C vs. Python Syntax
Spring 2017 – Version 4

C	Python
Purpose: Total control of the computer, at the expense of more detailed programming. Restrictions: None. C gives you direct access to all computer resources. Versions: "K&R" (1978), C89, C90, C11	Purpose: Fast programming, at the expense of efficiency. Restrictions: You don't have control of memory or direct access to RAM. Versions: 1.0 (Jan 1994), 2.0 (Oct 2000), 3.0 (Dec 2008), 3.6 (Dec 2016)

Comment – text that is removed from a program file during pre-processing.
`// Single line comment is all text after a double slash.`
`/*`
`Multi-line comments are any text inside /* and`
`*/`
 	
Docstring – a multiline comment that describes the purpose, parameters, and return value of a function. In CLion type `/**<enter>` to create a docstring.
`/**`
`* Describe the function's purpose.`
`*/`
	Comment – text ignored by the Python interpreter
	`# Single line comment`
	`'''`
	`Multi-line comments are any text inside triple quotes.`
	`'''`
	
	Docstring – a multiline comment that describes the purpose, parameters, and return value of a function. In PyCharm type `"""<enter>` to create a docstring.
	`"""`
	`Explain what the function does … :param alpha: alpha is … :return: the result of …`
	`"""`

Variable – a storage location whose value can change. · Name – Starts with letter, contains only a-z, A-Z, 0-9, _ · Data Type – `char`, `int`, `short int`, `long int`, `float`, `double`, … · Value – (depends on its data type) *Static typing:* A variable must be declared and assigned a data type before it can be used. A variable's datatype is determined by its declaration. A variable's datatype never changes. A variable is assigned a memory location at compile time and its location never changes while the program is running.	**Variable** – a storage location whose value can change. · Name – Starts with letter, contains only a-z, A-Z, 0-9, _ · Data Type – `int`, `float`, `str`, `object` · Value – (depends on its data type) *Dynamic typing:* A variable is automatically created the first time it is used. A variable's datatype is determined by its current value. A variable's datatype can change. A variable is assigned a memory location at run-time and its location can change while the program runs.

Data Type – a description of a memory value

C is not an object-oriented language. There are no *objects* in C.

character (1 byte)	`char`
integer (4 bytes)	`int`
real number (single, 4 bytes)	`float`
real number (double precision, 8 bytes)	`double`
short (16 bit integer)	`short`
long (32 bit integer)	`long`
double long (64 bit integer)	`long long`
positive or negative	`signed`
non-negative modulo 2m	`unsigned`
pointer to type	*`type`*`*`
enumeration constant	`enum tag` {name1=value1, … };
constant (read-only) value	type `const` name;
no value	`void`
create new name for data type	`typedef` type name;

Aggregate Data Types – a collection of values

array (all elements of the same data type)	*`type`* name`[`*`size`*`];`

structure (each element can be of a different data type)	`struct` name {
	datatype field1Name;
	datatype field2Name;
	…
	};

`

Declaration – create a new memory location with a specific data type

```
type variableName;  // contains a "garbage value"
type variableName = initialValue;
type const constantName = initialValue;
```

RULE: All variables, constants, functions, etc. must be declared before they can be used.

Data Type – a description of a memory value

All values in Python are *objects*

Simple data types are:

integer (infinite precision)	`int`
real number (infinite precision)	`float`

Aggregate Data Types – a collection of values

string (zero or more characters)	`str`	`""`
list (zero of more elements)	`list`	`[]`
tuple (zero or more elements)	`tuple`	`()`
dictionary (key à value pairs)	`dict`	`{}`
set (membership)	`set`	`()`
file	`file`	

Declaration – (Python has no concept of declaring the data type of a variable)

Scope – Determines the places in a program where a value can be used.
C has 4 scopes:

Global to program, value was declared in another file	`extern`
Global to file, can't be accessed outside the file's code	`static`
Local to function	
Local to function, persistent between calls	`static`

Scope – Determines the places in a program where a value can be used.
Python has two scopes:
- Local to a function.
- Global to your program.

Expression – a set of values and operators that evaluate to a single value.

struct member operator	`name.member`
struct member through pointer	`pointer->member`
increment, decrement	`++, --`
plus, minus, logical not, bitwise not	`+, -, !, ~`
indirection via pointer, address of object	`*pointer, &name`
cast expression to type	`(type) expr`
size of an object	`sizeof`
multiply, divide, modulus (remainder)	`*, /, %`
add, subtract	`+, -`
left, right shift [bit ops]	`<<, >>`
relational comparisons	`>, >=, <, <=`
equality comparisons	`==, !=`
and [bit op]	`&`
exclusive or [bit op]	`^`
or (inclusive) [bit op]	`|`
logical and	`&&`
logical or	`||`
conditional expression	`expr1 ? expr2 : expr3`
assignment operators	`+=, -=, *=, …`
expression evaluation separator	`,`

Unary operators, conditional expression and assignment operators group right to left; all others group left to right.

Expression – a set of values and operators that evaluate to a single value.
operand operator operand , e.g., (37 + alpha) / (2 ** 3)
- operators : `+ - * / // **`
- order of operations: `( ), **, * / //, + -`
- operators are context sensitive: ("ab" + "def") different from (37 + 12)

Note: C does **not** have an exponentiation operator, use the `pow( )` function.

Statement – a command given to the computer.
C is totally free form, meaning newlines have no meaning. A single line in a C code file can have 0 to an infinite number of statements. Statements must be separated by a semicolon (`;`).
Statements inside braces, `{ … }`, form a statement block.

Statement – a command given to the computer.
Python code is organized one statement per line.

Statements at the same level of indention form a statement block

Assignment Statement – set the value of a variable.

variable = *expression*

Selection - conditional execution of a group of statements.

```
if (expression) {
    statement(s) # if expression is true
}

if (expression) {
    statement(s) # if expression is true
} else {
    statement(s) # if expression is false
}

if (expression1) {
    statement(s) # if expression1 is true
} else if (expression2) {
    statement(s) # if expression1 is false and expression2 is true
} else if (expression3) {
    statement(s) # if expressio1 and expression 2 are false and expression3 is true
} else {
    statement(s) # if expression1 and expression2 and expression3 are all false
}

switch (expression) {
    case const1: statement(s); break;
    case const2: statement(s); break;
    default: statement(s)
}
```

Iteration – repeatedly execute a group of statements.

for loop – general format

```
for (initialize; test_for_continue; modification) {
    …
}
```

(Note: There is no such thing as an *iterable* object in C.)

Assignment Statement – set the value of a variable.

variable = *expression*

Selection - conditional execution of a group of statements.

```
if expression:
    statement(s) # if expression is true

if expression:
    statement(s) # if expression is true
else:
    statement(s) # if expression is false

if expression1:
    statement(s) # if expression1 is true
elif expression2:
    statement(s) # if expression2 is true
elif expression3:
    statement(s) # if expression3 is true
else:
    statement(s)
```

Python does not have a statement equivalent to a `switch` statement

Iteration – repeatedly execute a group of statements.

for loop – general format

```
for variable in iterable_object:
    statement(s)
```

```
for (int j=0; j<10; j++) {
   …
}
for (int j=20; j<30; j++) {
   …
}

for (int j=-5; j<10; j+=2) {
   …
}
```

while loop

```
while (expression) {
   …
}
```

repeat-until loop

```
do {
   …
} while(expression);
```

```
for variable in range(10):
   …

for variable in range(20,30):
   …


for variable in range(-5,10,2):
   …
```

while loop

```
while expression:
   …
```

repeat-until loop

```
while True:
   …
   if expression:
      break
```

Function Definition – create a group of statements to be executed as a unit.

```
datatype name ( parameters ) {
   statement(s)
}
```

```
def name ( parameters ):
   statement(s)
   …
```

Import Statement – load the variables and functions defined in another file.

include library file `#include <name>`
include user file `#include "name"`

`import filename` - import all definitions in the specified file
`from filename import name` - import the specified items
`from filename import name as newName` - import and rename

Console input – allow a user to input a value from the console

```
scanf(formatString, addressOfVariable)

printf("Enter an integer number: ");
scanf("%d", &value);
```

```
variable = input("prompt")
```

Codes for Formatted output - format specifier: %-+ 0w:pmc

-	left justify
+	print with sign
space	print space if no sign
0	pad with leading zeros
w	minimum field width
p	precision
m	conversion character: h short, l long, L long double
c	conversion character:

d,i	integer	u	unsigned		
o	octal	x,X	hexadecimal		
c	single char	s	char string		
f	float	e,E	exponential	lf	double
p	pointer				

Example:

```
printf("Fred weights %.1f lbs and is %d years old",
       weight, age)
```

Formatted output: format specifications:

`string` output:

`{:10}` – print exactly 10 characters, (default: left justified)

`{:<10}` – print exactly 10 characters, left justified.

`{:>12}` – print exactly 12 characters, right justified.

`{:^8}` – print exactly 8 characters, center justified.

`int` output:

`{:10d}` – print decimal integer using exactly 10 characters, right justified.

`{:8b}` – print binary integer using exactly 8 characters, right justified.

`{:6x}` – print hexadecimal integer using exactly 6 characters, right justified.

`float` output:

`{:10.2f}` – exactly 2 digits after the decimal point; use exactly 10 characters,

`{:12g}` – significant digits after the decimal point; use exactly 12 characters,

`{:8.2e}` – exponential notation; 2 digits accuracy; use exactly 8 characters,

Example:

```
print("Fred weights {:.1f} lbs and is {:d} years old"
      .format(weight, age))
```

Example Program

```c
// ============================================================
// A simple example C program that finds prime numbers.
// Created by Dr. Wayne Brown on 1/3/2017.
// ============================================================

#include <math.h>
#include <stdio.h>

#define TRUE    1
#define FALSE   0

// ------------------------------------------------------------
int is_prime(int n) {
    /**
     * Determines if the value of n is prime.
     * Returns True if n is prime, False otherwise
     */
    int max_divisor = (int) ceil(sqrt(n));
    for (int divisor = 2; divisor <= max_divisor; divisor++) {
        if (n % divisor == 0) {
            return FALSE;
        }
    }

    return TRUE;
}

// ------------------------------------------------------------
int main() {
    // Print all of the prime numbers <= 100
    for (int value = 3; value <= 100; value++) {
        if (is_prime(value)) {
            printf("%d\n", value);
        }
    }
}
```

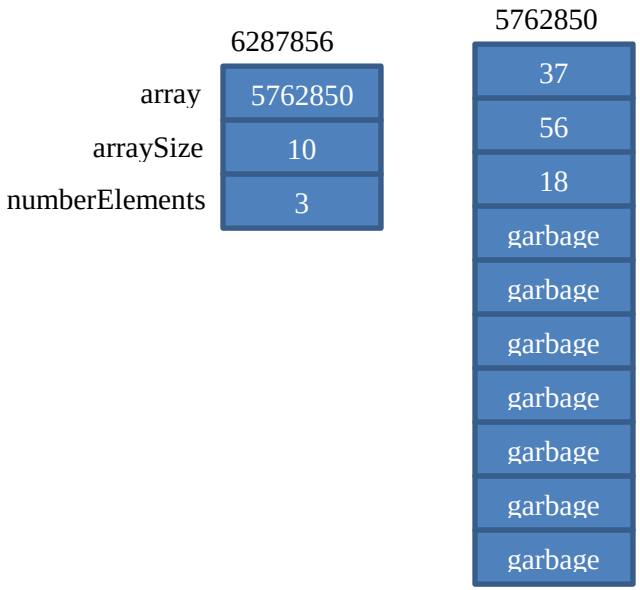
Example Program

```python
#!/usr/bin/env python

"""
A simple example Python program that finds prime numbers.
"""
#
# ============================================================================
from math import sqrt, ceil

# Metadata
__author__ = "Wayne Brown"


# ----------------------------------------------------------------------------
def is_prime(n):
    """
    Determines if the value of n is prime
    :param n: is an integer
    :return: True if n is prime, false otherwise
    """
    for divisor in range(2, ceil(sqrt(n))):
        if n % divisor == 0:
            return False

    return True


# ----------------------------------------------------------------------------
def main():
    # Print all of the prime numbers <= 100
    for value in range(3, 101):
        if is_prime(value):
            print(value)


# ----------------------------------------------------------------------------
if __name__ == "__main__":
    main()
```

The remainder of these pages describes only C syntax.

A variable has a *memory address* and a *value*. In the following diagrams, a variable is represented as a box. Inside the box is the variable's *value*. The variable's *address* is displayed above the box. The variable's *name* is displayed to the left of the box. For example:

A visual diagram of the memory for an `ArrayList`:

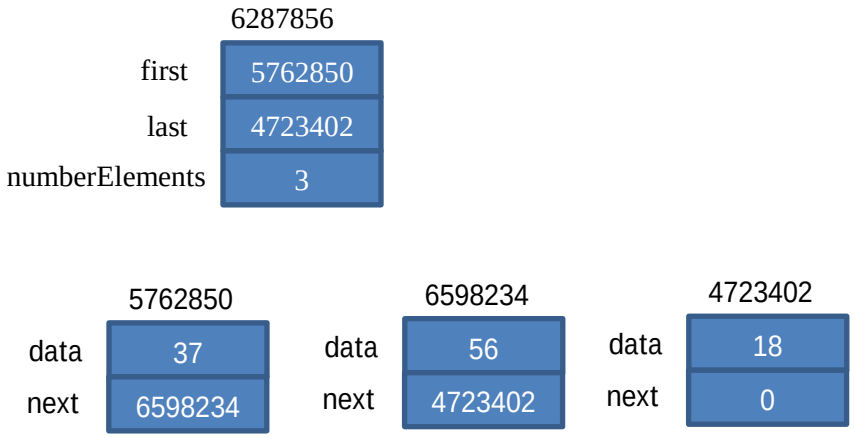


A C description of an `ArrayList`:

```c
// Define the data type for each element of the list.
typedef int ElementType;

// Define the structure that holds the meta-data for one list.
typedef struct ArrayList {
    ElementType   *array;
    int            arraySize;
    int            numberElements;
} ArrayList;

// Create the structure that holds the list information
ArrayList * myList = (ArrayList *) malloc(sizeof(ArrayList));

// Allocate a block of memory to hold the elements
myList->array = (ElementType *) malloc(sizeof(ElementType) * 10);
```

C Syntax:	Evaluates to these values for the example to the left:
myList	6287856
myList->array	5762850
myList->arraySize	10
myList->numberElements	3
myList->array[0]	37
myList->array[1]	56
myList->array[2]	18
myList->array[3]	(garbage)

A visual diagram of the memory for a `LinkedList`:



A C description of a `LinkedList`:

```c
// Define the data type for each element of the list.
typedef int ElementType;

// Define on node of the linked list
typedef struct node {
    ElementType   data;
    struct node * next;
} Node;

// Define the meta-data that stores the linked list.
typedef struct linkedList {
    Node * first;
    Node * last;
    int    numberElements;
} LinkedList;

// Create the structure that holds the list information
LinkedList * myList = (LinkedList *) malloc(sizeof(LinkedList));

// Allocate a block of memory to hold one node of the list
Node * oneNode = (Node *) malloc(sizeof(Node));
```

C Syntax:	Evaluates to these values for the example to the left:
myList	6287856
myList->first	5762850
myList->last	4723402
myList->numberElements	3
myList->first->data	37
myList->first->next	6598234
myList->last->data	18
myList->last->next	0
Node * aNode;	(aNode contains garbage)
aNode = myList->first	5762850
aNode->data	37
aNode->next	6598234
aNode->next->data	56

Using the CLion debugger:

- To see the contents of an array if you only have a pointer to the array, in the "variables panel" create a new "watch" and cast the pointer to an array.
  - For example, if you have declared: `TypeXyz *alpha;`
    and `alpha` points to a valid memory block (because you malloc'ed memory or assigned it to an existing array),
    then the CLion *watch* would be `(TypeXyz (*) [size]) alpha`, where `size` it the number of elements you want CLion to be able to display.

bit manipulation in C:

Operator	What it does:
`a << n`	bit-wise **left shift** the bits in `a` by `n` bits
`a >> n`	bit-wise **right shift** the bits in `a` by `n` bits
`a & b`	bit-wise **logical AND** on corresponding bits in `a` and `b`
`a | b`	bit-wise **logical OR** on corresponding bits in `a` and `b`
`a ^ b`	bit-wise **logical XOR** on corresponding bits in `a` and `b`
`~a`	bit-wise **logical NOT** (complement) bits in `a`

bit manipulation in Python: (Identical to C syntax)

Operator	What it does:
`a << n`	bit-wise **left shift** the bits in `a` by `n` bits
`a >> n`	bit-wise **right shift** the bits in `a` by `n` bits
`a & b`	bit-wise **logical AND** on corresponding bits in `a` and `b`
`a | b`	bit-wise **logical OR** on corresponding bits in `a` and `b`
`a ^ b`	bit-wise **logical XOR** on corresponding bits in `a` and `b`
`~a`	bit-wise **logical NOT** (complement) bits in `a`