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

Systems Software &
Architecture Lab.

Seoul National University

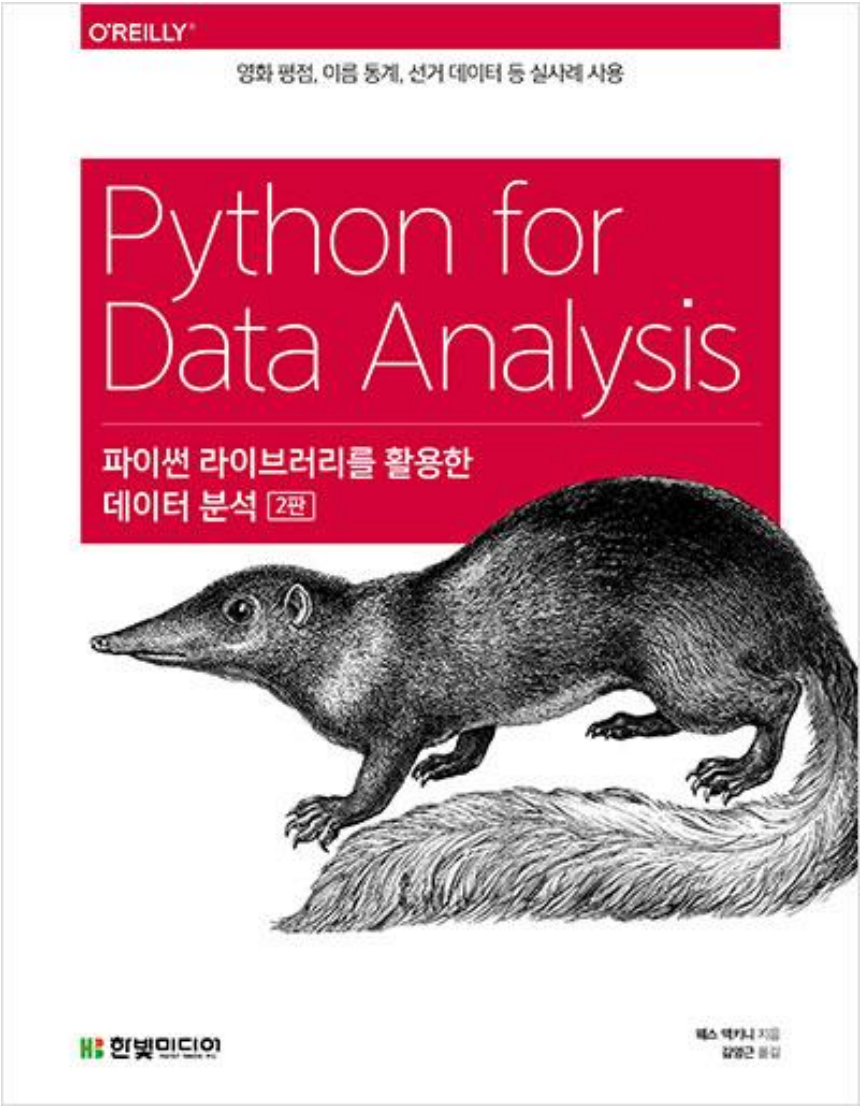
Jan. 3 – 14, 2022

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 (1주차)

		1/3 (Mon)	1/4 (Tue)	1/5 (Wed)	1/6 (Thu)	1/7 (Fri)
오전	8:30	확률통계	확률통계	확률통계	확률통계	확률통계
	9:30					
	10:30					
	11:30					
	12:30	점심 시간 (12:30 ~ 1:30)				
오후	1:30	Python Basics	Python Sequence and Collections	NumPy I	NumPy II	Pandas I
	2:30					
	3:30	[실습]	[실습]	[실습]	[실습]	[실습]
	4:30					

일정 (2주차)

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

About Us

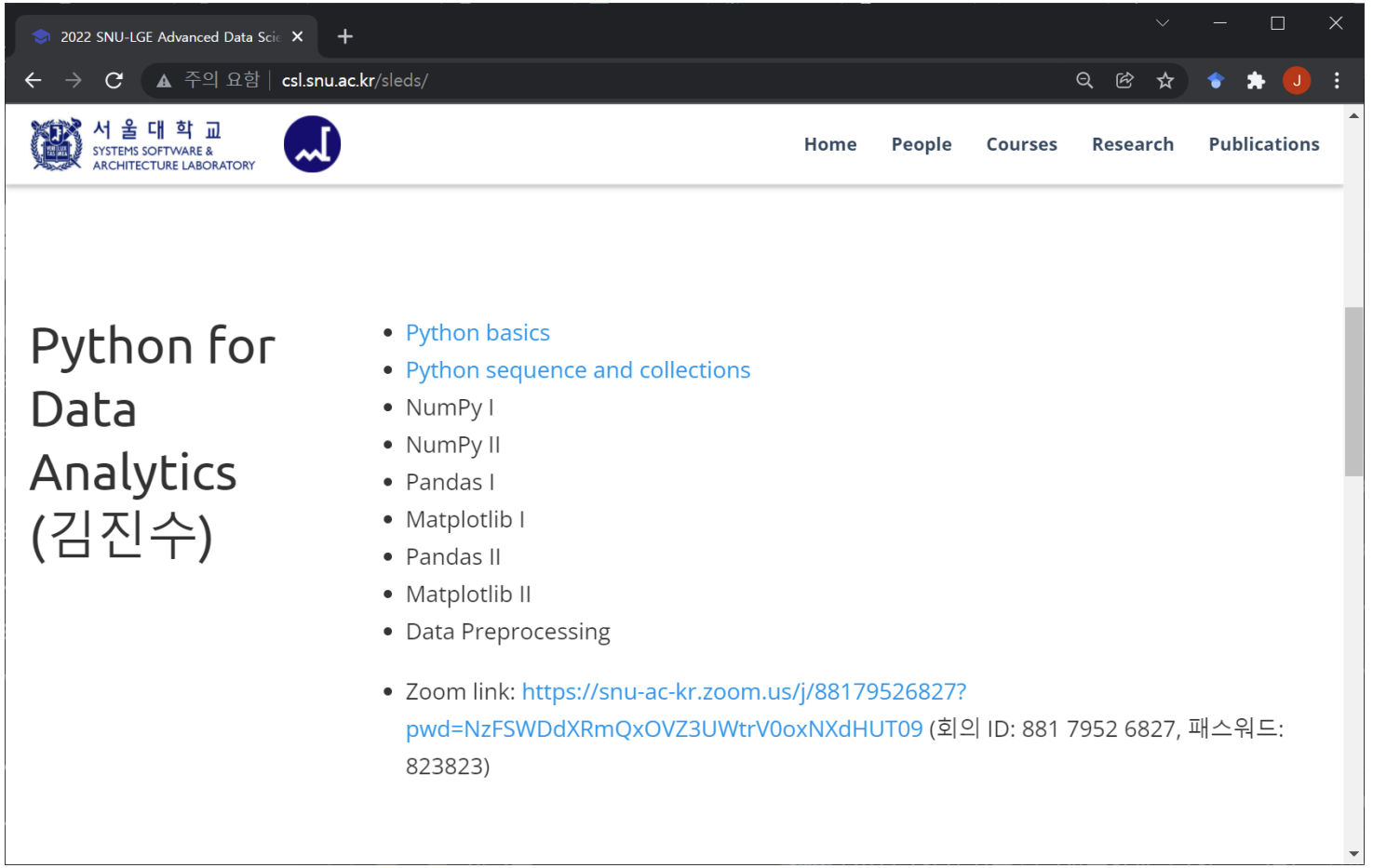
- 김진수 (Jin-Soo Kim)
  - Professor @ Dept. of Computer Science & Engineering, SNU
  - Systems Software & Architecture Laboratory
  - Operating systems, storage systems, parallel and distributed computing, embedded systems, …
- E-mail: jinsoo.kim@snu.ac.kr
- http://csl.snu.ac.kr
- Tel: 02-880-7302
- Office: SNU Engineering Building #301-504
- TAs: 정성엽(seongyeop.jeong@snu.ac.kr), 이준석(shawn159@snu.ac.kr)

Course Homepage

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

- Lecture slides
- Lab. materials

- ID: lge
- PW: 6060



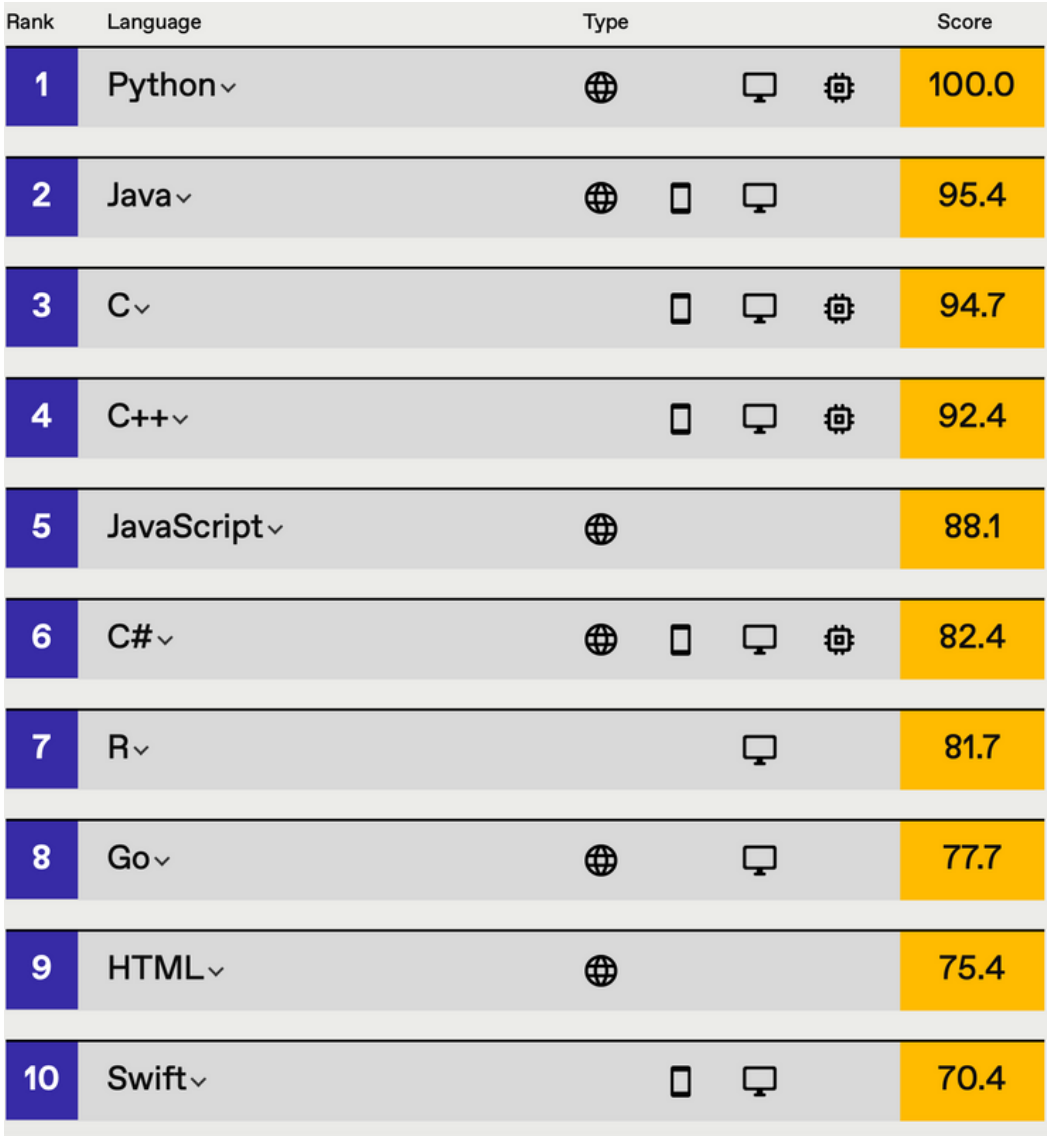
Outline

- Introduction to Python
- Basic data types
- Control Flow
- Functions

Introduction to Python

Why Python?

- 1st place among the "Top Programming Languages" (2017-2021, IEEE Spectrum)
- "Fastest growing major programming language" (stackoverflow.com, 2019)
- The 2nd most popular programming language in GitHub (Nov. 2021)
- "The language of AI"



Rank	Language	Type	Score
1	Python		100.0
2	Java		95.4
3	C		94.7
4	C++		92.4
5	JavaScript		88.1
6	C#		82.4
7	R		81.7
8	Go		77.7
9	HTML		75.4
10	Swift		70.4

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:

```
#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
```

```
    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%
```

```
    }
}

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/

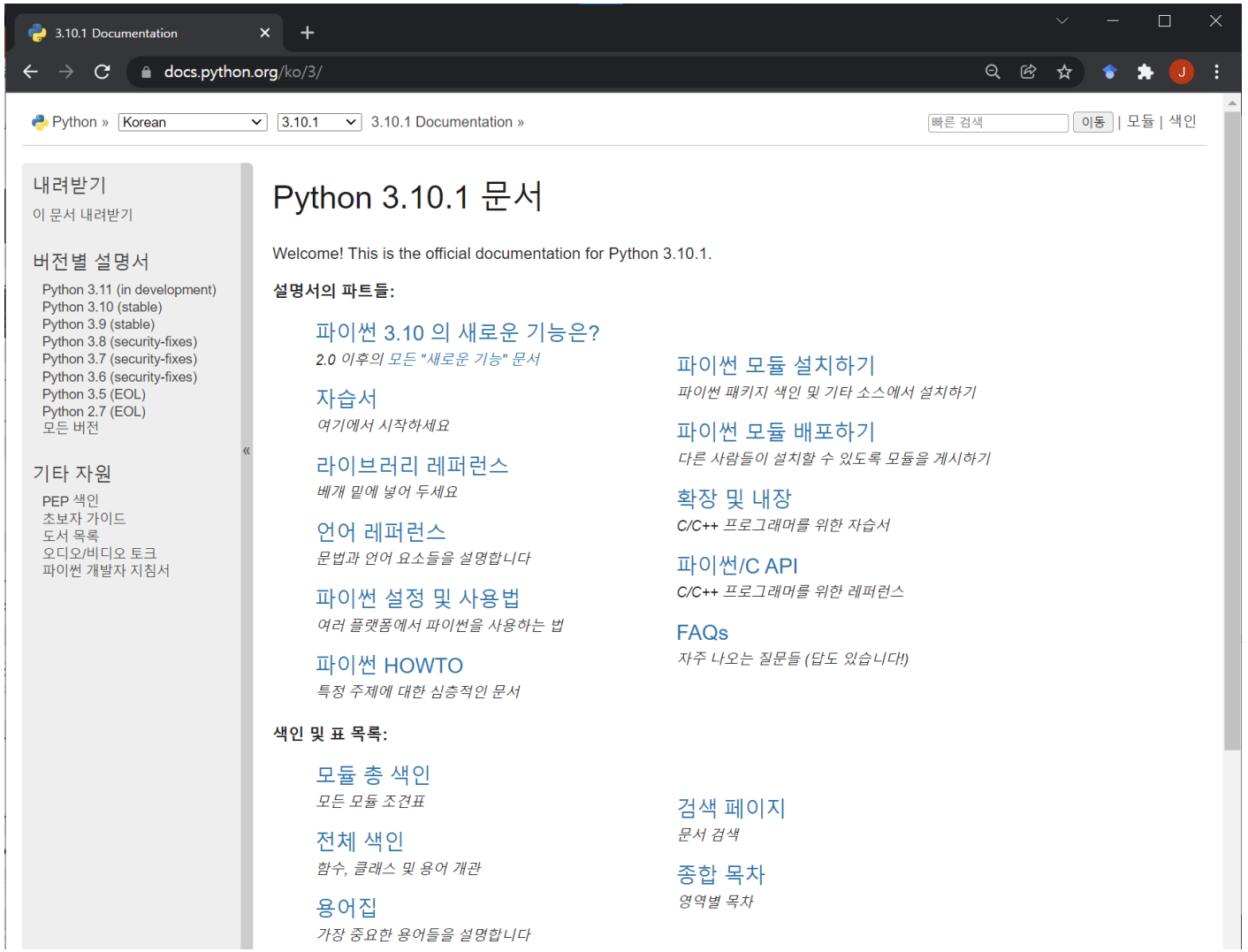
Python Versions

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

- The latest version: 3.10.1

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

https://docs.python.org



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 (audio, video, etc.)
- Scientific computing and many more …



Welcome to the World of Spam!

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

Basic Data Types

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"

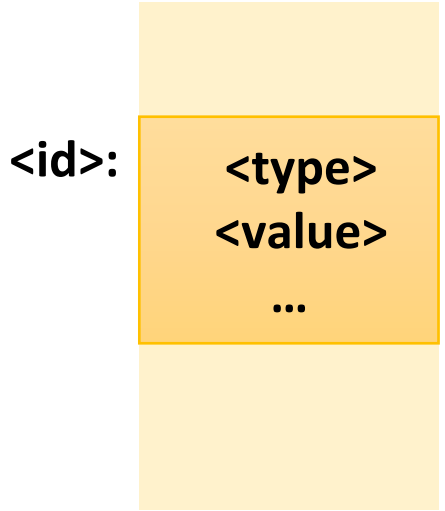
Data Types

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

Object-oriented Data Model

- Objects are Python's abstraction for data

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

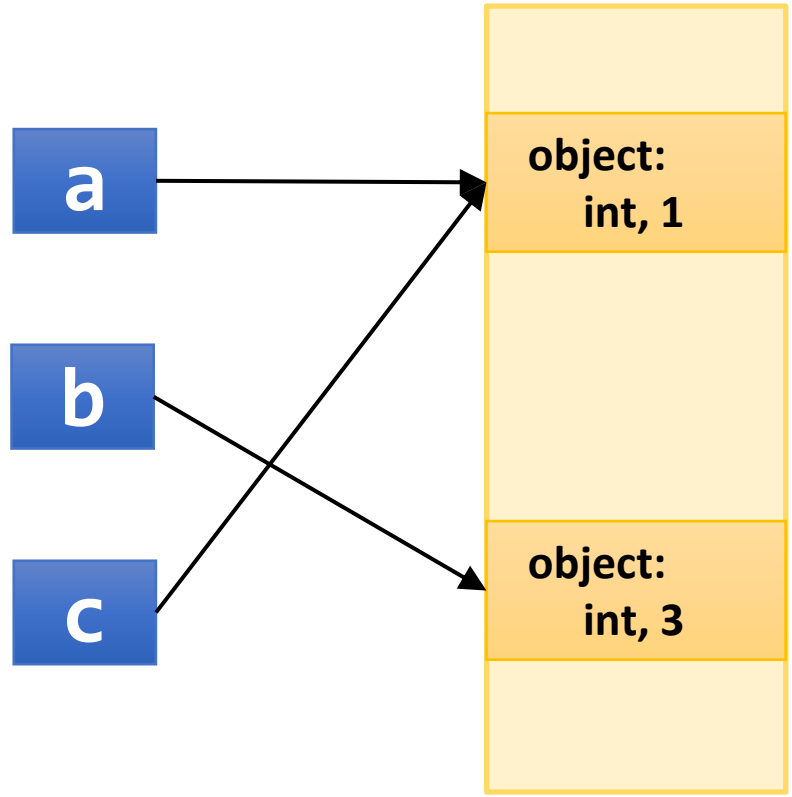


- The '`is`' operator compares the identity of two objects
- Objects can be immutable (e.g., numbers, strings, tuples, …)
- Different variables can refer to the same object

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))
```



Numbers

- Integers (정수): unlimited range
  - Decimal representation: `0` `12` `1_00` `0001` ⊗
  - Binary representation: `0b0` `0b1100` `0b110_0100`
  - Octal representation: `0o0` `0o14` `0o144`
  - Hexadecimal representation: `0x0` `0xc` `0x64`
- Floating-points (부동소수점수)
  - Double-precision only ($4.9 \times 10^{-324} \sim 1.8 \times 10^{308}$)
  - Always use a decimal representation (no binary/octal/hexadecimal)
  - `12.0`
  - `0.0000_0000_001`
  - `3.14e-5`

Arithmetic Operations

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

Bitwise Operations (Integers Only)

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

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
- …

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)     # Use math.isclose()

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

Boolean

- `bool`
  - `False` or `True`
  - A subtype of the integer type (Non-zero values are treated as `True`)
  - Boolean values behave like the integer 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)
```

String

- A sequence of characters (immutable)
  - 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('안녕\n-\t'+이름)
>>> print(r'C:\abc\name')
```

ord() and chr()

- ord(*c*)
  - Return the Unicode for a character *c*
- chr(*i*)
  - Return a Unicode string of one character with ordinal *i*

```
>>> ord('a')
97
>>> ord('b')
98
>>> ord('A')
65
>>> ord('*')
42
>>> ord('\n')
10
>>> ord('가')
44032
>>> ord('각')
44033
```

```
>>> chr(97)
'a'
>>> chr(97+1)
'b'
>>> c = 'x'
>>> chr(ord(c)+1)
'y'
>>> chr(ord('A')+\
... ord(c)-ord('a'))
'X'
>>> chr(44032)
'가'
```

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

String Indexing and Slicing

- *str*[*start*:*stop*:*step*]
- Same as list slicing

M	o	n	t	y		P	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> s = 'Monty Python'
>>> print(s[1:2])
o
>>> print(s[8:])
thon
>>> print(s[:])
Month Python
>>> print(s[::2])
MnyPto
```

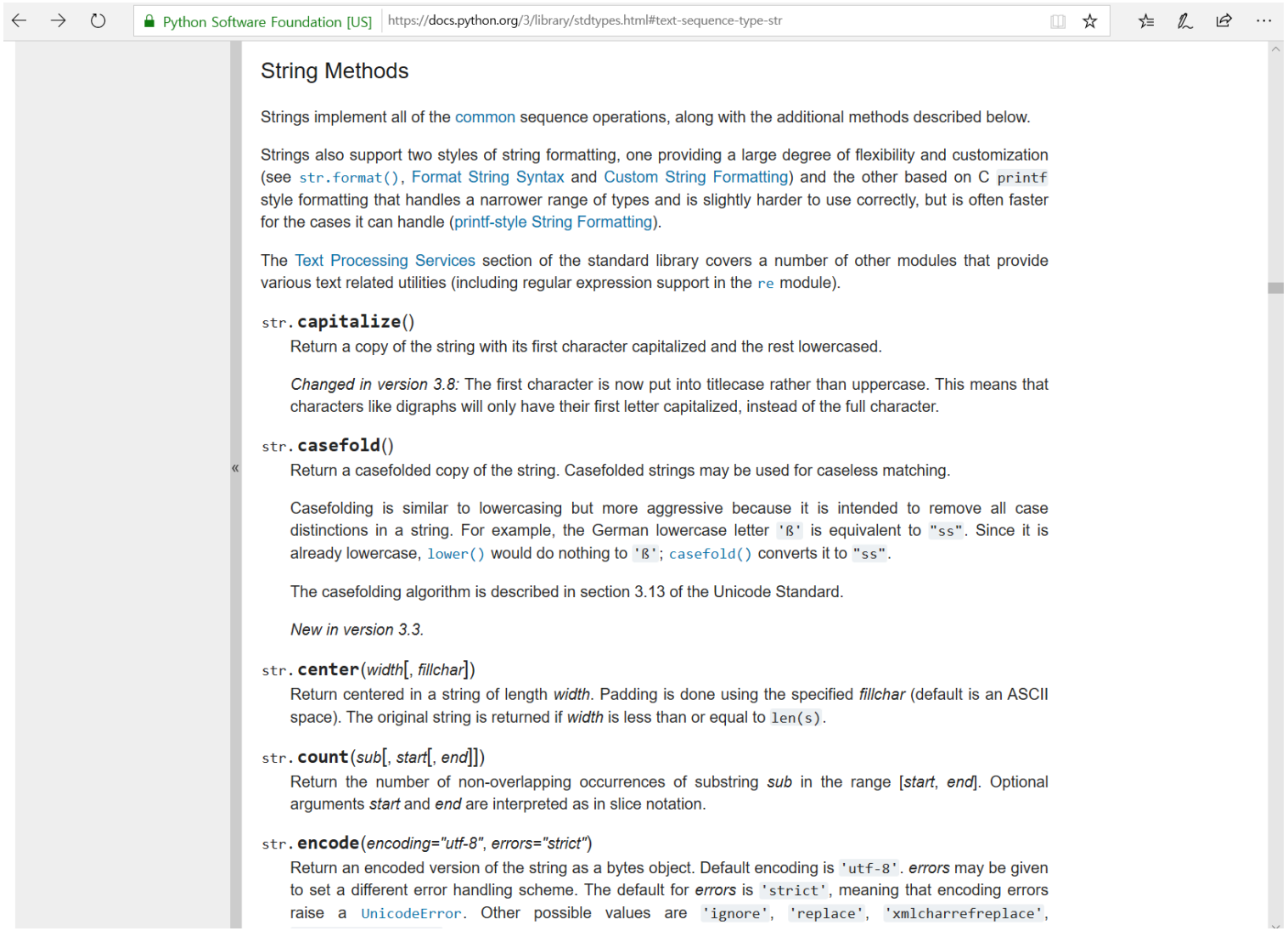
```
>>> print(s[-4:])
thon
>>> print(s[:-5])
Monty P
>>> print(s[:-6:-1])
nohty
>>> print(s[::-1])
nohtyP ytnoM
```

The `in` Operator

- Check to see if one string is in another string
- Return `True` or `False`

```
>>> fruit = 'banana'
>>> 'n' in fruit
True
>>> 'm' in fruit
False
>>> 'nan' in fruit
True
>>> if 'a' in fruit:
...     print('Found it!')
Found it!
```

String Methods



Test

- *str*.`startswith(`*prefix*[,*start*[,*end*]]`)`
  - `True` if string starts with the *prefix*
- *str*.`endswith(`*suffix* [,*start*[,*end*]]`)`
  - `True` if string ends with the suffix
- *str*.`isalpha()`
  - `True` if all characters are alphabetic
- *str*.`isdigit()`
  - `True` if all characters are digits
- *str*.`isprintable()`
  - `True` if all characters are printable
- *str*.`islower()`
  - `True` if all characters are lower case
- *str*.`isupper()`
  - `True` if all characters are uppercase
- *str*.`isspace()`
  - `True` if there are only whitespace characters

Find / Replace

- *str*.`count(`*sub*[,*start*[,*end*]]`)`
- *str*.`find(`*sub*[,*start*[,*end*]]`))`
  - Return the lowest index where substring *sub* is found (-1 if *sub* is not found)
- *str*.`index(`*sub*[,*start*[,*end*]]`))`
  - Like `find()`, but raise `ValueError` if *sub* is not found
- *str*.`replace(`*old*, *new*[,*count*]`))`
  - Return a copy of the string with all occurrences of substring *old* replaced by *new*

```
>>> b = 'banana'
>>> print(b.count('a'))
3

>>> print(b.find('x'))
-1

>>> print(b.index('na'))
2

>>> print(b.replace('a','x'))
bxnxnx
```

Reformat (1)

- *str*.`lower()`
  - Return a copy of the string with all the characters converted to lowercase
- *str*.`upper()`
  - Return a copy of the string with all the characters converted to uppercase
- *str*.`capitalize()`
  - Return a copy of the string with its first character capitalized and the rest lowercased

```
>>> s = 'MoNtY PyThOn'

>>> print(s.lower())
monty python

>>> print(s.upper())
MONTY PYTHON

>>> print(s.capitalize())
Monty python
```

Reformat (2)

- *str*.`lstrip`([*chars*])
  - Return a copy of the string with leading characters removed.
  - If omitted, the *chars* argument defaults to whitespace characters
- *str*.`rstrip`([*chars*])
  - Like `lstrip()`, but trailing characters are removed
- *str*.`strip`([*chars*])
  - *str*.`lstrip`([*chars*]) + *str*.`rstrip`([*chars*])

```
>>> s = '-- monty python --'

>>> print(s.lstrip(' -'))
monty python --

>>> print(s.rstrip('- '))
--- monty python

>>> print(s.strip(' -mno'))
ty pyth
```

Split

- *str*.`split`(*sep*, *maxsplit*)
  - Return a list of the words in the string, using *sep* as the delimiter string
  - If *maxsplit* is given, at most *maxsplit* splits are done. Otherwise all possible splits are made
  - The *sep* argument may consist of multiple characters (None = whitespaces)
  - If *sep* is given, consecutive delimiters are NOT grouped together

```
>>> s = 'hi hello world'
>>> s.split()
['hi', 'hello', 'world']
>>> t = '1:2:3'
>>> t.split(':')
['1', '2', '3']
>>> t.split(':', 1)
['1', '2:3']
>>> t = '1:2::3'
>>> t.split(':')
['1', '2', '', '3']
>>> t.split('::')
['1:2', '3']
```

Join

- *str*.`join(`*iterable*`)`
  - Return a string which is the concatenation of the strings in *iterable*
  - *iterable*: List, Tuple, String, Dictionary, Set
  - The separator between elements is the string (*str*) providing this method

```
>>> menu = ['spam', 'ham', 'egg']
>>> ','.join(menu)
'spam,ham,egg'
>>> ' '.join(menu)
'spam ham egg'
>>> ' * '.join(menu)
'spam * ham * egg'
>>> '#'.join('spam')
's#p#a#m'
```

input(): Getting User Input

- Read a line and convert it to a string with 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!
```

Formatting Strings

```
>>> a = 2
>>> b = 'spam'
>>> c = 4.99
```

- Old string formatting with % operator

```
>>> print('%d %ss cost $%f' % (a, b, c))
2 spams cost $4.990000
```

- The string `format()` method

```
>>> print('{} {}s cost ${}'.format(a, b, c))
2 spams cost $4.99
```

- f-strings (formatted string literals, since Python 3.6)

```
>>> print(f'{a} {b}s cost ${c}')
2 spams cost $4.99
```

Formatting with % Operator

- String formatting expression: `'... %d ...' % (values)`

```
>>> a = 2
>>> fruit = 'apples'

>>> print('I have %s.' % fruit)
I have apples.

>>> print('I have %d %s.' % (a, fruit))
I have 2 apples.

>>> print('sqrt(%d) is %f' % (a, a**0.5))
sqrt(2) is 1.414214
```

Formatting with % Operator: Type Code

- %[*flags*][*width*][.*precision*]*typecode*
  - *flags*
    - Left justification (–)
    - Numeric sign (+)
    - Zero fills (0)
    - …
  - *width*: a total minimum field width for the substitute text
  - *precision*: the number of digits to display after a decimal point for floating-point numbers

Code	Meaning
%s	String (or most objects with str())
%c	Character
%d	Integer
%o	Octal integer
%x	Hexadecimal integer
%X	Same as %x, but with uppercase letters
%f	Floating-point decimal
%e	Floating point with exponent, lowercase
%E	Floating point with exponent, uppercase
%%	Literal %

Formatting with % Operator: Examples

```
>>> x = 1234
>>> s = 'integers: ...%d...%-6d...%06d' % (x, x, x)
>>> print(s)
integers: ...1234...1234  ...001234

>>> f = 3.14159265
>>> print('%e | %E | %f' % (f, f, f))
3.141593e+00 | 3.141593E+00 | 3.141593
>>> print('%-6.2f | %05.2f | %+06.1f' % (f, f, f))
3.14   | 03.14 | +003.1

>>> h = '0x%08x' % x
>>> print(hex(x), h)
0x4d2 0x000004d2
```

Formatting with format()

- The string `format()` method: `'... {} ...'.format(values)`

```
>>> a = 2
>>> fruit = 'apples'

>>> print('I have {}.'.format(fruit))
I have apples.

>>> print('I have {} {}.'.format(a, fruit))
I have 2 apples.

>>> print('sqrt({}) is {}'.format(a, a**0.5))
sqrt(2) is 1.414214
```

Formatting with format(): Examples

```
>>> print('{}, {} and {}'.format('spam', 'ham', 'eggs'))
spam, ham and eggs
>>> print('{0}, {2}, and {1}'.format('spam', 'ham', 'eggs'))
spam, eggs, and ham
>>> print('{x}, {y}, and {z}'.format(z='spam', x='ham', y='eggs'))
ham, eggs, and spam
>>> print('{x}, {0}, and {y}'.format('spam', x='ham', y='eggs'))
ham, spam, and eggs

>>> print('{}, {} and {}'.format(-1, 3.14159265, [1, 2, 3, 4]))
-1, 3.14159265 and [1, 2, 3, 4]
>>> x = 3.14156295
>>> print('{0:<10.2f}, {1:>10.5f}, and {val:+10.2f}'.format(x, x, val=x))
3.14      ,    3.14156, and      +3.14
```

Formatting with f-strings

- Formatted string literals: `f'... {value} ...'`

```
>>> a = 2
>>> fruit = 'apples'

>>> print(f'I have {fruit}.')
I have apples.

>>> print(f'I have {a} {fruit}.')
I have 2 apples.

>>> print(f'sqrt({a}) is {a**0.5}')
sqrt(2) is 1.4142135623730951
```

Formatting with f-strings: Examples

```
>>> val = 1234
>>> print(f'...{val}...{val:6}...{val:+6}...{val:06}')
...1234...  1234... +1234...001234
>>> print(f'...{val:x}...{val:o}...{val:b}...{val:e}')
...4d2...2322...10011010010...1.234000e+03
>>> st = 'hello'
>>> print(f'|{val:<10}|{val:>10}|{st:<10}|{st:>10}|')
|1234      |      1234|hello     |     hello|
>>> sq2 = 2**0.5
>>> print(f'{sq2:f}...{sq2:e}...{sq2:E}')
1.414214...1.414214e+00...1.414214E+00
>>> print(f'{sq2:<10.4f}...{sq2:010.4f}...{sq2:+6.1f}')
1.4142    ...00001.4142...  +1.4
```

Control Flow

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**

Branching

- <condition> has a value True or False
- Evaluate expressions in that block if <condition> is True

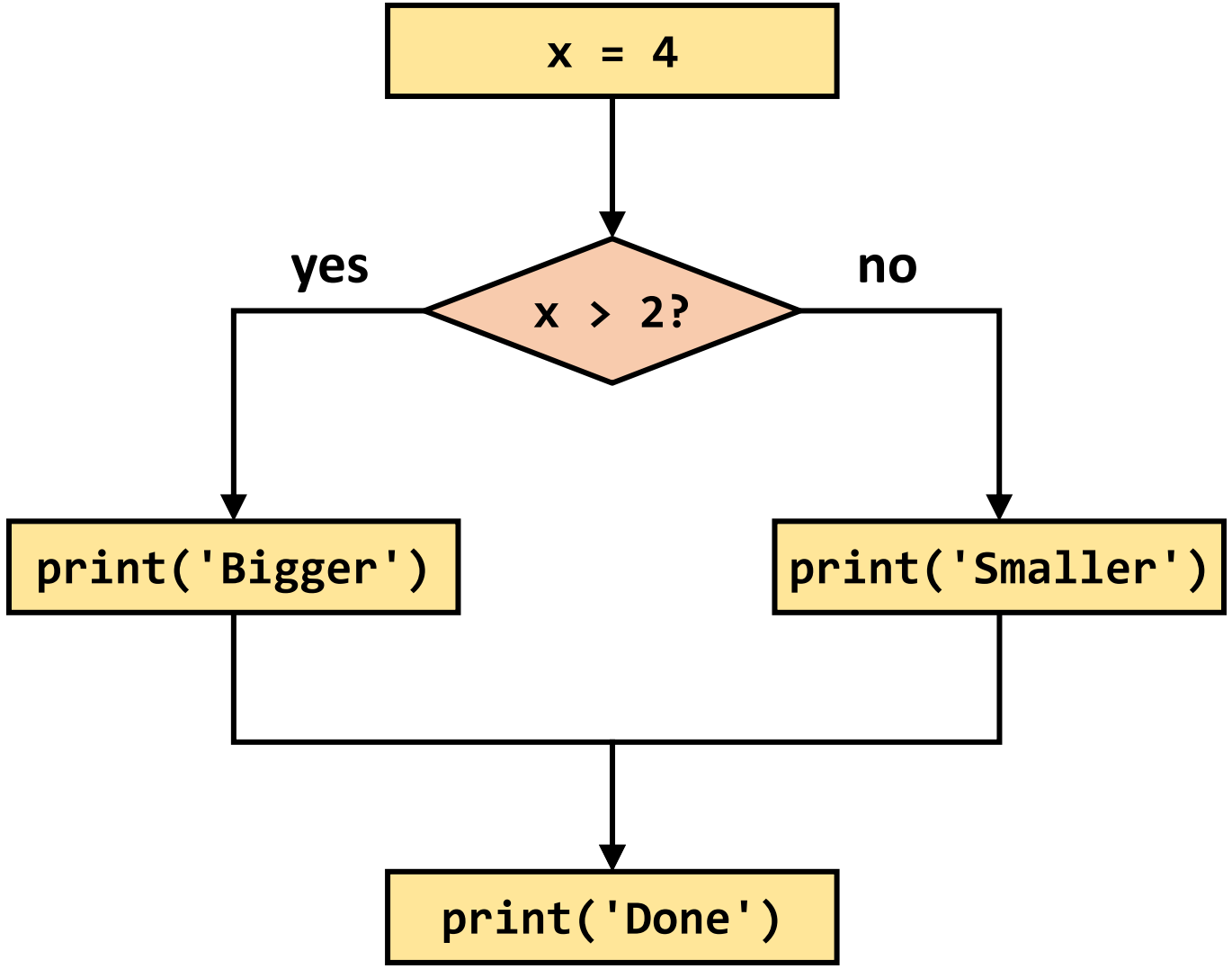
```
if <condition>:
    <expression>
    <expression>
    ...
```

```
if <condition>:
    <expression>
    ...
else:
    <expression>
    ...
```

```
if <condition>:
    <expression>
    ...
elif <condition>:
    <expression>
    ...
elif <condition>:
    <expression>
    ...
else:
    <expression>
    ...
```

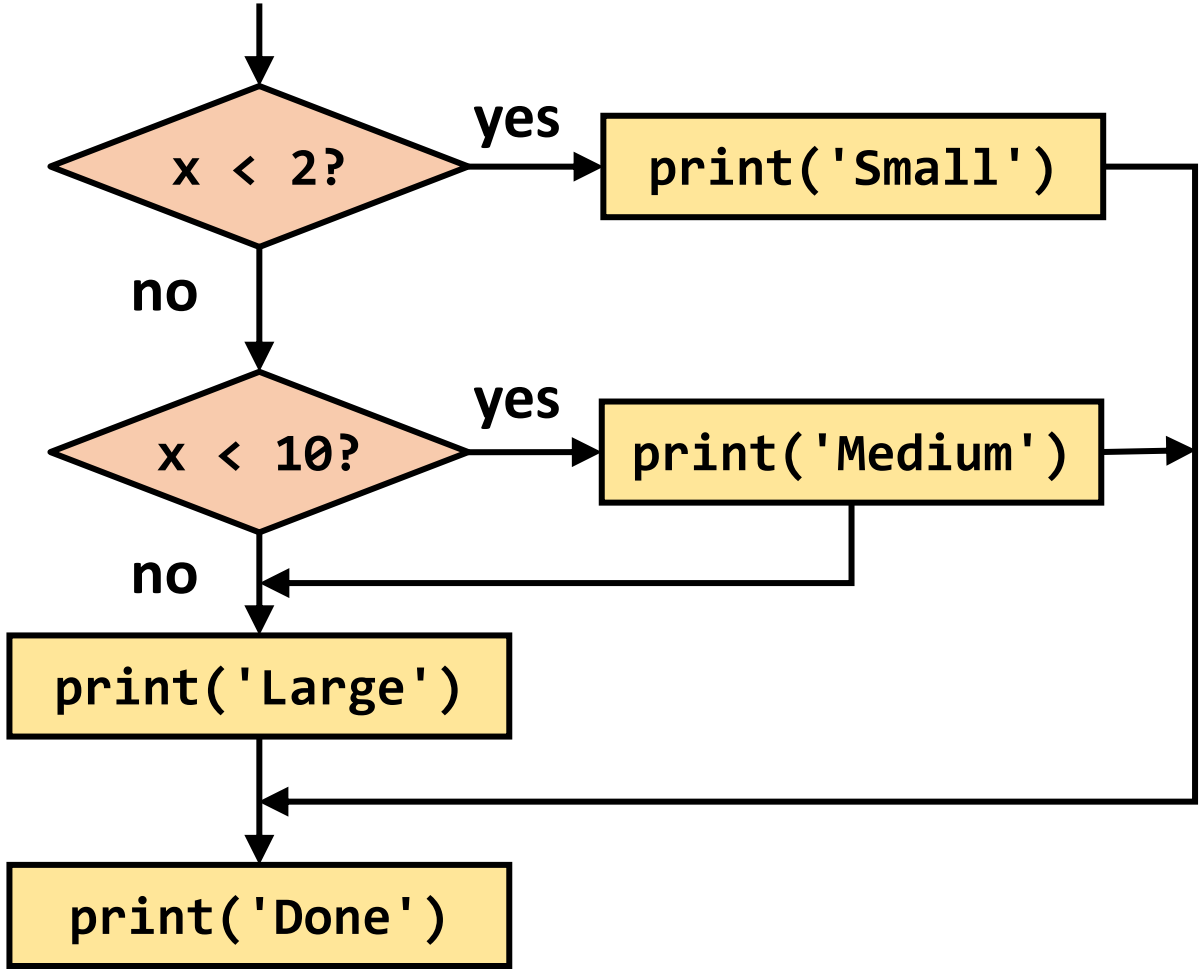
Two-way Decisions

```
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'
```

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, tuples, etc. (anything of determinate size)
  - Run the loop once for each of the items

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

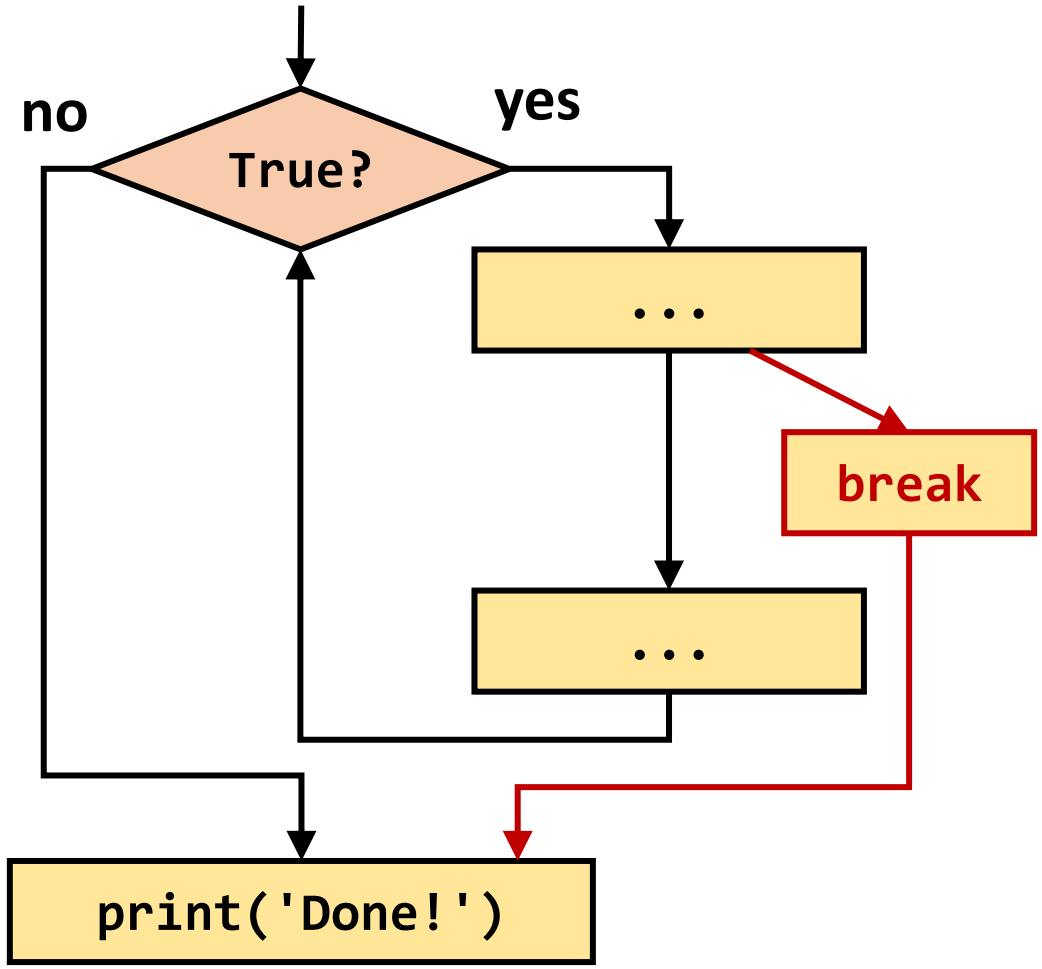
```
range(5)              # 0, 1, 2, 3, 4
range(-1, 4)          # -1, 0, 1, 2, 3
range(0,10,2)         # 0, 2, 4, 6, 8
range(5,0,-1)         # 5, 4, 3, 2, 1
range(10,2)           # ???
```

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

break: Breaking Out of a Loop

- Immediately exists whatever loop it is in
  - Skips remaining expressions in code block
  - Exits only innermost loop!

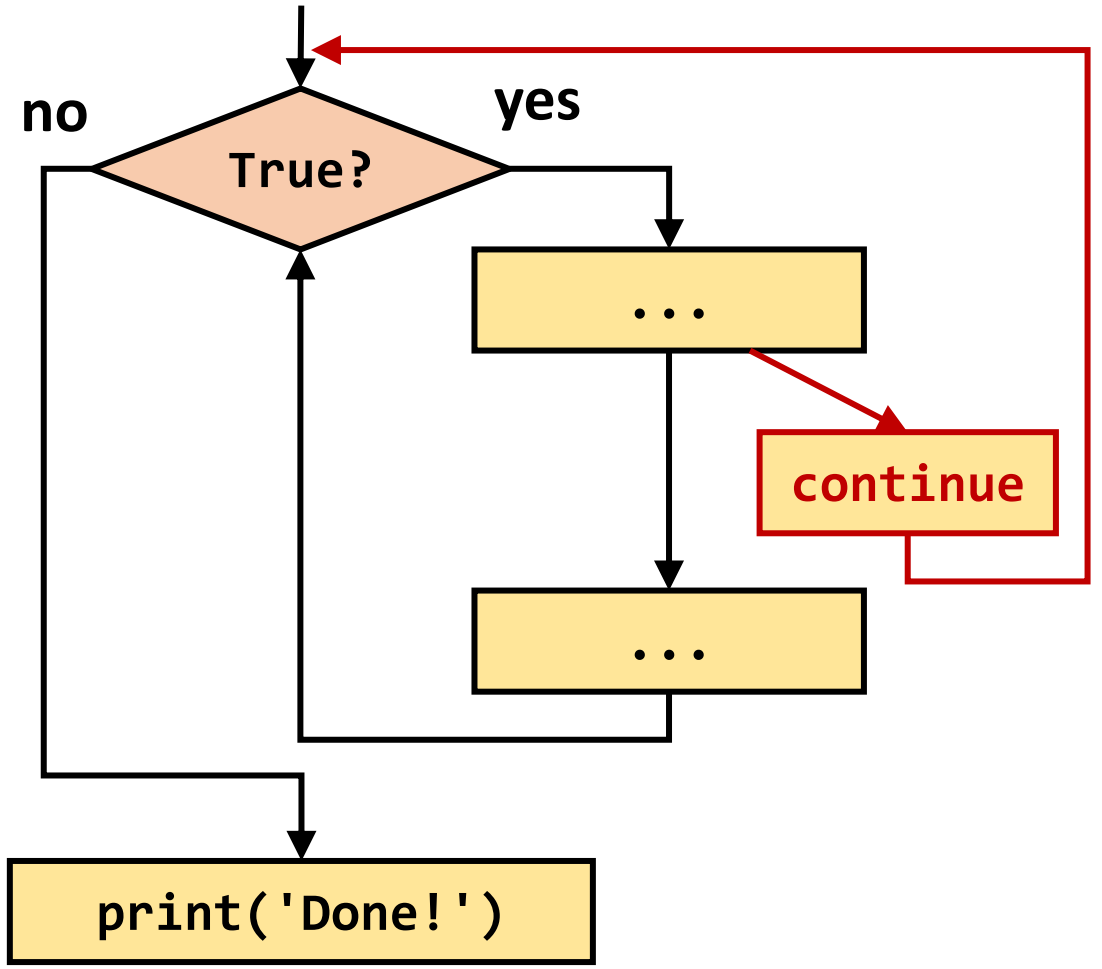
```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)
print('Done!')
```



continue: Finishing an Iteration

- 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!')
```



Looping Through a List

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

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

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

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

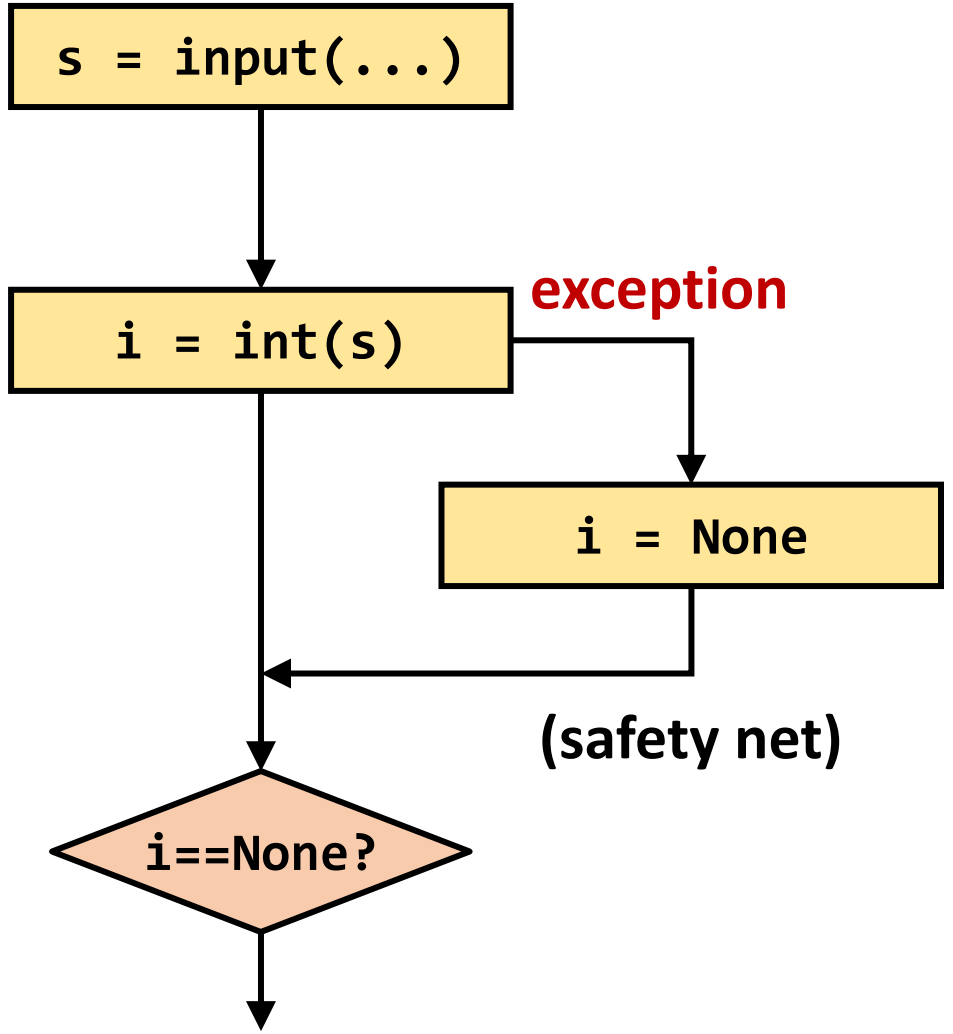
- Surround a dangerous section of code with `try` and `except`
- If the code in the `try` works – the `except` is skipped
- If the code in the `try` fails – it jumps to the `except` code block

```
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')
```



Functions

Python Functions

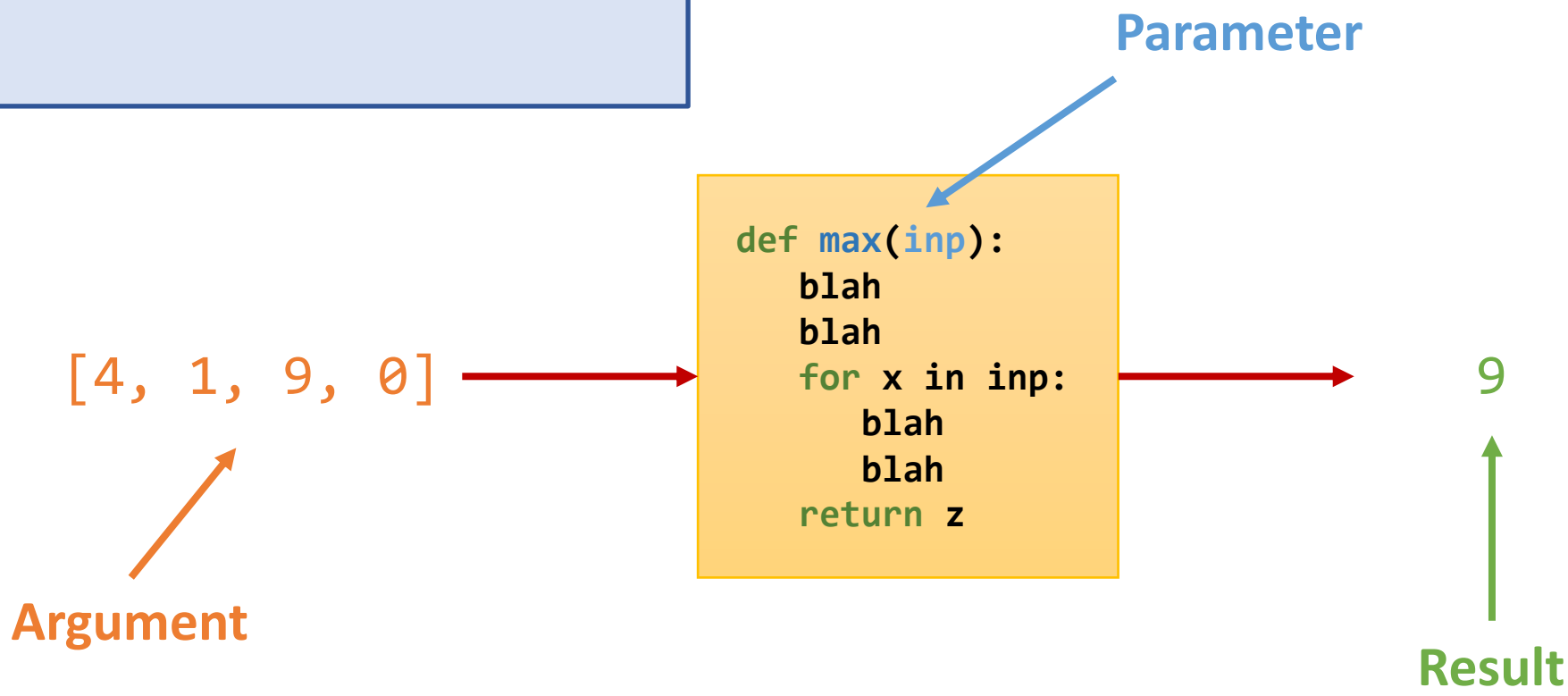
- A function is some reusable code that takes argument(s) as input, does some computation, and then returns a result or results.

- 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

Arguments, Parameters, and Results

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



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

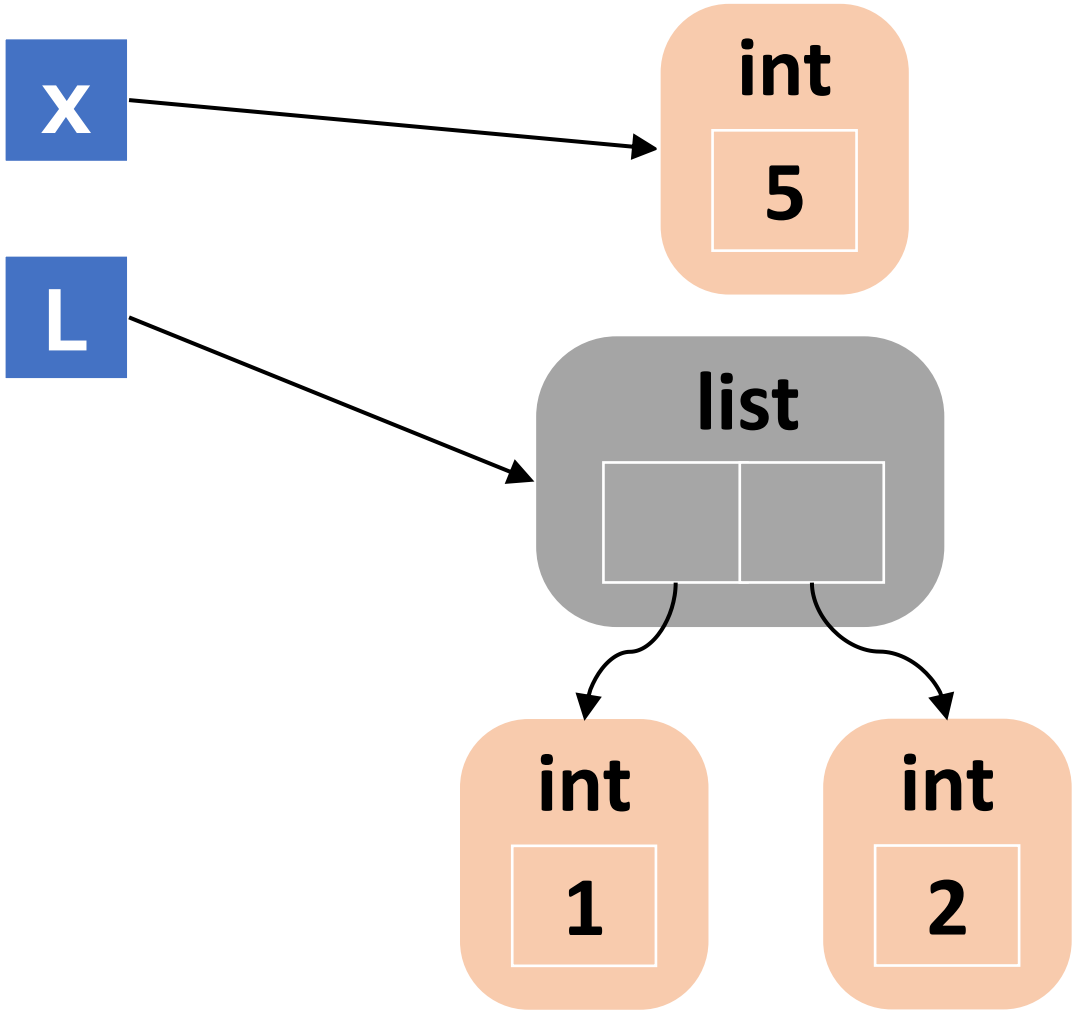
```
def food(name, *likes):
    print(name, 'likes ', end='')
    for d in likes:
        print(d, end=' ')
    print()

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]
```

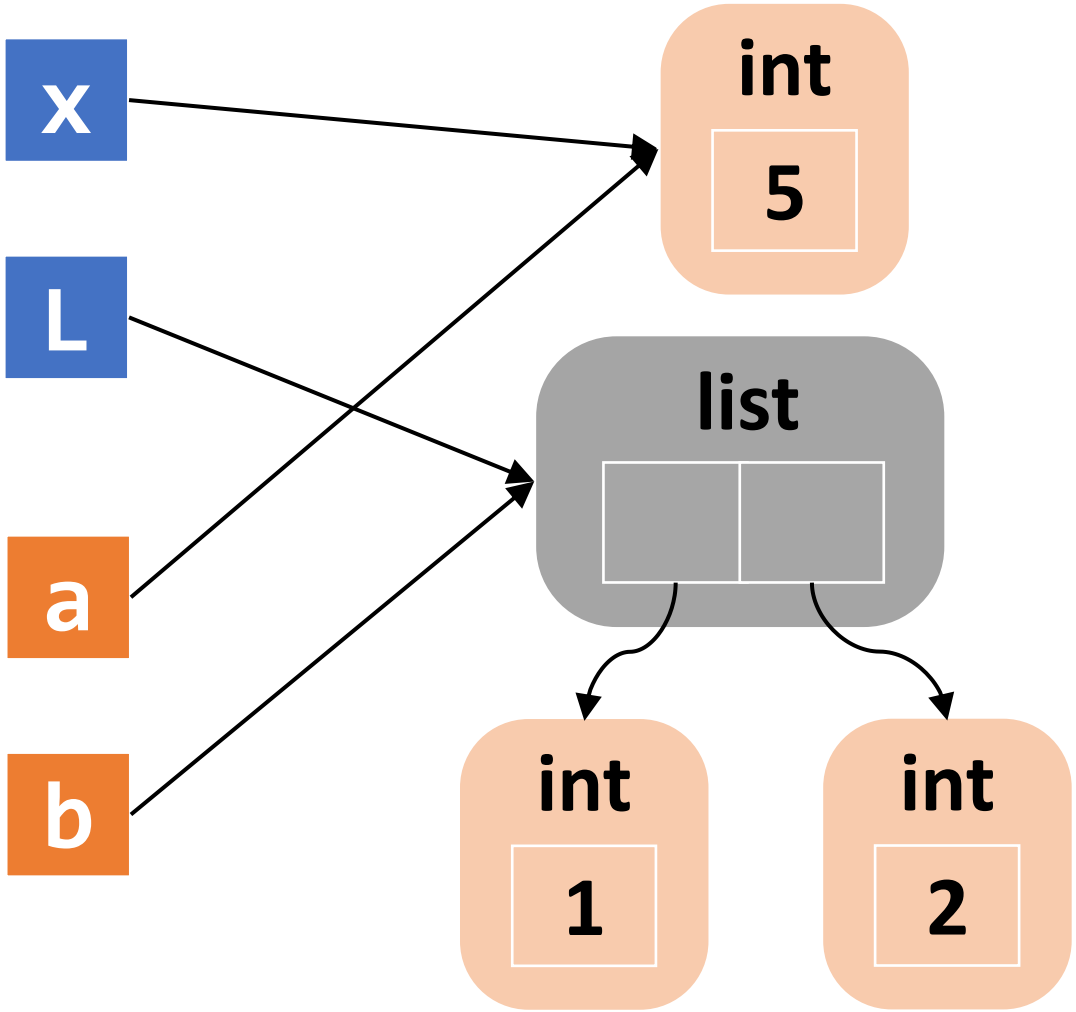


Passing Arguments

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

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

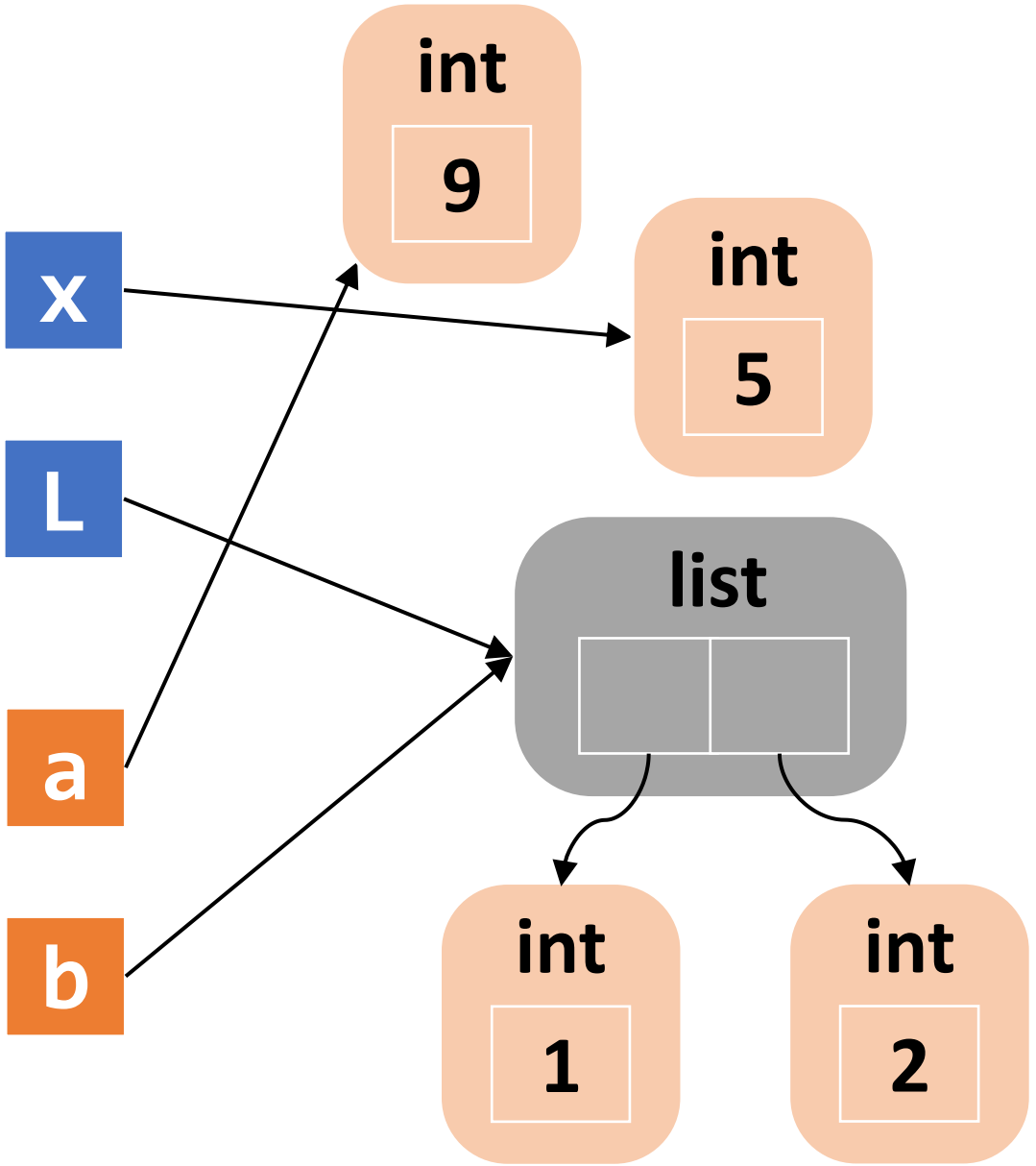
b = L

a = x



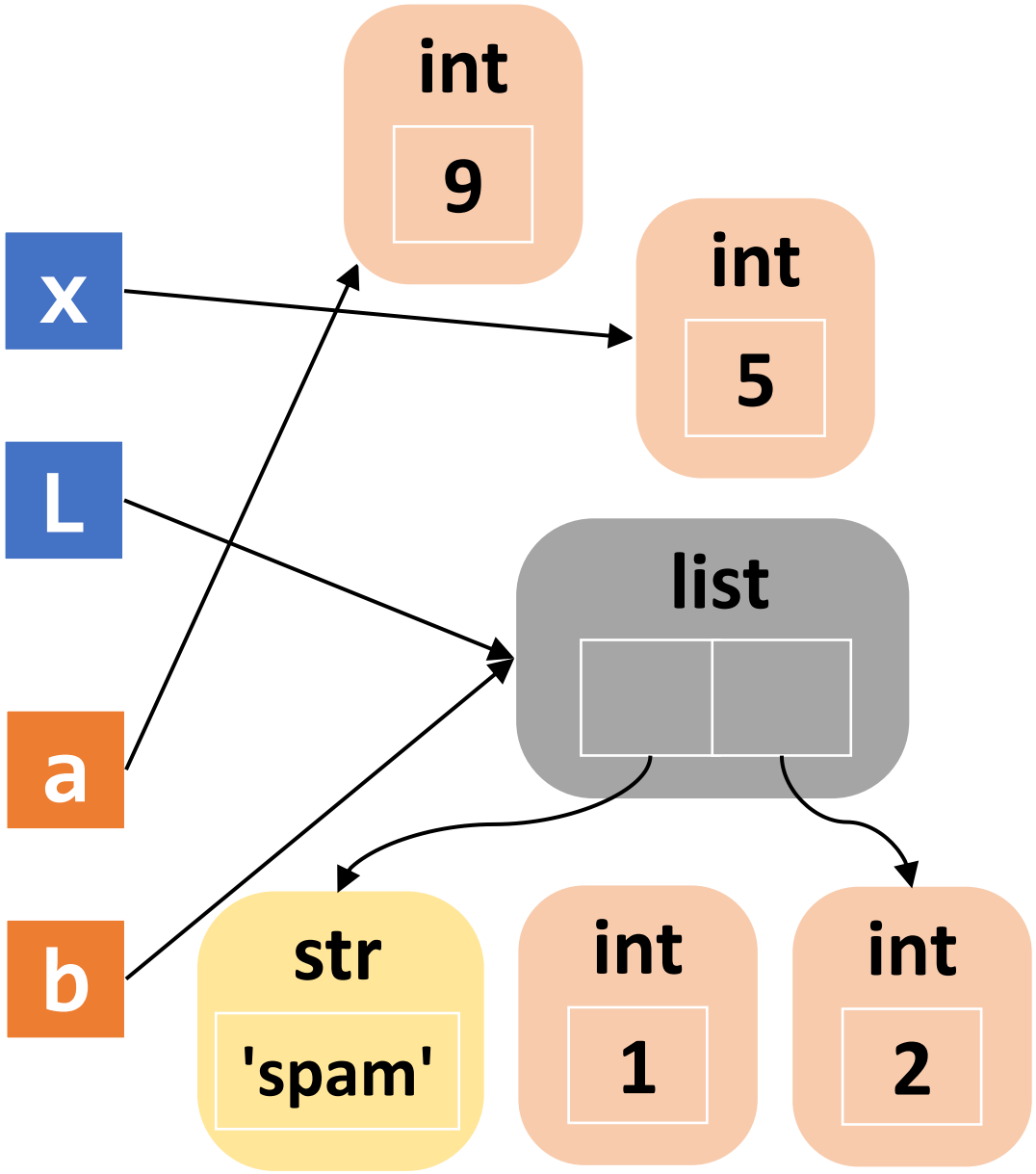
Passing Arguments

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

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



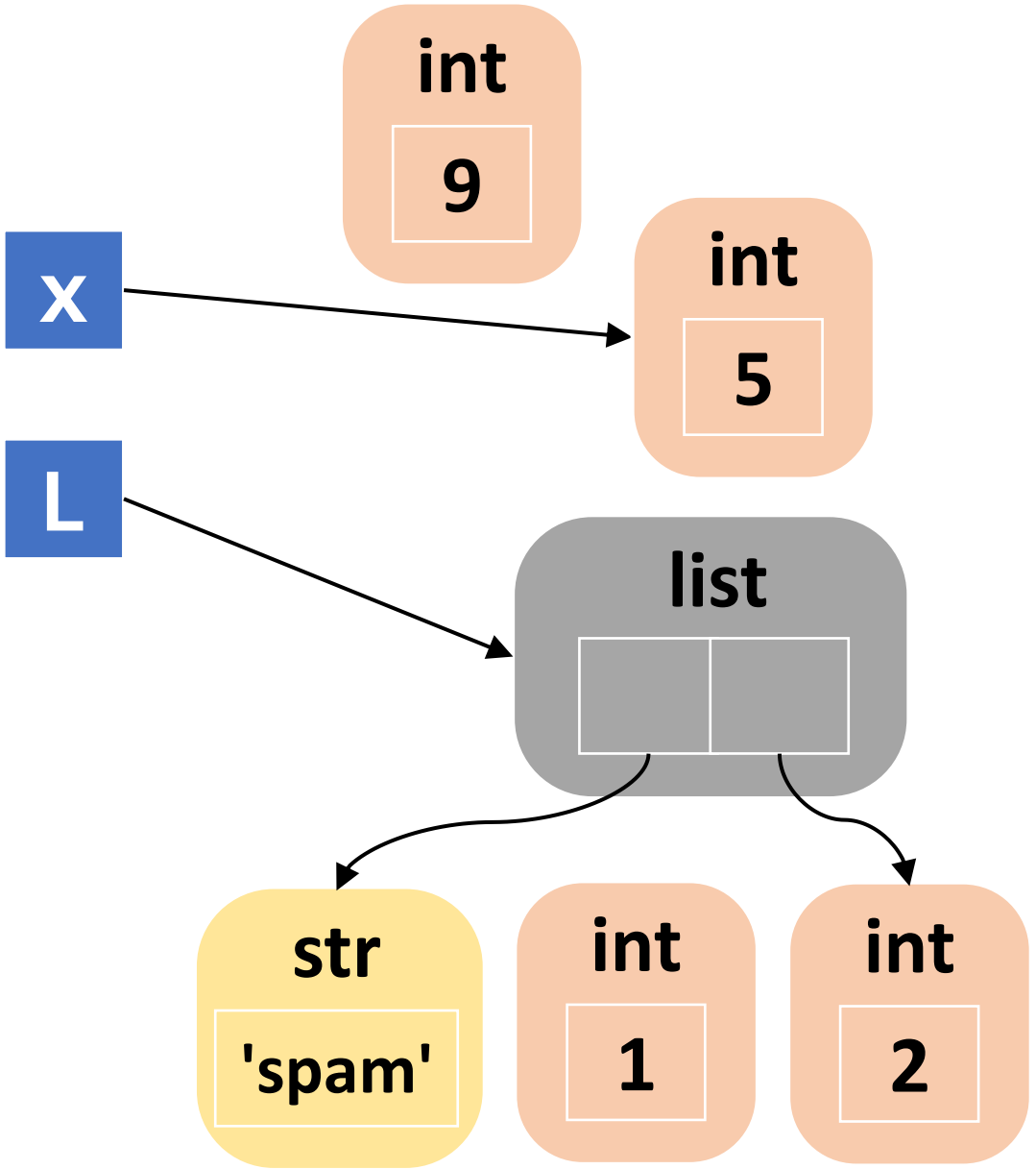
Passing Arguments

```
>>> 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

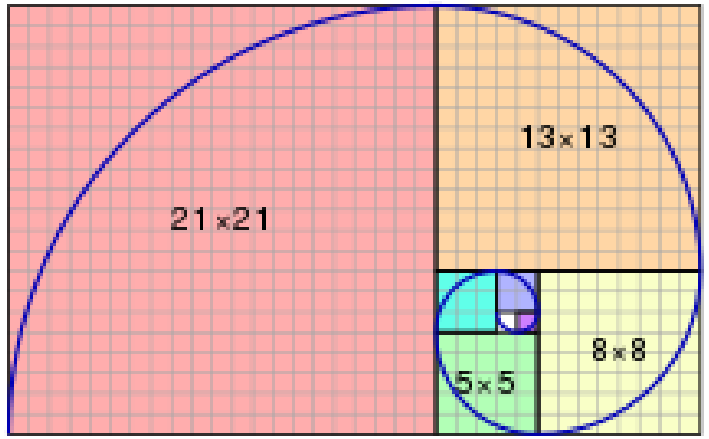
$$f(n) = \begin{cases} n \times f(n-1) & n > 1 \\ 1 & n \leq 1 \end{cases}$$

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

Fibonacci Numbers

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

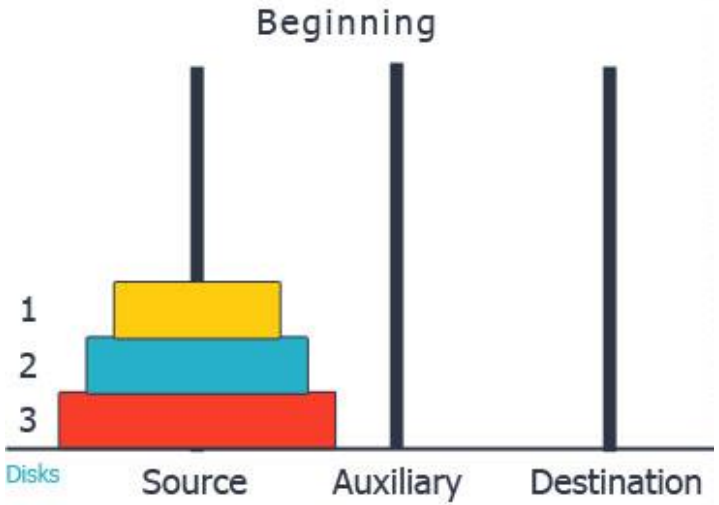
$$f(0) = 0,\ f(1) = 1$$

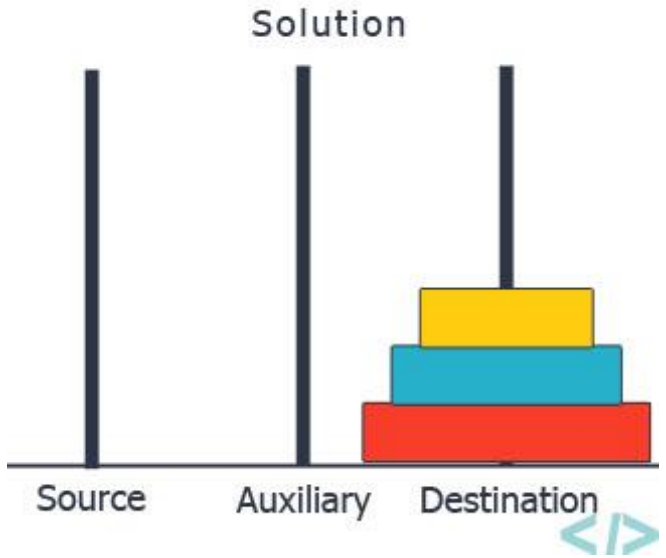



```
def fib(n):
```

Tower of Hanoi

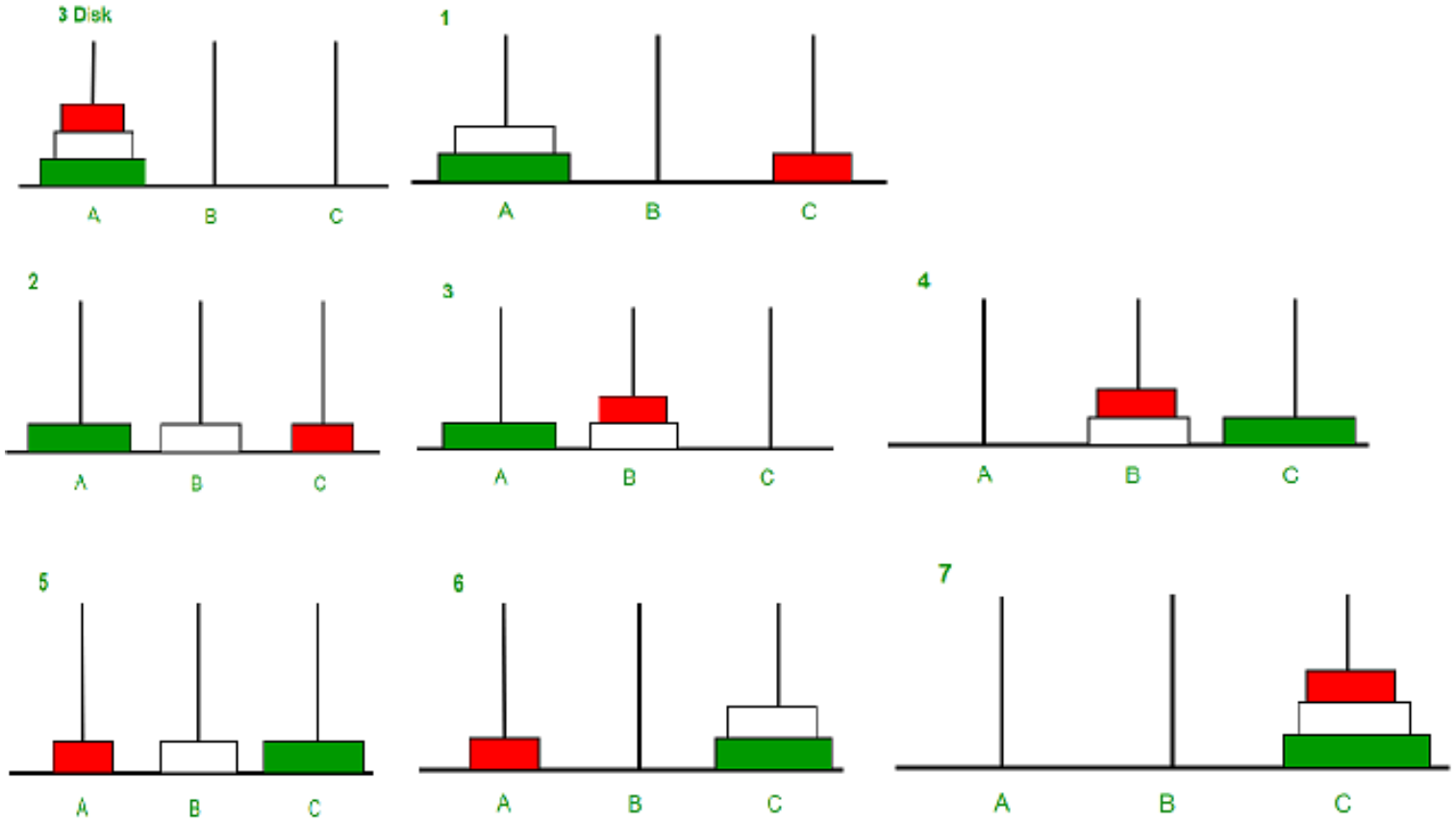
```
def hanoi(n, source, dest, aux):
  if n > 0:
    hanoi(n-1, source, aux, dest)
    print('Move disk', n,'from', \
      source, 'to', dest)
    hanoi(n-1, aux, dest, source)


hanoi(3, 'A', 'C', 'B')
```





Example: Tower of Hanoi

```
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```



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)