Revised: 1/12/2019

Arithmetic and Type Conversion

"On two occasions I have been asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question."

-Charles Babbage

Michael C. Hackett

Computer Science Department

Community College of Philadelphia

Lecture Topics

- Numeral Systems
  - Binary
  - Octal
  - Hexadecimal
  - Conversion between types
- Binary Addition
  - Arithmetic Overflow
- Signed and Unsigned Integers
- Binary Fractions
- Arithmetic Operators
  - Precedence Rules
  - Augmented Assignment
- Rounding and Python's Math Module
- Concatenation and Appending
- Type Conversion
  - Type Coercion
  - Type Casting

Colors/Fonts

- Variable Names – **Brown**
- Standard data types – **Fuchsia**
- Literals – **Blue**
- Keywords – **Orange**
- Operators/Punctuation – **Black**
- Function Names – **Purple**
- Comments – **Gray**
- Module Names – **Pink**

Source Code – **`Consolas`**

Output – `Courier New`

Numeral Systems

- In computer science, it's common to see different numeral systems used.
- A ***numeral system*** is a form of notation for expressing numbers using a certain set of symbols or digits.
- The number of unique digits used by a numeral system is referred to as its **base** or *radix*.
- A ***computer number format*** is how numeric values are represented in computer hardware and software.

Decimal / Base 10

- In every day life, we use the decimal number system.
  - Otherwise known as *Base 10*.

- Numbers are represented using ten different digits or symbols
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9

- In the decimal system, we have the ones, tens, hundreds (and so on) places.

109

100 1

10

Decimal / Base 10

- Another way to look at it:

109

10^2 10^1 10^0

- $\boxed{1 \text{ x } 10^2} + \boxed{0 \text{ x } 10^1} + \boxed{9 \text{ x } 10^0}$
- 100 + 0 + 9
- 109

Binary / Base 2

- In the binary number system, numbers are represented using two different digits/symbols.
  - 0 and 1
  - “Bits”

- Since transistors operate in on and off states, binary is the ideal number system for computing.

Binary / Base 2

- The first eleven (non-negative) decimal and binary numbers.

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011

Binary / Base 2

- The binary number system does not have the ones, tens, hundreds (and so on) places.
  - Binary has no concept of “ten” or “one hundred”

01101101

2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0

Notation

- 100
  - Is this decimal (“*one hundred*”) or binary (“*one zero zero*”)?

- The number’s radix is in subscript after the number’s digits.
  - The number is usually in parentheses.

$(100)_2$

Indicates this is a binary number “one zero zero”

$(100)_{10}$

Indicates this is a decimal number “one hundred”

Units (Decimal)

Unit	Size	Unit Abbreviation
bit	1	b
byte	8 bits	B
kilobyte	10^3 bytes (1000 bytes)	kB (proper) but also KB
megabyte	10^6 bytes (1000 kilobytes)	MB
gigabyte	10^9 bytes (1000 megabytes)	GB
terabyte	10^{12} bytes (1000 gigabytes)	TB
petabyte	10^{15} bytes (1000 terabytes)	PB
exabyte	10^{18} bytes (1000 petabytes)	EB
zettabyte	10^{21} bytes (1000 exabytes)	ZB
yottabyte	10^{24} bytes (1000 zettabytes)	YB

Units (Binary)

Unit	Size	Unit Abbreviation
bit	1	b
byte	8 bits	B
kibibyte	2^{10} bytes (1024 bytes)	KiB
mebibyte	2^{20} bytes (1024 kibibytes)	MiB
gibibyte	2^{30} bytes (1024 mebibytes)	GiB
tebibyte	2^{40} bytes (1024 gibibytes)	TiB
pebibyte	2^{50} bytes (1024 tebibytes)	PiB
exbibyte	2^{60} bytes (1024 pebibytes)	EiB
zebibyte	2^{70} bytes (1024 exbibytes)	ZiB
yobibyte	2^{80} bytes (1024 zebibytes)	YiB

Octal / Base 8

- In the octal number system, numbers are represented using eight different digits/symbols.
  - 0, 1, 2, 3, 4, 5, 6, and 7
- Octal has been a convenient number system in computer science, because every octal digit can be represented by three bits.

Decimal	Binary	Octal
0	000 **000**	0
1	000 **001**	1
2	000 **010**	2
3	000 **011**	3
4	000 **100**	4
5	000 **101**	5
6	000 **110**	6
7	000 **111**	7
8	**001 000**	10
9	**001 001**	11
10	**001 010**	12
11	**001 011**	13
12	**001 100**	14
13	**001 101**	15
14	**001 110**	16
15	**001 111**	17

Hexadecimal / Base 16

- In the hexadecimal number system, numbers are represented using sixteen different digits/symbols.
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F

- Hex has also been a convenient number system in computer science, because every hex digit can be represented by four bits.

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Converting Binary to Decimal

01101101

2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0

- $0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
- $0 + 64 + 32 + 0 + 8 + 4 + 1$
- $\boxed{109}$

Converting Decimal to Binary

- Keep dividing the decimal number by 2.

- Make note of the remainders.
  - The remainder is always 1 or 0

- Stop when the quotient is 0.

Converting Decimal to Binary

- $\boxed{109}$ / 2 = 54 with a remainder of 1
- 54 / 2 = 27 with a remainder of 0
- 27 / 2 = 13 with a remainder of 1
- 13 / 2 = 6 with a remainder of 1
- 6 / 2 = 3 with a remainder of 0
- 3 / 2 = 1 with a remainder of 1
- 1 / 2 = $\boxed{0}$ with a remainder of 1

1101101

Converting Decimal to Binary

- $\boxed{72}$ / 2 = 36 with a remainder of 0
- 36 / 2 = 18 with a remainder of 0
- 18 / 2 = 9 with a remainder of 0
- 9 / 2 = 4 with a remainder of 1
- 4 / 2 = 2 with a remainder of 0
- 2 / 2 = 1 with a remainder of 0
- 1 / 2 = $\boxed{0}$ with a remainder of 1

1001000

Converting Binary/Octal

$$(\boxed{01}\boxed{101}\boxed{101})_2$$

$$(155)_8$$

Converting Binary/Hex

$$(\boxed{0110}\boxed{1101})_2$$

$$(6D)_{16}$$

Binary Addition

$$\begin{array}{r} 0\ 1 \\ +\ 1\ 0 \\ \hline 1\ 1 \end{array}$$

- Start by adding the right-most bits first.
  - 1 and 0 are added together resulting in 1.
- 0 and 1 are then added together, also resulting in 1.
  - Thus, the result of the addition is 11_2 (or 3_{10}).

Binary Addition

$$\begin{array}{cccc} & 1 & 1 & \\ & & 0 & 1 \\ + & & 1 & 1 \\ \hline & 1 & 0 & 0 \end{array}$$

- The right-most bits, 1 and 1, are added together resulting in 10 (or 2_{10}).
  - The 0 is placed in the final result and the 1 is carried over.
- Next, 1, 0, and 1 are added together resulting, again, in 10.
  - The 0 is placed in the final result and the 1 is carried over.
- Finally, add the carried 1 with 0 and 0 (not shown) which results in 1.
- The calculated sum of these two binary numbers is 100_2 (or 4_{10}).

Arithmetic Overflow

- A computer will typically allot a finite amount of space for representing a number.
- Let’s consider a particular computer system that limits us to using only four bits for storing numbers in memory.

$$
\begin{array}{cccccc}
 & 1 & & & & \\
 & & 1 & 0 & 0 & 1 \\
+ & & 1 & 0 & 1 & 0 \\
\hline
 & 1 & \underline{0} & \underline{0} & \underline{1} & \underline{1}
\end{array}
$$

- Only four bits of this result (beginning with the right-most bit) will be considered.

Signed and Unsigned Numbers

- In computing, there exists signed and unsigned numbers.
  - ***Signed*** numbers can be positive or negative.
  - ***Unsigned*** numbers can only be positive.

- Consider a number 8 bits (1 byte) long.
  - The range of values we can represent is 00000000 through 11111111
    - Or, 0 through 255 (in base 10/decimal.)
  - Here, we are dealing with unsigned numbers.
    - We can’t represent negative numbers in memory this way.

Sign Bits

- When using sign bits, the first bit is the sign and the remaining bits are used to represent the number.
  - 1 means negative and 0 means positive.

Binary	Unsigned Integer	Signed Integer
0 00	0	0
0 01	1	1
0 10	2	2
0 11	3	3
1 00	4	-0
1 01	5	-1
1 10	6	-2
1 11	7	-3

Sign Bits

- There are some limitations to this format:
  - Can’t be used for arithmetic.
  - Negative/Positive zero.

- The preferred format for negative integers is two’s complement notation.

Two's Complement

- ***Two's Complement*** is a format for expressing positive and negative binary numbers.

- Apply two's complement by:
  - Flipping the bits.
  - Adding one.

$$\begin{array}{cccc} & 1 & 0 & 1 \\ & \downarrow & \downarrow & \downarrow \\ & 0 & 1 & 0 \\ + & & & 1 \\ \hline & 0 & 1 & 1 \end{array}$$

Two's Complement

Binary	Two's Complement
000	000
001	111
010	110
011	101
100	100
101	011
110	010
111	001

Two's Complement

Binary	Unsigned Integer	Two's Complement	Signed Integer
000	0	000	0
001	1	111	-1
010	2	110	-2
011	3	101	-3
100	4	100	-4
101	5	011	3
110	6	010	2
111	7	001	1

Binary Fractions

- While 101.11 is a perfectly valid base 2 number, we have no way to specify a “binary point” in memory.

$$\begin{array}{cccccc} 1 & 0 & 1 & . & 1 & 1 \\ 2^2 & 2^1 & 2^0 & & 2^{-1} & 2^{-2} \end{array}$$

$$(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (1 \times 2^{-2})$$

$$(1 \times 4) + (0 \times 2) + (1 \times 1) + (1 \times 0.5) + (1 \times 0.25)$$

$$4 + 0 + 1 + 0.5 + 0.25 = 5.75_{10}$$

Binary Fractions

- Two such number formats for representing fractionals are:
  - Fixed Point Notation
  - Floating Point Notation

- Floating point is the more common of the two.

Fixed Point Format

- A specific number of bits allocated for digits to the left of the binary point.
- A specific number of bits allocated for digits to the right of the binary point.

Binary	Decimal
00**0**	0.0
00**1**	0.5
01**0**	1.0
01**1**	1.5
10**0**	2.0
10**1**	2.5
11**0**	3.0
11**1**	3.5

Binary	Decimal
0**00**	0.00
0**01**	0.25
0**10**	0.50
0**11**	0.75
1**00**	1.00
1**01**	1.25
1**10**	1.50
1**11**	1.75

Floating Point Format

- In floating-point, non-integers are represented via scientific notation.
  - The decimal number 12.41 could be re-written as 1.241×10^1
  - The decimal number -0.003 could be rewritten as -3.0×10^{-2}

- The binary number 1001.1 is expressed in scientific notation as $1.0011_2 \times 2^3$
  - Where $2^3 = 4 = 100_2$
  - The number 1.0011 is referred to as the significand or *mantissa*; The digits to the right of the binary point are called the *fraction*.

Floating Point Format

$$1.01101 \times 2^{3} = 1011.01$$

$$-1.11 \times 2^{-1} = -0.111$$

- The placement of the binary point “floats” to where it needs to be.
- Memory space is allotted for (using 16 bits):
  - A sign bit – 1 bit
  - The exponent – 5 bits
  - The mantissa – 10 bits

Floating Point Format

- (This example will use 16 bits to store 11.25, or 1.01101×2^3)
- This number is positive, so the sign bit is zero.

Sign Bit	Exponent	Fraction
0		

Floating Point Format

- (This example will use 16 bits to store 11.25, or 1.01101×2^3)
- The exponent is expressed as 2^{b-1} more than the exponent's actual value.
  - Where *b* is number of bits allotted for the exponent.
  - $3 + 2^4 = 19 = 10011_2$

Sign Bit	Exponent	Fraction
0	10011	

Floating Point Format

- (This example will use 16 bits to store 11.25, or 1.01101×2^3)
- The mantissa is supposed to start with “1.” so only the fractional bits will be stored.

Sign Bit	Exponent	Fraction
0	10011	0110100000

Floating Point Format

- Working backwards with a different number:

Sign Bit	Exponent	Fraction
1	10100	0110100000

- A 1 in the sign bit means the number is negative.
- Exponent: $10100_2 = 20$ -> $20 - 2^4 = 4$
- Mantissa: 1.01101

$$-1.01101_2 \times {2_{10}}^4 = -1.01101_2 \times 1000_2 = -10110.1_2 = -22.5_{10}$$

Floating Point Format

- Half Precision – 16 bits
  - One sign bit
  - Five exponent bits
  - Ten fractional bits
- Single Precision – 32 bits
  - One sign bit
  - Eight exponent bits
  - Twenty-three fractional bits
- Double Precision – 64 bits
  - One sign bit
  - Eleven exponent bits
  - Fifty-two fractional bits

Arithmetic Operators

- Addition: **+**
- Subtraction: **-**
- Multiplication: *****
- Float Division: **/**
- Integer Division: **//**
- Mod Division: **%**
- Exponents: ******

Addition

```
number1 = 6
number2 = 5
sum = number1 + number2
```

- The variable *sum* is assigned a reference to the value 11.

Mixed Type Arithmetic

- Special rules apply when performing arithmetic operations on numbers of different types. For example, adding an int and a float together. *What data type is the result of that arithmetic*?

- Different languages have different rules for mixed type arithmetic.
  - Usually implements some sort of ranking system to determine the data type of arithmetic results.

Mixed Type Arithmetic

- Python's rules are pretty straight-forward:
  - Arithmetic operations performed only on ints result in an int.
  - Arithmetic operations performed only on floats result in an float.
  - Arithmetic operations performed on a combination of ints and floats result in a float.

First Operand	Second Operand	Resulting Type
int	int	int
int, float	float	float

Mixed Type Arithmetic

```
value1 = 10
value2 = 15
result1 = value1 + value2
```

- The data type of the result1 variable will be int.

```
value3 = 11.7
value4 = 12
result2 = value3 + value4
```

- The data type of the result2 variable will be float.

Mixed Type Arithmetic

```
value5 = 13.5
value6 = 18.6
result3 = value5 + value6
```

- The data type of the result3 variable will be float.

```
value7 = 21
value8 = 19
value9 = 2.3
result4 = value7 + value8 + value9
```

- The data type of the result4 variable will be float.

Subtraction

```
number1 = 6
number2 = 5
difference = number1 - number2
```

- The variable *difference* is assigned a reference to the value -1.

Multiplication

```
number1 = 6
number2 = 5
product = number1 * number2
```

- The variable *product* is assigned a reference to the value 30.

Float Division

- The float division operator always returns a float result.

```
number1 = 8
number2 = 2
quotient = number1 / number2
```

- The variable *quotient* is assigned a reference to the value 4.0

Float Division (Another Example)

```
number1 = 5
number2 = 2
quotient = number1 / number2
```

- The variable *quotient* is assigned a reference to the value 2.5

Integer Division

- The integer division operator returns a quotient with any fractional portion truncated/dropped.
  - Value returned depends on the data types of the operands.

```
number1 = 8
number2 = 2
quotient = number1 // number2
```

- The variable *quotient* is assigned a reference to the value 4

Integer Division (Another Example)

```
number1 = 5
number2 = 2
quotient = number1 // number2
```

- The variable *quotient* is assigned a reference to the value 2

Integer Division (Another Example)

```
number1 = 10.5
number2 = 2
quotient = number1 // number2
```

- The variable *quotient* is assigned a reference to the value 5.0
  - (The result is a float because of the mixed type arithmetic rules.)

Integer Division (Another Example)

- Negative results are rounded away from zero.

```
number1 = -5
number2 = 2
quotient = number1 // number2
```

- The variable *quotient* is assigned a reference to the value -3
  - -5 / 2 = -2.5

Mod Division

- Finds the remainder of a division.

```
number1 = 11
number2 = 4
remainder = number1 % number2
```

- The variable *remainder* is assigned a reference to the value 3.
- "11 divided by 4 is 2 <u>with a remainder of 3</u>"

Exponents

```
number1 = 2
number2 = 3
result = number1 ** number2
```

- The variable *result* is assigned a reference to the value 8.

Operator Precedence

- PE[MD%][AS] (left to right)
- Multiplication, Integer or Float Division, Mod Division – same priority
- Addition, Subtraction – same priority

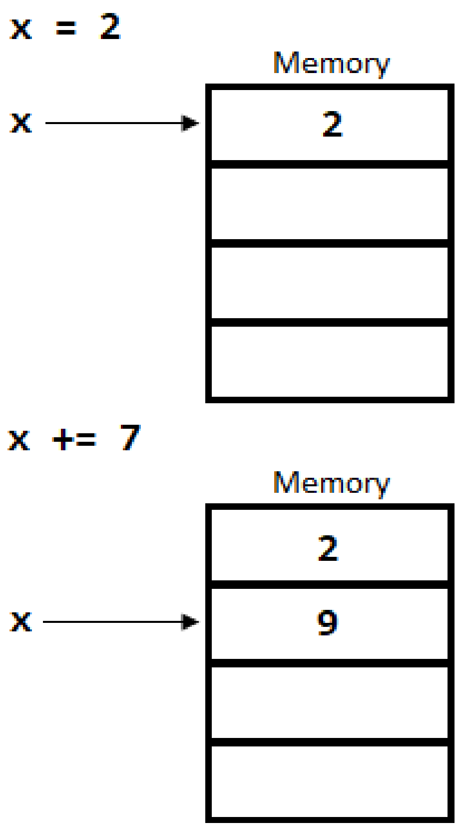
```
num1 = 13
num2 = 5
num3 = 3
num4 = 2

answer = (num1 % num2 * num2) / num3 - num3 ** num4
```

Augmented Assignment

- **Augmented assignment operators** (sometimes called “compound assignment operators”) perform both an arithmetic and assignment operation.

```
x  =  2
x  +=  7
```



Augmented Assignment Operators

```
my_number = 11
my_number += 4
my_number -= 5
my_number *= 2
my_number /= 4
my_number //= 3
my_number %= 2
my_number **= 3
```

Equivalent to:

```
my_number = my_number + 4
my_number = my_number - 5
my_number = my_number * 2
my_number = my_number / 4
my_number = my_number // 3
my_number = my_number % 2
my_number = my_number ** 3
```

Rounding Functions

- A programming language typically provides ways to round floats to an integer/whole number value.

- Sometimes, these functions are available by default.

- Other times, the functions must be imported into your program.

Types of Rounding Functions

- A ***round function*** will round a float, up or down, to the nearest whole number.
  - .5 or higher -> Rounded up
  - Lower than .5 -> Rounded down
    - 20.5 -> 21.0
    - 20.4 -> 20.0
- A ***floor function*** will round a float down.
  - Fractional is irrelevant.
  - 45.9 -> 45.0
- A ***ceiling function*** will round a float up.
  - Fractional is irrelevant.
  - 32.1 -> 33

Rounding Floats

- Python's built-in round function will round a float to the nearest whole number.
  - The function's return value is a float type.

```
original_number = 25.6
rounded_number = round(original_number)
print(rounded_number)
```

```
26.0
```

```
original_number = 25.4
rounded_number = round(original_number)
print(rounded_number)
```

```
25.0
```

Python Modules

- A Python module is a file that contains Python code, specifically functions.
- Python comes with many modules, but their functions are not readily available to call upon.
  - Unlike the print or round functions which are always available.
- Modules can be imported into our own programs.
  - Allows us to use the functions contained within them.

Math module

- The math module provides mathematical functions beyond what is provided by default (like addition and subtraction).
- Common uses are:
  - Rounding values up or down.
  - Square roots
  - Trig functions

- Import the math module using the following statement:

```python
import math
```

When/Where to Import Modules

- A Python module’s import statement can appear anywhere in your source code.
- However, the module must be imported before you try to use any of its functionality.
- Most programmers opt to put any and all import statements at the beginning of their source code.

Math module – Rounding Up

- The math module's *ceil* function rounds a float up.

```
import math
original_number = 15.1
rounded_number = math.ceil(original_number)
print(rounded_number)
```

```
16.0
```

Math module – Rounding Down

- The math module's *floor* function rounds a float down.

```
import math
original_number = 15.9
rounded_number = math.floor(original_number)
print(rounded_number)
```

```
15.0
```

Math module – Square Roots

- The math module's square root (sqrt) function returns the square root of a number.

```python
import math
original_number = 16
square_root = math.sqrt(original_number)
print(square_root)
```

```
4.0
```

String Concatenation

- ***Concatenation*** is the process of joining data together into one string using the addition operator.
  - This is not the same as *appending*. When you concatenate Strings together, the references of the variables are not changed.

```
hello = "Hello "
world = "World!"
hello_world = hello + world
print(hello_world)
```

```
Hello World!
```

Note: The values of the string variables hello and world **do not change**.

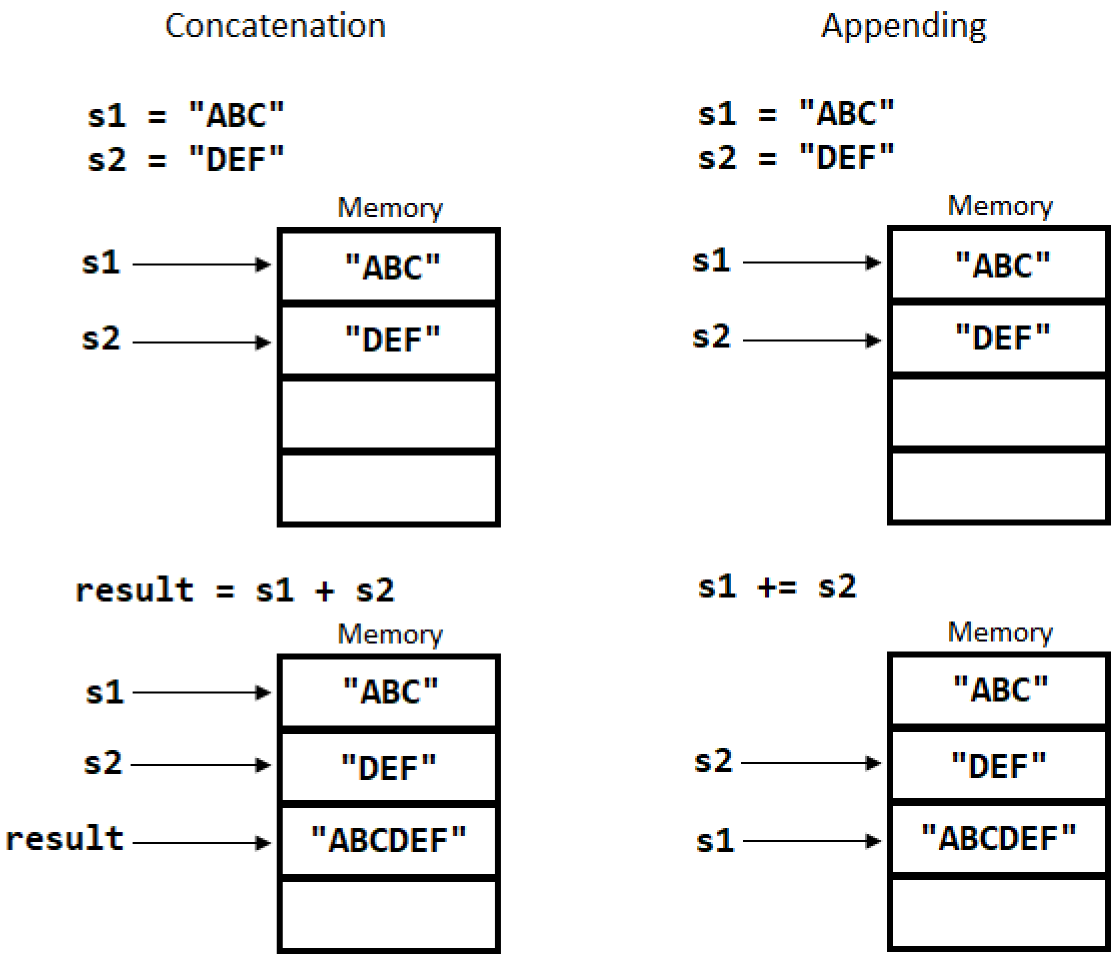
Appending to Strings

- ***Appending*** is the process of joining data together into one string that replaces or updates the original.
  - Unlike concatenation, appending changes the reference of a variable.
  - To append to a string, use the addition combined assignment operator.

```
hello = "Hello "
world = "World!"
hello += world
print(hello)
```

```
Hello World!
```

Concatenation vs Appending



Type Conversion

- Type conversion is a process that changes the type of some stored data.
  - For example, converting a string to an int and vice-versa.

- **Type Coercion** – An implicit conversion of one type to another.

- **Type Casting** – An explicit conversion of one type to another.

Type Coercion

- Mixed type arithmetic is a form of type coercion.
- Both operands must be the same type.

```
value1 = 13.5
value2 = 18
result = value1 + value2
```

- The value2 variable is implicitly converted to a float.

Note: The value2 variable is still an int.

Type Casting Strings to Numbers

```
ten = "10"
result = ten + 15
```

- The above code will not work. You cannot perform arithmetic with strings, even if the string’s characters are numbers.
- Numeric strings must be converted to int or float form before you can use them as a numeric type.

Type Casting Strings to ints

- We can use Python's built-in int function to get the numeric value of a string as an int.

```
ten = "10"
result = int(ten) + 15
print("The result is", result)
```

```
The result is 25
```

Note: The ten variable is still a string.

Type Casting Strings to floats

- We can use Python's built-in float function to get the numeric value of a string as an float.

```python
ten = "10.57"
result = float(ten) + 15
print("The result is", result)
```

```
The result is 25.57
```

Note: The ten variable is still a string.

Value Error

- A **Value Error** is a Python error that will often occur as the result of trying to convert a string that isn't a number into a number.

```
letters = "abcd"
to_number = int(letters)
```

```
>>> letters = "abcd"
>>> toNumber = int(letters)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'abcd'
>>>
```

Type Casting ints to floats

- Use Python's built-in float function to convert the value of an int to a float.

```python
my_number = 34653
print(my_number)
my_number = float(my_number)
print(my_number)
```

```
34653
34653.0
```

Type Casting floats to ints

- Use Python's built-in int function to convert the value of a float to an int.
  - Any fractional portion of the float value will be truncated.

```
my_number = 346.87
print(my_number)
my_number = int(my_number)
print(my_number)
```

```
346.87
346
```

Type Casting ints/floats to Strings

- In some languages, like Python, ints and floats cannot be directly concatenated with a string.

```
first_half = "There are "
days = 31
second_half = " days in January."
sentence = first_half + days + second_half
print(sentence)
```

WILL NOT WORK IN PYTHON

Type Casting ints to Strings

- ints must be converted to a string type using Python's built-in str function.
  - The str function returns the int argument in string form.

```python
first_half = "There are "
days = 31
second_half = " days in January."
sentence = first_half + str(days) + second_half
print(sentence)
```

```
There are 31 days in January.
```

Note: The days variable is still an int.

Type Casting while printing

- Be sure to convert any non-string variables when concatenating.

```
first_half = "There are "
days = 31
second_half = " days in January."
print(first_half + str(days) + second_half)
```

```
There are 31 days in January.
```

Type Casting floats to Strings

- floats must be converted to a string type using Python’s built-in str function.
  - The str function returns the float argument in string form.

```
first_half = "Today's temperature is "
temperature = 67.5
second_half = " degrees."
sentence = first_half + str(temperature) + second_half
print(sentence)
```

```
Todays temperature is 67.5 degrees.
```

Note: The temperature variable is still a float.

Type Casting with Keyboard Input

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))
print("Nice to meet you " + name + "!")
print("You are " + str(age) + " years old.")
```

```
Enter your name: John
Enter your age: 45
Nice to meet you John!
You are 45 years old.
```