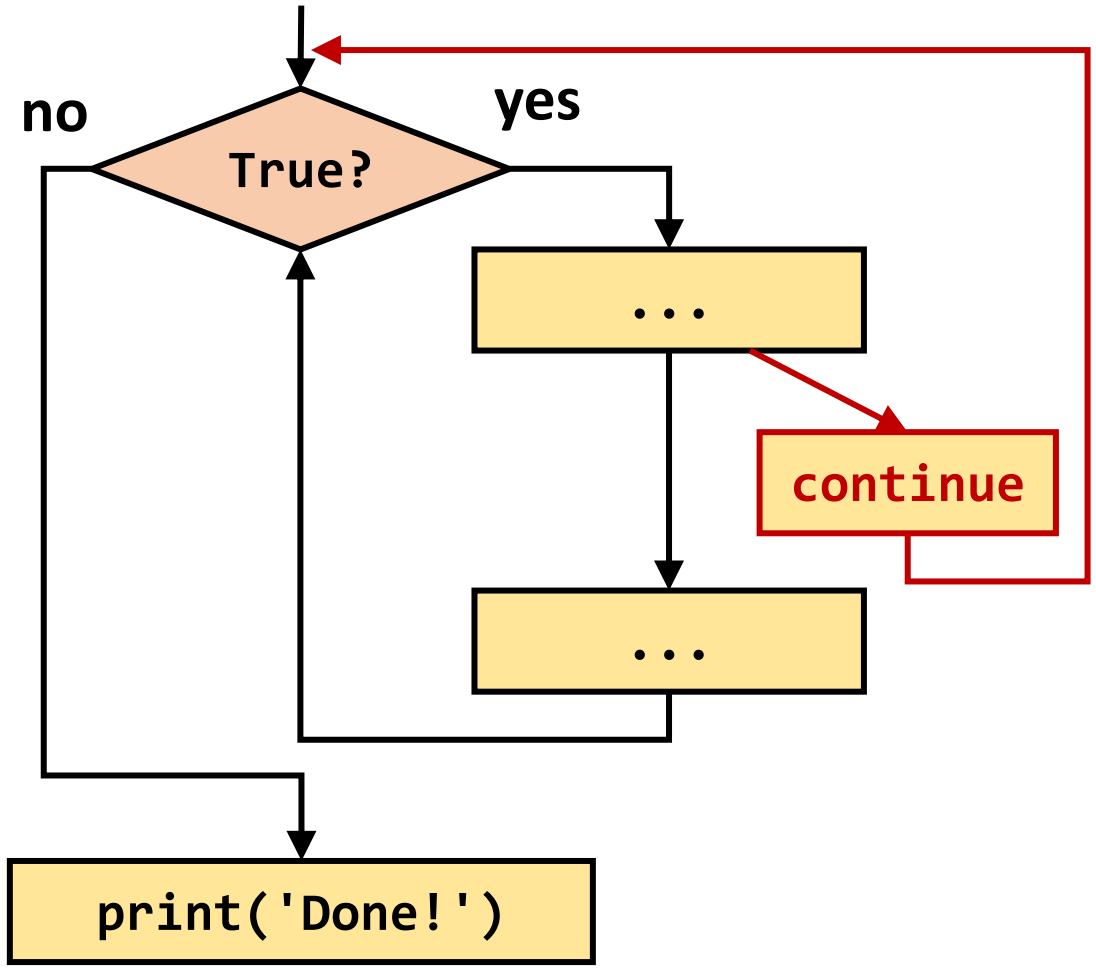
```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)
print('Done!')
```



continue: Finishing an Iteration

- The continue statement ends the current iteration and jumps to the top of the loop and starts the next iteration

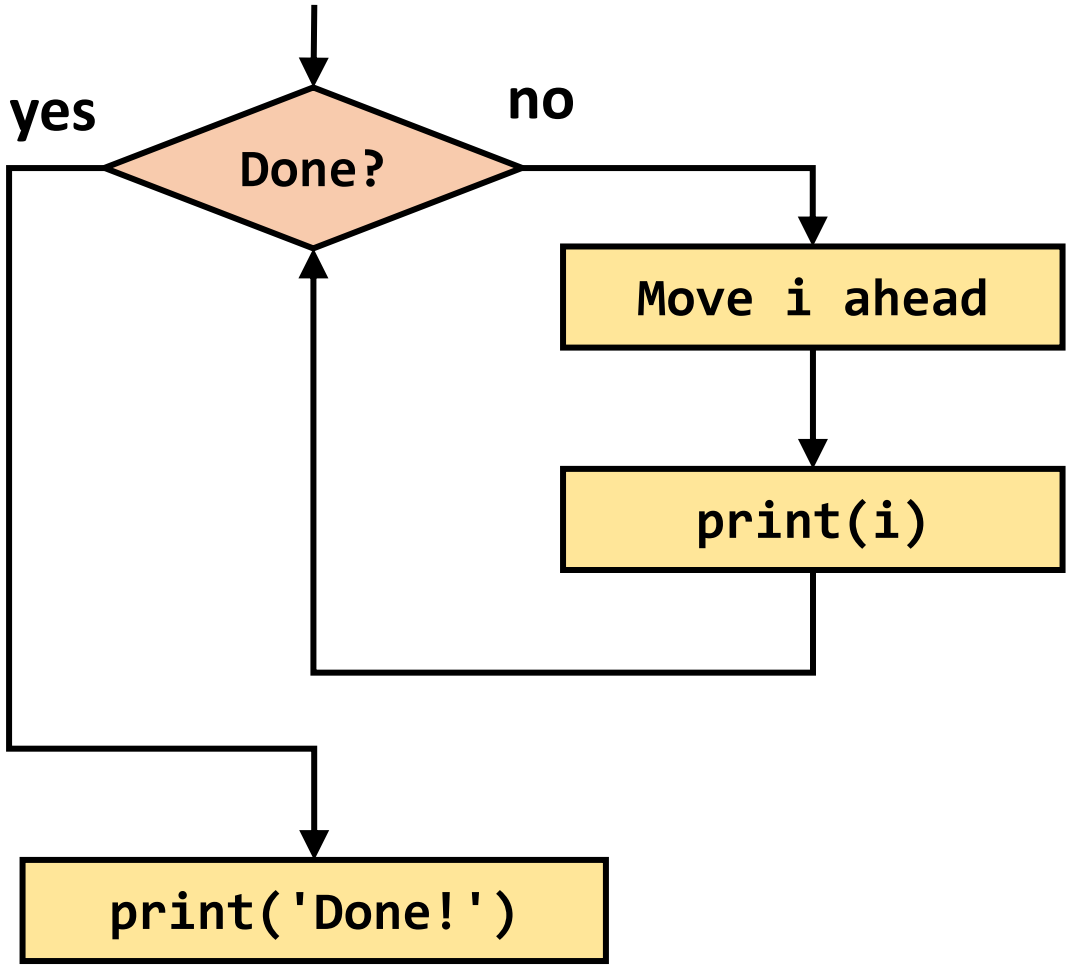
```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')
```



Definite Loop with for

- Definite loops have explicit iteration variables that change each time through a loop

```
for i in range(5):
    print(i)
print('Done!')
```



Specifying an Integer Range

- `range([start,] stop[, step])`
  - Represents an immutable sequence of numbers
  - If the `step` argument is omitted, it defaults to 1 (`step` should not be zero)
  - If the `start` argument is omitted, it defaults to 0

```
range(5)              # 0, 1, 2, 3, 4
range(-1, 4)          # -1, 0, 1, 2, 3
range(0,10,2)         # 0, 2, 4, 6, 8
range(5,0,-1)         # 5, 4, 3, 2, 1
range(10,2)           # ???
```

- `list(range(100))` → [0, 1, 2, …, 99]

Looping Through a List (1)

```
print('Prime numbers')
for p in [2, 3, 5, 7, 11, 13, 17, 19]:
    print(p)
```

```
for name in ['Liam', 'Noah', 'William', 'James']:
     print('Hi,', name)
```

Looping Through a List (2)

```
friends = ['Harry', 'Sally', 'Tom', 'Jerry']

for friend in friends:
    print('Merry Christmas,', friend)

for i in range(len(friends)):
    print('Merry Christmas,', friends[i])
```

Summing in a Loop

```
sum = 0
print('Before', sum)
for n in [24, 12, 4, 19, 31, 27]:
    sum = sum + n
    print(n, sum)
print('After', sum)
```

```
Before 0
24 24
12 36
4 40
19 59
31 90
27 117
After 117
```

Finding the Largest Value?

Functions

Python Functions

- A function is some reusable code that takes argument(s) as input, does some computation, and then returns a result or results.

- Built-in functions
  - Provided as part of Python
  - `print()`, `input()`, `type()`, `float()`, `int()`, `max()`, …

- User-defined functions
  - Functions that we define ourselves and then use
  - A function can be defined using the `def` reserved word
  - A function is called (or invoked) by using the function name, parentheses, and arguments in an expression

Function Example

Function name

Argument

```
big = max([4, 1, 9, 0])
```

Assignment

9

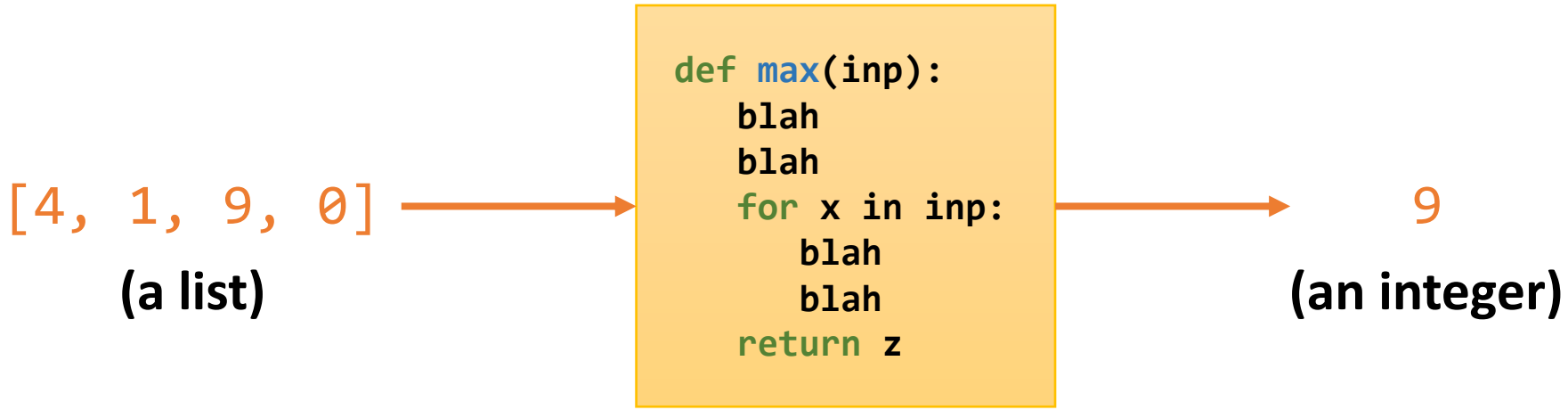
Result

```
>>> big = max([4, 1, 9, 0])
>>> print(big)
9
>>> tiny = min([4, 1, 9, 0])
>>> print(tiny)
0
```

max()

- A function is some stored code that we use
- A function takes some input and produces an output

```
>>> big = max([4, 1, 9, 0])
>>> print(big)
9
```



Building Our Own Functions

- We create a new function using the `def` keyword followed by optional parameters in parentheses
- We indent the body of the function
- This defines the function but does not execute the body of the function

```
def print_lyrics():
    print('I bless the day I found you')
    print('I want to stay around you')
```

Parameters

- A parameter is a variable which we use in the function definition.

- It is a "handle" that allows the code in the function to access the arguments for a particular function invocation.

```
>>> def greet(lang):
...     if lang == 'kr':
...         print('안녕하세요')
...     elif lang == 'fr':
...         print('Bonjour')
...     elif lang == 'es':
...         print('Hola')
...     else:
...         print('Hello')

>>> greet('en')
Hello
>>> greet('es')
Hola
>>> greet('kr')
안녕하세요
>>>
```

Return Value

- A "fruitful" function is one that produces a result (or return value)

- The `return` statement ends the function execution and "sends back" the result of the function

```
>>> def greet(lang):
...     if lang == 'kr':
...         return '안녕하세요'
...     elif lang == 'fr':
...         return 'Bonjour'
...     elif lang == 'es':
...         return 'Hola'
...     else:
...         return 'Hello'

>>> print(greet('en'), 'Jack')
Hello Jack
>>> print(greet('es'), 'Sally')
Hola Sally
>>> print(greet('kr'), '홍길동')
안녕하세요 홍길동
>>>
```

Multiple Parameters / Arguments

- We can define more than one parameter in the function definition
- We simply add more arguments when we call the function
- We match the number and order of arguments and parameters

```
def mymax(a, b):
    if a > b:
        return a
    else:
        return b

x = mymax(3, 5)
print(x)
5
```

Default and Keyword Arguments

- Default arguments
  - You can specify default values for arguments that aren't passed
- Keyword arguments
  - Callers can specify which argument in the function to receive a value by using the argument's name in the call

```
def student(name, id='00000', dept='CSE'):
    print(name, id, dept)

student('John')
student('John', '00001')
student(name='John')
student(id='20191', dept='EE', name='Jack')
```

Arbitrary Arguments

- For functions that take any number of arguments
- Zero or more normal arguments may appear before the variable number of arguments,
- All the arbitrary arguments are collected and transferred using a tuple

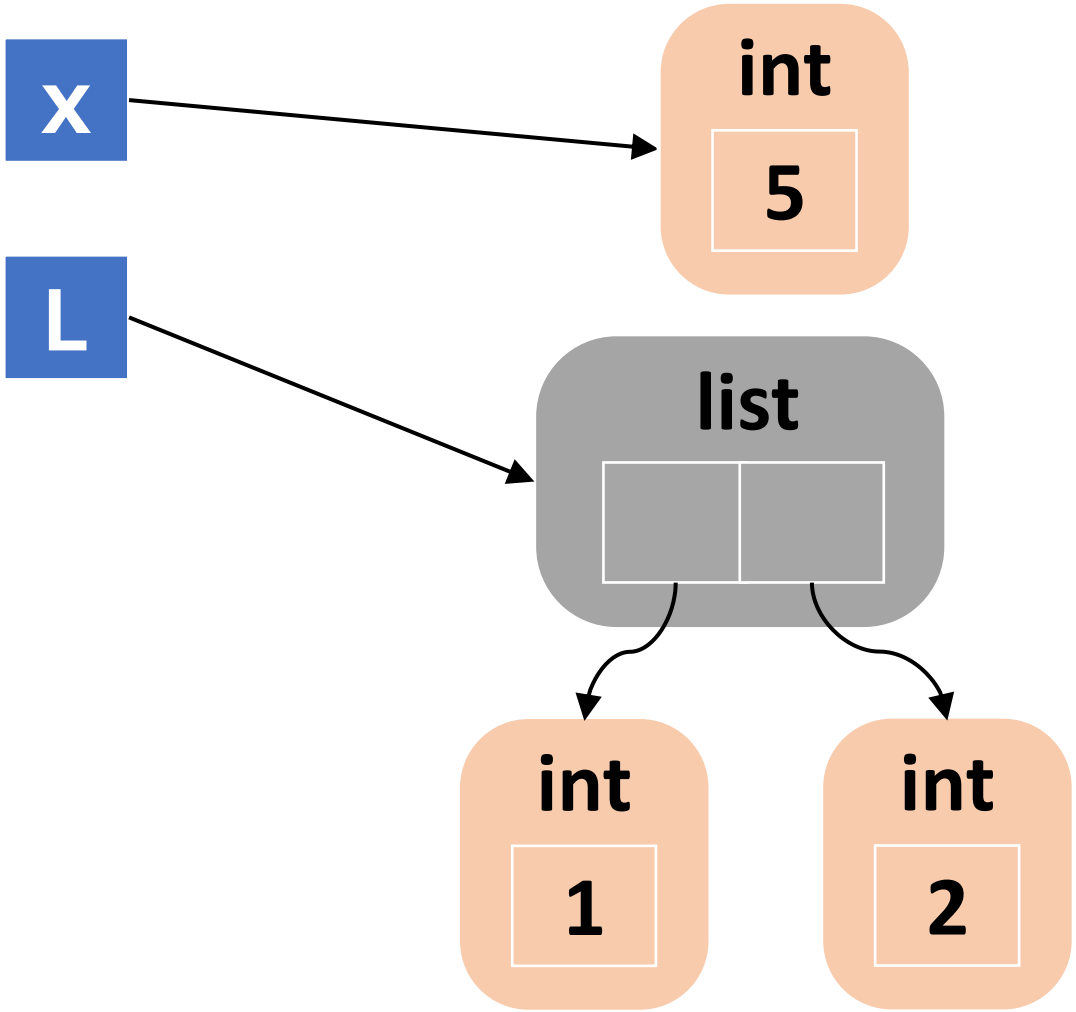
```
def food(name, *likes):
    print(name, 'likes ', end='')
    for d in likes:
        print(d, end=' ')
    print()

food('John', 'spam', 'egg', 'bacon')
```

Passing Arguments

```
>>> def f(a, b):
...     a = 9
...     b[0] = 'spam'

>>> x = 5
>>> L = [1, 2]
>>> f(x, L)
>>> print(x)
5
>>> print(L)
['spam', 2]
```

Passing Arguments

```
>>> def f(a, b):
...     a = 9
...     b[0] = 'spam'

>>> x = 5
>>> L = [1, 2]
```

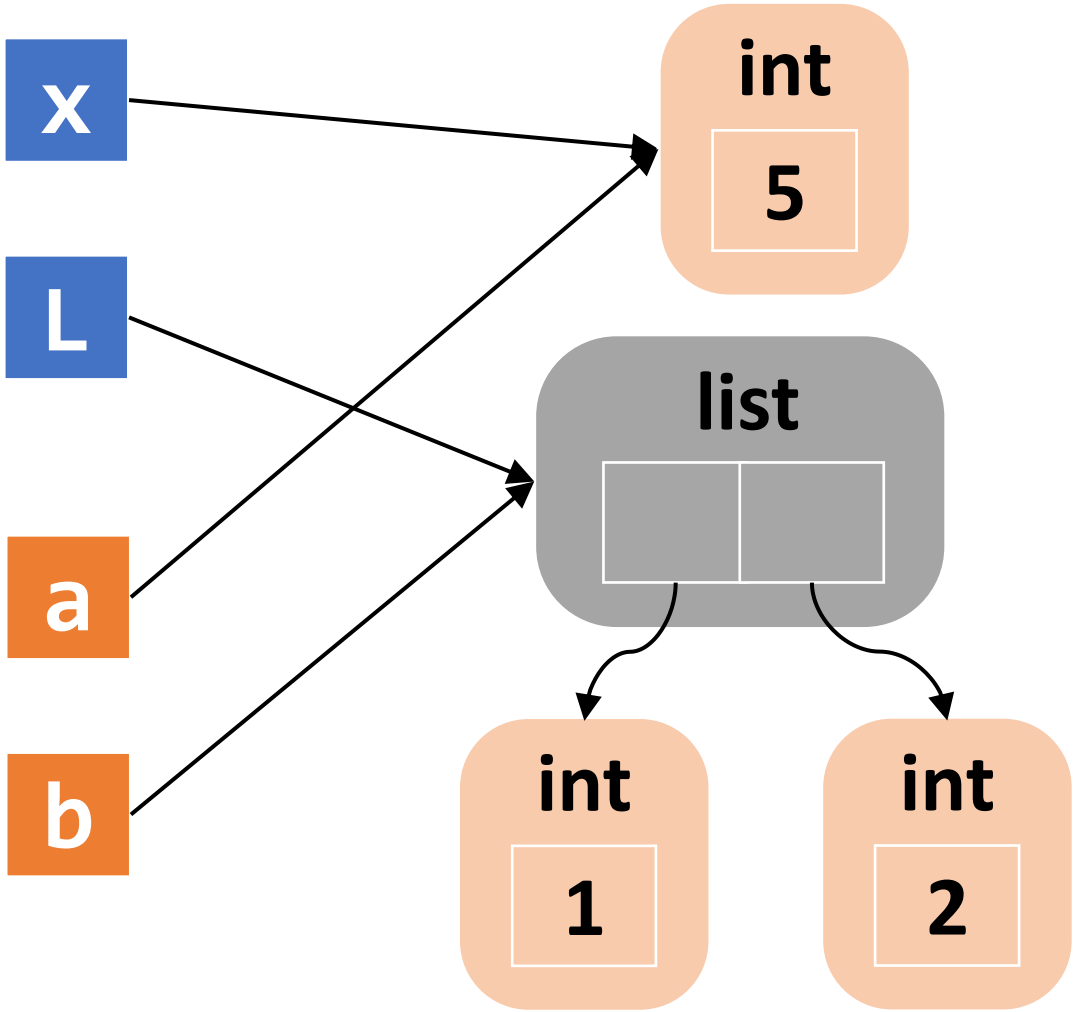


Passing Arguments

```
>>> def f(a, b):
...     a = 9
...     b[0] = 'spam'

>>> x = 5
>>> L = [1, 2]
>>> f(x, L)
```

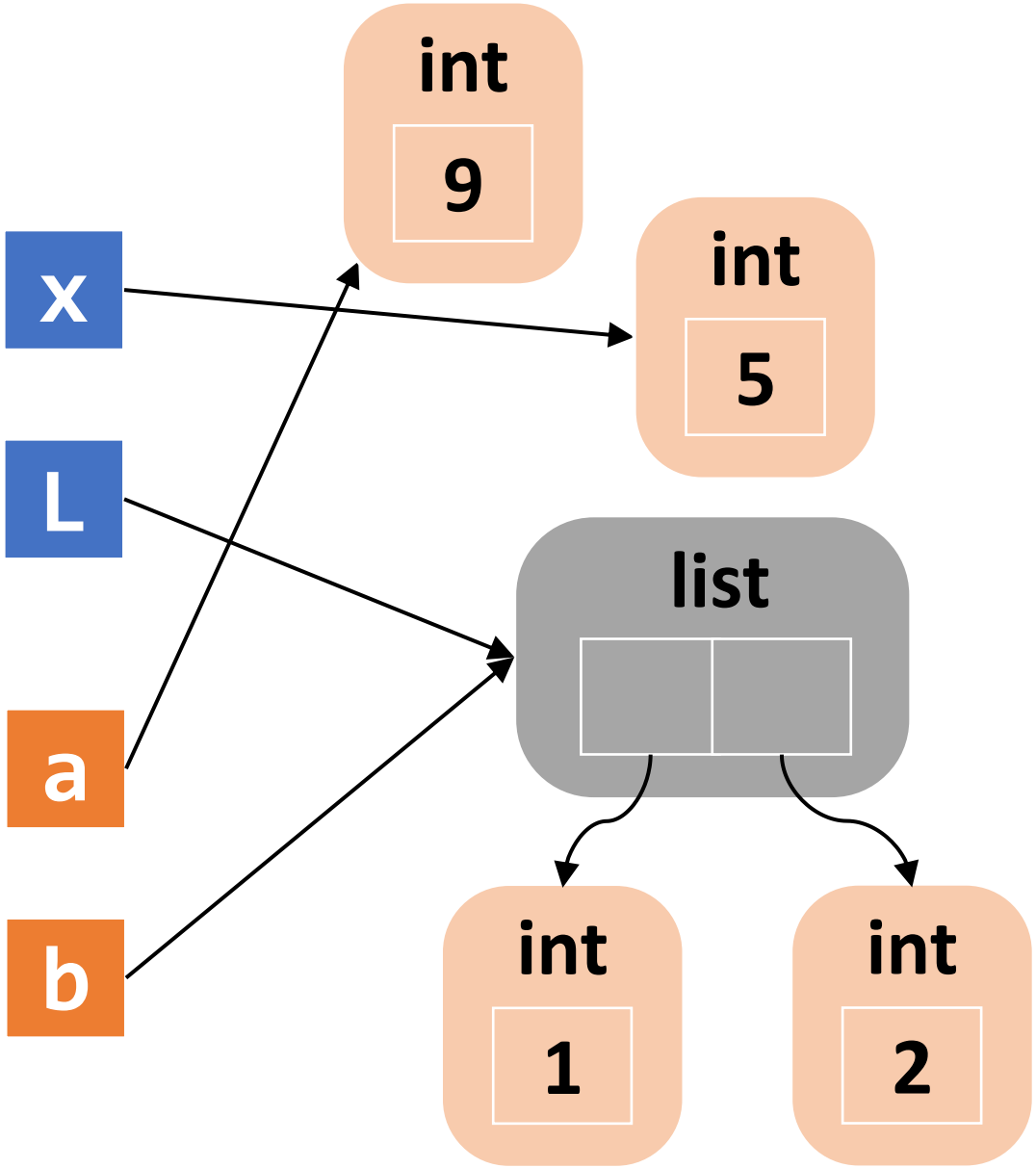
b = L

a = x



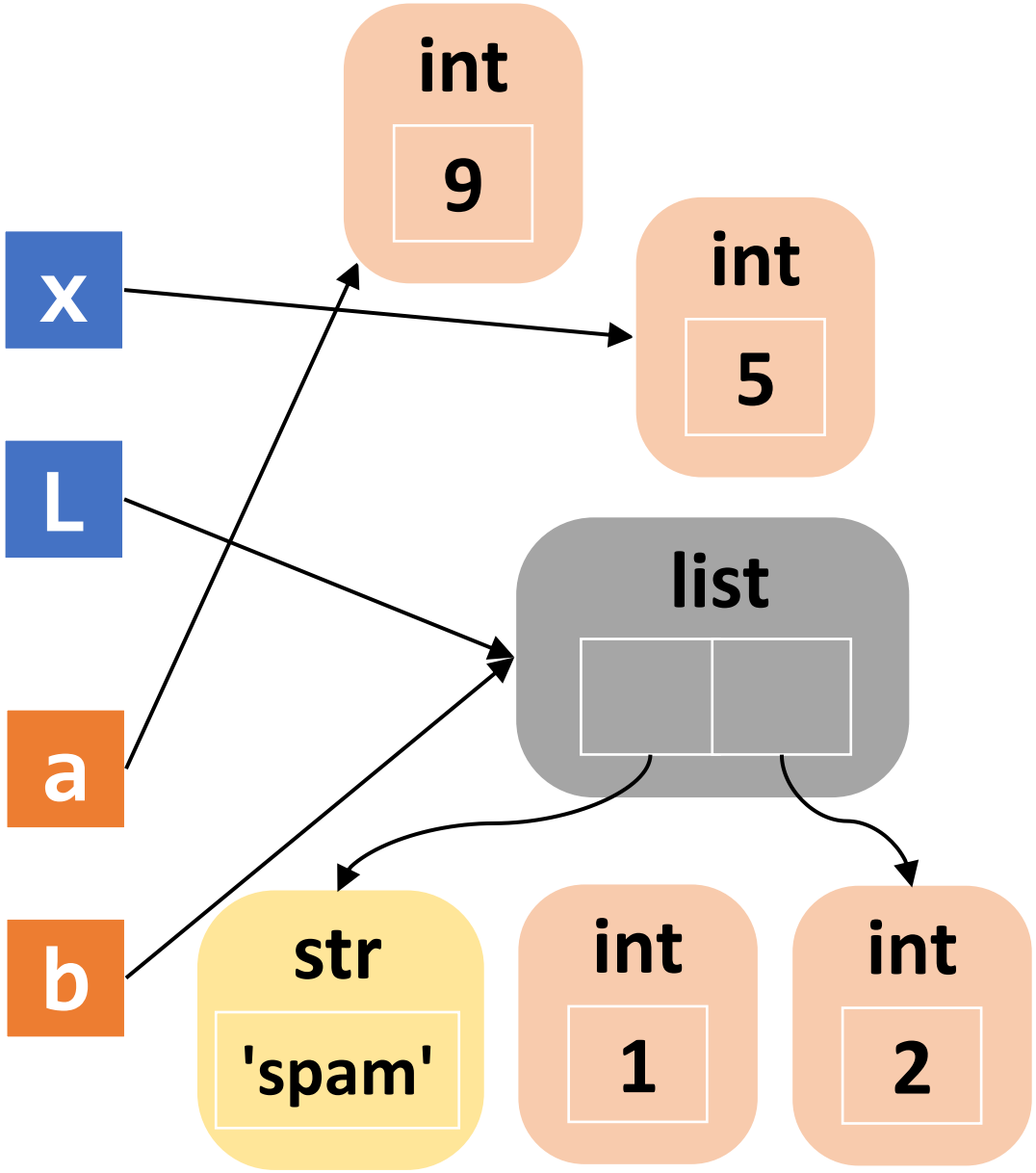
Passing Arguments

```
>>> def f(a, b):
...     a = 9
...     b[0] = 'spam'

>>> x = 5
>>> L = [1, 2]
>>> f(x, L)
```



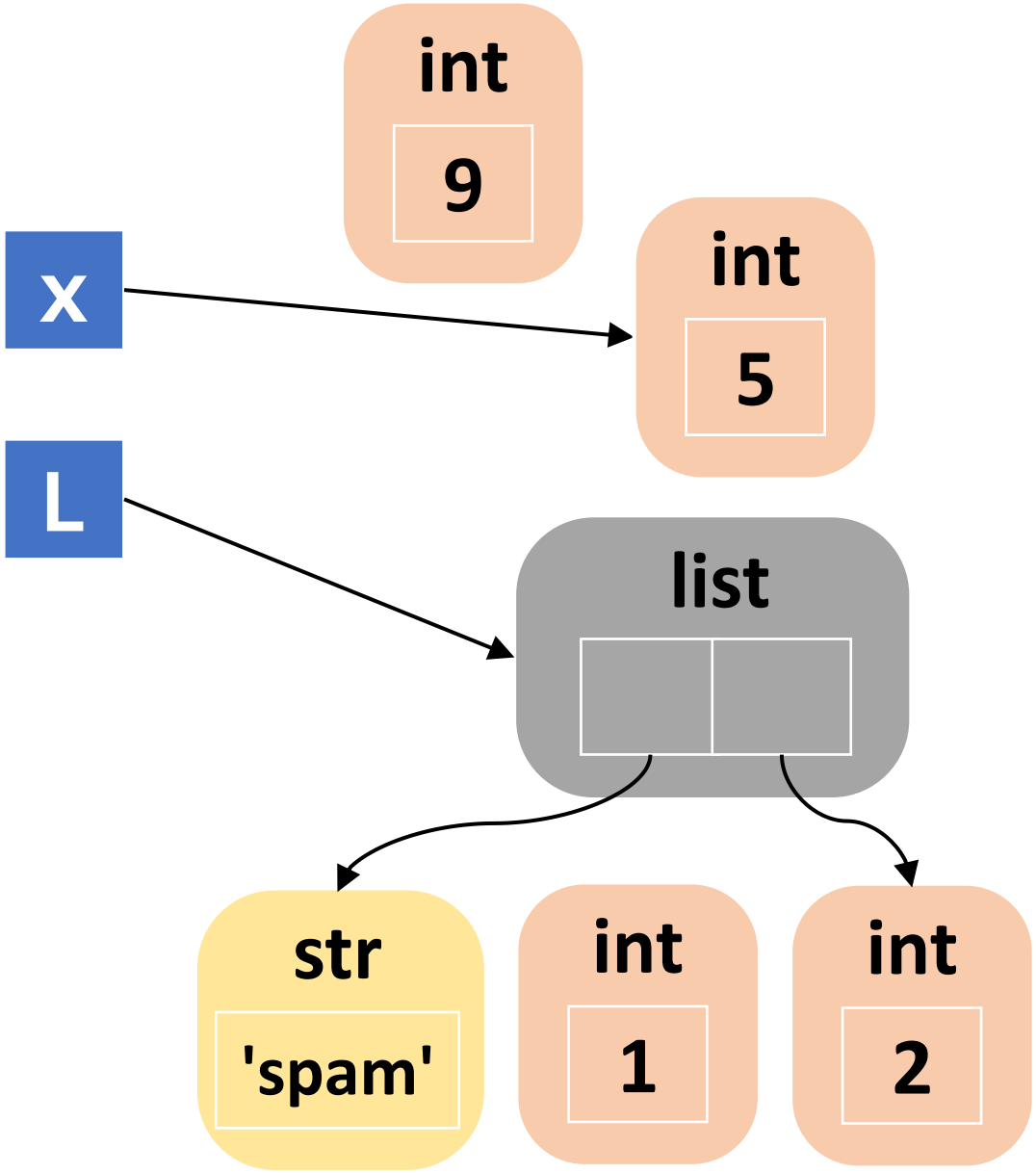
Passing Arguments

```
>>> def f(a, b):
...     a = 9
...     b[0] = 'spam'

>>> x = 5
>>> L = [1, 2]
>>> f(x, L)
```



Passing Arguments

```
>>> def f(a, b):
...     a = 9
...     b[0] = 'spam'

>>> x = 5
>>> L = [1, 2]
>>> f(x, L)
>>> print(x)
5
>>> print(L)
['spam', 2]
```



Recursive Function

- Functions that call themselves either directly or indirectly

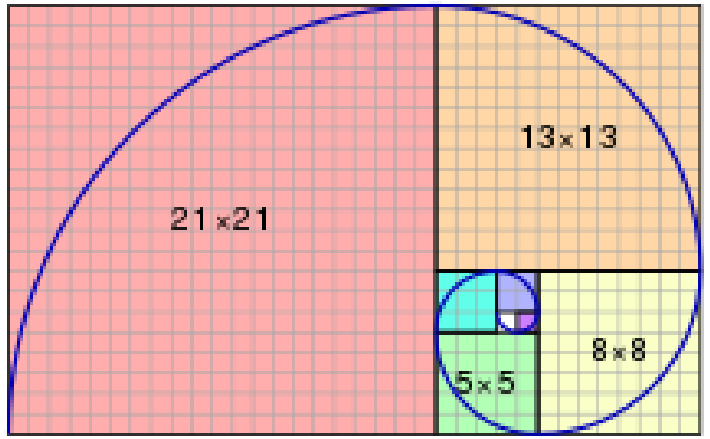
$$f(n) = \begin{cases} n \times f(n-1) & n > 1 \\ 1 & n \leq 1 \end{cases}$$

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

Fibonacci Numbers

$$f(n) = f(n-1) + f(n-2) \quad n \geq 2$$

$$f(0) = 0,\ f(1) = 1$$




```
def fib(n):
```

Lambda Expressions

- A lambda expression is an anonymous function
- Allow us to define a function much more easily

```
>>> f = lambda x: x * x
>>> print(f(4))

>>> L = ['hello', 'World', 'hi', 'Bye']
>>> sorted(L)
>>> sorted(L, key=str.lower)
>>> sorted(L, key=len)
>>> sorted(L, key=lambda x: x[-1])
```

Why Functions?

- Make the program modular and readable
- Can be reused later
- You can even package them as a library (or a module)