CptS 111, Spring 2023
Lect. #6, Jan. 30, 2023
Class Notes

Today's Agenda:

1. A review of `input()`
2. A little more on `int()`, `float()`, and `str()`
3. String formatting using f-strings

---

1. Review of `input()` With and Without a Prompt

- The `input()` function is used to get input from a user.
- The `input()` function always returns the item that has been entered *AS A STRING.*
- The string that is returned by a user must be assigned to an lvalue, i.e., a VALUE (name) to the LEFT of the assignment operator `=`.
- The prompt inside the `input()` function is a string that is printed to stdout, i.e., your screen.
- **Don't use the `print()` function to print the input prompt because the `input()` function does this for you.**
- When a prompt is written to the screen, you're supposed to enter the input you're prompted for.
- The `input()` function can be nested with other functions such as `float()` and `int()`. *Don't nest it with the `str()` function because `input()` already returns a string.*
- In your zyBook, participation and challenge activities don't allow you to use prompts because of the way your zyBook works.
- **However, in zyLab_PAs, you're supposed to use the prompts that are given in the examples!**

In [1]:
```
# Example 1: Using input() with a prompt
# Recall that type() gives you the type of an item

name = input('Enter a name: ')
print('lvalue value:', name)
print('lvalue type:', type(name))
```

```
Enter a name: SpongeBob
lvalue value: SpongeBob
lvalue type: <class 'str'>
```

In [2]:
```
# Example 2: Using input() and float() as nested functions
# with a prompt

number = float(input('Enter a number: '))
print('lvalue value:', number)
print('lvalue type:', type(number))
```

```
Enter a number: 1.618
lvalue value: 1.618
lvalue type: <class 'float'>
```

In [3]:
```
# Example 3: Using input() and int() as nested functions
# without a prompt

integer = int(input())
print('lvalue value:', integer)
print('lvalue type:', type(integer))
```

```
42
lvalue value: 42
lvalue type: <class 'int'>
```

2. A Little More on `int()`, `float()`, and `str()`

-We can use `str()` to convert floats and integers to strings.
-We can use `float()` to convert string floats and string integers to floats.
-We can use `float()` to convert integers to floats.
-We can use `int()` to convert floats to integers, but they'll be truncated.
-We can use `int()` to convert string integers to integers.
-**We *CANNOT* use `int()` to convert string floats to integers.**

In [4]:
```
# Converting a float to an integer or string
# Note that int() truncates a float

float_num = 3.141592653
print('float_num type:', type(float_num), '\nfloat_num =', float_num)
print()

int_num = int(float_num)
print('int_num type:', type(int_num), '\nint_num =', int_num)
print()

string_float = str(float_num)
print('string_float type:', type(string_float), '\nstring_float =', string_float)
```

```
float_num type: <class 'float'>
float_num = 3.141592653

int_num type: <class 'int'>
int_num = 3

string_float type: <class 'str'>
string_float = 3.141592653
```

In [5]:
```
# Converting a string integer to a float or an integer

str_int = '42'
print('str_int type:', type(str_int), '\nstr_int =', str_int)
print()

float_num = float(str_int)
print('float_num type:', type(float_num), '\nfloat_num =', float_num)
print()

int_num = int(str_int)
print('int_num type:', type(int_num), '\nint_num =', int_num)
```

```
str_int type: <class 'str'>
str_int = 42

float_num type: <class 'float'>
float_num = 42.0

int_num type: <class 'int'>
int_num = 42
```

In [6]:
```
# Converting a string float to a float

str_float = '3.141592653'
print('str_float type:', type(str_float), '\nstr_float =', str_float)
print()

float_num = float(str_float)
print('float_num type:', type(float_num), '\nfloat_num =', float_num)
```

```
str_float type: <class 'str'>
str_float = 3.141592653

float_num type: <class 'float'>
float_num = 3.141592653
```

In [7]:
```
# Can't convert a string float to an integer

str_float = '3.141592653'
print('str_float type:', type(str_float), '\nstr_float =', str_float)
print()

int_num = int(str_float)
print('int_num type:', type(int_num), '\nint_num =', int_num)
```

```
str_float type: <class 'str'>
str_float = 3.141592653


---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [7], in <cell line: 7>()
      4 print('str_float type:', type(str_float), '\nstr_float =', str_float)
      5 print()
----> 7 int_num = int(str_float)
      8 print('int_num type:', type(int_num), '\nint_num =', int_num)

ValueError: invalid literal for int() with base 10: '3.141592653'
```

We can't use `int()` to convert a string float, but we *can* use `int()` to convert a float. Note the difference between using `int()` and `round()`. `int()` truncates a float while `round()` rounds a float up or down.

Ch. 3 (cont.)

3. String Formatting Using f-strings

There are a number of ways to format a string. We begin with the newest way which was introduced in Python 3.6, and in Ch. 8 we'll look at the old ways because so many people still use them. Importantly, note that string formatting can be used for any functions and methods with string arguments. Thus, you can use string formatting with both the `print()` and `input()` functions, for the latter in the string prompt.

A. String Formatting Template

```
f' {} '
F' {} '
f"  {} "
F"  {} "
```

where

```
'  ' = format string: combination of string literals and replacement fields
 {}  = replacement field (as many replacement fields as desired)
```

The replacement fields contain the values to print and also format specifiers. We'll learn some format specifiers today and more in the future. The easiest way to demonstrate is with some examples.

In [8]:
```
# Simple f-string examples

integer = 2 ** 200
x = 3.141592653 ** 4
print(f'The value of integer is {integer}.')
print(f'x equals {x}.')
```

```
The value of integer is 1606938044258990275541962092341162602522202993782792835301376.
x equals 97.40909096085326.
```

Again, we see that integers are exact but floats are approximated.

What's special about f-strings is that we can use operations and function inside the replacement fields even though we're "inside" a string!

In [9]:
```
# Let's do the previous examples using operations.

print(f'The value of integer is {2 ** 200}.')
print(f'x equals {3.141592653 ** 4}.')
```

```
The value of integer is 1606938044258990275541962092341162602522202993782792835301376.
x equals 97.40909096085326.
```

We see that we get the same results! Moreover, we can include the expression we're calculating together with the results by including the assignment operator (although the formatting isn't very pretty).

In [10]:
```
# Previous examples with operations and the assignment operator

print(f'The value of integer is {2 ** 200 = }.')
print(f'x equals {3.141592653 ** 4 = }.')
```

```
The value of integer is 2 ** 200 = 1606938044258990275541962092341162602522202993782792835301376.
x equals 3.141592653 ** 4 = 97.40909096085326.
```

We're not going to use the assignment operator in replacement fields in CptS 111, but it's useful to know you can do this.

B. Format Specifiers

A number of different format specifiers exist that do what the name implies, i.e., specify formats. We'll cover the most useful ones (IMO).

1. Format specifier: Field width (= number of spaces)

In [11]:
```
# len('gray') > (specified width) so ignored.

color1 = 'crimson'
color2 = 'gray'
f'{color1:10} and {color2:3}' # field widths of 10 and 3
```

Out[11]:
```
'crimson    and gray'
```

- Numbers after colons are width specifiers, i.e., give total number of spaces.
- If spaces greater than specified width, it's ignored, e.g., `len('gray')` = 4 > 3

- If spaces less than specified width, spaces added, e.g., `len('crimson') + 3 = 10`

2. Format specifier: Alignment

In [12]:
```
# 'crimson' left justified; 'gray' right justified

f"{'crimson':<10} and {'gray':>10}"
```

Out[12]:
```
'crimson    and       gray'
```

```
<  left
^  center
>  right
=  left justify sign and right justify number
```

In [13]:
```
# Left justify sign and right justify number

f'{-3.14:=8}'
```

Out[13]:
```
'-   3.14'
```

In [14]:
```
# Everything right justified

f'{-3.14:>8}'
```

Out[14]:
```
'   -3.14'
```

In [ ]:
```
# Everything left justified

f'{-3.14:<8}'
```

In [15]:
```
# Everything center justified

f'{-3.14:^8}'
```

Out[15]:
```
' -3.14  '
```

3. Format specifier: Fill and padding (use between colon and alignment chr)

In [16]:
```
# Center justify and add *'s to fill'

f'{"cat":*^9}'
```

Out[16]:
```
'***cat***'
```

In [17]:
```
# Zero padding is useful for time (as in zyLab_PA2!)

hrs = 1
mins = 9
secs = 13
f'{hrs}:{mins:0>2d}:{secs:0>2d}'
```

Out[17]:
```
'1:09:13'
```

4. Format specifier: Maximum width and precision

In [18]:
```
# Max width of 10 and 5 decimal places

x = 1 / 7
f'{x:10.5}'
```

Out[18]:
```
'   0.14286'
```

The 10 gives the number of spaces, and the 5 gives the number of decimal places.

5. Format specifier: Type and precision

In [19]:
```
f'{65:b}, {65:c}, {65:d}, {65:f}, {65:e}, {65:g}'
```

Out[19]:
```
'1000001, A, 65, 65.000000, 6.500000e+01, 65'
```

where 65 is expressed using the following representations:

```
b binary
c character
d integer (decimal integer vs binary, octal, hexadecimal)
f float
e exponential
g general (Python's idea of the nicest way to display a value)
```

In [20]:
```
# Note the number of values used after the decimal place; values are rounded
# at the DEFAULT number of digits for the type

num = 65.12345678
f'{num:f}, {num:e}, {num:g}'
```

Out[20]: '65.123457, 6.512346e+01, 65.1235'

Interpretation of precision depends on the type. For `e` and `f`, it means the number of digits to the right of the decimal point. For `g`, it means the total number of digits. Notice how all of these round the values rather than truncating.

In [21]:
```
# Note the difference in the interpretation of precision based on type

f'{num:.2f}, {num:.2e}, {num:.2g}'
```

Out[21]: '65.12, 6.51e+01, 65'

In the example above, we want to change the default precision to two decimal places, but precision is interpreted differently for the different format specifiers.