Operators

Below is a table of operators in C++, Java, and Python. Assignment operators are in red. Some operators dealing with type checking and casting and storage allocation have been omitted.

Name and symbol		C++	Java	Python	Example (these return `True`)
Assignment[1]	: = :=	`a = b`	`a = b`	`a = b`	`a = True; a`
Addition	+	`a + b` `a += b`	`a + b` `a += b`	`a + b` `a += b`	`6 + 2 == 8`
Subtraction	−	`a - b` `a -= b`	`a - b` `a -= b`	`a - b` `a -= b`	`6 - 2 == 4`
Multiplication	×	`a * b` `a *= b`	`a * b` `a *= b`	`a * b` `a *= b`	`6 * 2 == 12`

Division[2]	÷	`a / b`
		`a /= b`
			`a / b`
			`a /= b`
				`a / b`
				`a /= b`
				`a // b`
				`a //= b`
					`5 / 2 == 2.5`
					`5 // 2 == 2`

Modulo[3]	mod	`a % b` `a %= b`	`a % b` `a %= b`	`a % b` `a %= b`	`7 % 3 == 1`
Exponent[4]	a^b			`a ** b` `a **= b`	`4 ** 2 == 16`
Unary plus[5]		`+a`	`+a`	`+a`	`a = 2; +a == 2`
Unary minus	−	`-a`	`-a`	`-a`	`a = 2; -a == -2`
Increment[6]		`++a` `a++`	`++a` `a++`		
Decrement[7]		`--a` `a--`	`--a` `a--`		
Equality	=	`a == b`	`a == b`	`a == b`	`6 == 6`
Inequality	≠	`a != b` `a not_eq b`	`a != b`	`a != b` `a <> b`	`6 != 7`

[1] In C++ and Java, assignment operators return the value of the variable after assignment (so `(a = b) == b`). In Python, they do not.

[2] In C++ and Java, `/` performs floating-point division on floats and integer division on integers. In Python, `/` performs floating-point division and `//` performs floor division. Floor division is like integer division, except that it always rounds down but integer division rounds toward zero. For example, in C++, `-5 / 2 == -2`, but in Python, `-5 // 2 == -3`.

[3] In C++ and Java, the result of a modulo operation has the same sign as the dividend (`-5 % 4 == -1` because $-5 = -(1 \times 4 + 1)$). In Python, it has the same sign as the divisor (`-5 % 4 == 3` because $-5 = -2 \times 4 + 3$). In Java, Python's behavior can be obtained using the `Math.floorMod` function.

[4] In C++, `#import <cmath>` and use `std::pow(a, b)`. In Java, use `Math.pow(a, b)`. In Python, `math.pow(a, b)` is also available after `from math import pow` and may produce slightly different floating-point results.

[5] You probably do not want this, especially in Python. In statically typed languages it is occasionally useful because it casts its operand to an `int`; for example, in C/C++, `+'a'` is the integer 97.

[6] When used in an expression, `++a` returns the value of `a` after incrementing whereas `a++` returns the value of `a` before incrementing. `int a = 6, b = ++a, c = a++; printf("%i %i %i\n", a, b, c);` prints `6 7 7`. In Python, the increment operator is unavailable; use `a += 1`.

[7] When used in an expression, `--a` returns the value of `a` after decrementing whereas `a--` returns the value of `a` before decrementing. `int a = 6, b = --a, c = a--; printf("%i %i %i\n", a, b, c);` prints `4 5 5`. In Python, the decrement operator is unavailable; use `a -= 1`.

Name and symbol	C++	Java	Python	Example (these return `True`)
Greater than $>$	`a > b`	`a > b`	`a > b`	`88 > 4`
Less than $<$	`a < b`	`a < b`	`a < b`	`8 < 43`
Greater than or equal $\geq$	`a >= b`	`a >= b`	`a >= b`	`88 >= 4` `4 >= 4`
Less than or equal $\leq$	`a <= b`	`a <= b`	`a <= b`	`8 <= 43` `8 <= 8`
Boolean NOT $\sim$ $\neg$	`!a` `not a`	`!a`	`not a`	`not False`
Boolean AND $\wedge$	`a && b` `a and b`	`a && b`	`a and b`	`True and True`
Boolean OR $\vee$	`a || b` `a or b`	`a || b`	`a or b`	`True or False`

Bitwise NOT	`~a`
	`compl a`
		`~a`	`~a`	`~0b100101&0b111111==0b11010`
				`~37 & 0x3F == 26`
				Be careful when using this with signed integers!
				`~37 == -38`

Bitwise AND	`a & b`
	`a bitand b`
	`a &= b`
	`a and_eq b`
		`a & b`
		`a &= b`
			`a & b`
			`a &= b`
				`0b110101&0b101111==0b100101`

Bitwise OR	`a | b`
	`a bitor b`
	`a |= b`
	`a or_eq b`
		`a | b`
		`a |= b`
			`a | b`
			`a |= b`
				`0b100101|0b101000==0b101101`
				`37 | 40 == 45`

Bitwise XOR	`a ^ b`
	`a xor b`
	`a ^= b`
	`a xor_eq b`
		`a ^ b`
		`a ^= b`
			`a ^ b`
			`a ^= b`
				`0b100101^0b101111==0b001010`
				`37 ^ 47 == 10`

Left shift	`a << n` `a <<= n`	`a << n` `a <<= n`	`a << n` `a <<= n`	`0b100101 << 2 == 0b10010100` `37 << 2 == 148`
Right shift	`a >> n` `a >>= n`	`a >> n` `a >>= n`	`a >> n` `a >>= n`	`0b100101 >> 2 == 0b1001` `37 >> 2 == 9`
Subscript a_b	`a[b]`	`a[b]`	`a[b]`	`[True, False][0]`
Indirection[8] Dereference	`*a`		`*a`	
Reference	`&a`			
Member[9]	`a.b` `a->b`	`a.b`	`a.b`	
Function call	`f(a)`	`f(a)`	`f(a)`	
Scope resolution	`::`	`::`		
Comma	`a, b`			
Ternary[10]	`a ? b : c`	`a ? b : c`	`b if a else c`	`True if 3 > 2 else False`
User-defined literal	`"a"_b`			

[9] In C++, `a->b` dereferences the pointer `a` to an object with a member `(*a).b`.
[10] Shorthand for `if(a) { b; } else { c; }`.

C++ Demo

The following C++ program demonstrates templates, operator overloading, and operations on vectors and deques:

```cpp
#include <iostream>
#include <sstream>
#include <vector>
#include <deque>
#include <algorithm>
#include <string>

template<typename Container>
void print(const Container& v) {
    std::ostringstream ss; // include sstream
    ss << '{';
    for(auto e : v) {
        ss << e << ", ";
    }
    if(v.size()) {
        ss.seekp(-2, ss.cur); // go back over last ,
    }
    ss << "}\n";
    std::cout << ss.str(); // include iostream
}

class IntAndStrings {
public:
    long long int number;
    std::vector<std::string> strings; // include vector and string

    // overload < operator so we can sort
    bool operator<(const IntAndStrings& rhs) const {
        if(number == rhs.number) { // sort on strings if numbers are equal
            unsigned int minStringCount = std::min(strings.size(),
            ↪ rhs.strings.size()); // include algorithm
            for(int i = 0; i < minStringCount; i++) { // sort on strings
                int strcmp = strings.at(i).compare(rhs.strings.at(i));
                if(strcmp) {
                    return strcmp < 0;
                }
            }
            // all strings were equal
            return strings.size() < rhs.strings.size(); // sort on number of
            ↪ strings.
        }
        // numbers are not equal so did not sort on strings
```

```
41          return number < rhs.number;
42      }
43
44      // overload << operator so we can print - this gets defined outside the
        ↪  class
45      friend std::ostream& operator<<(std::ostream&, const IntAndStrings&);
46
47      // Constructors
48      IntAndStrings() {}
49      IntAndStrings(long long int i) : number(i) {}
50      IntAndStrings(long long int i, std::vector<std::string> sv) : number(i),
        ↪  strings(sv) {}
51      IntAndStrings(long long int i, std::string s) : number(i) {
52          std::vector<std::string> sv = {s};
53          strings = sv;
54      }
55
56      // Methods
57      void add(std::string s) {
58          strings.push_back(s);
59      }
60      void add(std::vector<std::string> s) {
61          strings.insert(strings.end(), s.begin(), s.end());
62      }
63  };
64
65  // Definition for friend operator to IntAndStrings
66  std::ostream& operator<<(std::ostream& stream, const IntAndStrings& ias) {
67      std::ostringstream ss;
68      ss << ias.number << ": [";
69      for(std::string s : ias.strings) {
70          ss << '"' << s << "\",";
71      }
72      if(ias.strings.size()) {
73          ss.seekp(-1, ss.cur);
74      }
75      ss << "]";
76      stream << ss.str();
77      return stream;
78  }
79
80  int main() {
81      std::vector<int> v = {1, 2}; // include vector
82      v.push_back(3); // {1, 2, 3}
83      print(v);
84      v.pop_back(); // {1, 2}
85      print(v);
```

```
86      v.push_back(4); // {1, 2, 4}
87      print(v);
88      v.at(1) = 3; // {1, 3, 4}
89      print(v);
90      v.erase(v.begin()); // {3, 4}
91      print(v);
92      std::vector<int> v2 = {7, 2, 24, 1, -6};
93      v.insert(v.begin() + 1, v2.begin(), v2.end()); // {3, 7, 2, 24, 1, -6, 4}
94      print(v);
95      v.insert(v.begin() + 2, 0); // {3, 7, 0, 2, 24, 1, -6, 4}
96      print(v);
97      std::cout << v.empty() << '\n'; // 0 (false)
98      if(std::find(v.begin(), v.end(), -6) != v.end()) { // include algorithm
99          std::cout << "v contains -6\n";
100     }
101     std::cout << "max is " << *max_element(v.begin(), v.end()) << '\n'; //
        ↪ include algorithm
102     v.erase(min_element(v.begin(), v.end())); // erase min element
103     print(v);
104     if(std::find(v.begin(), v.end(), -6) == v.end()) { // include algorithm
105         std::cout << "v does not contain -6\n";
106     }
107     std::sort(v.begin(), v.end()); // include algorithm
108     print(v); // {0, 1, 2, 3, 4, 7, 24}
109     std::sort(v.begin(), v.end(), std::greater<int>()); // include algorithm
110     print(v); // {24, 7, 4, 3, 2, 1, 0}
111     std::sort(v.rbegin(), v.rend(), std::greater<int>()); // sort from "reverse
        ↪ begin" to "reverse end" using "greater" - equivalent to normal sort
112     print(v); // {0, 1, 2, 3, 4, 7, 24}
113
114
115     std::deque<IntAndStrings> q = {IntAndStrings(1, "this is string"),
        ↪ IntAndStrings(2)}; //include deque
116     print(q); // {1: ["this is string"], 2: []}
117     q.at(0).add("another string");
118     print(q); // {1: ["this is string","another string"], 2: []}
119     std::vector<std::string> someStrings = {"string 1", "string 2", "string 3"};
120     q.push_front(IntAndStrings(2, someStrings)); // this is why one would use a
        ↪ deque instead of a vector
121     q.at(0).add("I am now greater than the other one with int 2");
122     q.at(2).add(someStrings);
123     print(q); // {2: ["string 1","string 2","string 3","I am now greater than
        ↪ the other one with int 2"], 1: ["this is string","another string"], 2:
        ↪ ["string 1","string 2","string 3"]}
124     std::sort(q.begin(), q.end());
```

```
125     print(q); // {1: ["this is string","another string"], 2: ["string 1","string
        ↪  2","string 3"], 2: ["string 1","string 2","string 3","I am now greater
        ↪  than the other one with int 2"]}
126     q.erase(q.begin(), q.begin() + 2); // this is faster than with a vector
127     print(q); // {2: ["string 1","string 2","string 3","I am now greater than
        ↪  the other one with int 2"]}
128
129     return 0;
130 }
```

C++ User-Defined Literals

Potentially useful but probably not:

```cpp
#include <iostream>

#ifndef M_PI
    #define M_PI 3.14159265358979323846264338
#endif

long double operator ""_pi(long double x) {
    return x * M_PI;
}
long double operator ""_pi(unsigned long long int x) {
    return (long double) x * M_PI;
}

long double operator ""_tau(long double x) {
    return x * M_PI * 2;
}
long double operator ""_tau(unsigned long long int x) {
    return (long double) x * M_PI * 2;
}

int main() {
    std::cout << 2_pi << " = " << 1_tau << " < " << 2.2e0_pi << " = " <<
    ↪  11e-1_tau << std::endl;
    // prints: 6.28319 = 6.28319 < 6.9115 = 6.9115
    return 0;
}
```

The following parameter lists are allowed on literal operators:

- (const char *)
- (unsigned long long int)
- (long double)
- (char)
- (wchar_t)
- (char16_t)
- (char32_t)
- (const char *, std::size_t)
- (const wchar_t *, std::size_t)
- (const char16_t *, std::size_t)
- (cosnt char32_t *, std::size_t)