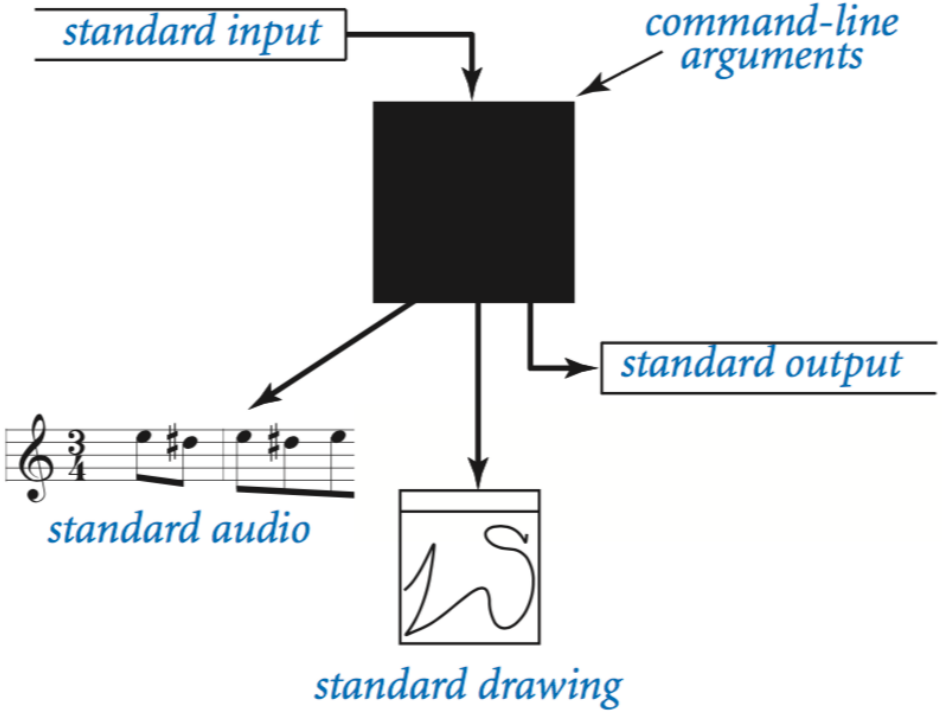
BASIC INPUT/OUTPUT



Outline: Basic Input/Output

- Screen Output
- Keyboard Input
- Command Line Input
- File Input

Simple Screen Output

```
print("The count is " + str(count))
```

- Outputs the sting literal `"The count is "`
- Followed by the current value of the variable `count`, converted to a string.

- We've seen several examples of screen output already.
  - `print()` is a function in python
    - The “stuff” inside the parenthesis are arguments to that function

Screen Output

- The line continuation operator (\) is useful when everything does not fit on one line.

```
print("Lucky number = " + str(13) + \
  " Secret number = " + str(42))
```

- You don’t want to break the line in the middle of s string though.

Screen Output

- To get text to print out on the same line as the previous print, use:

```
print("One, two, ", end="")
print(" buckle my shoe. ", end="")
print(" Three, four,")
print(" shut the door.")
```

Pretty Text Formatting

- printf-style formatting
  - Common way to nicely format output
  - Present in many programming languages
    - Java, C++, Perl, PHP, ...
  - Use a special format language:
    - Format string with special codes
    - One or more variables get filled in
- In Python:

```
print("text %code" %(value))
```

Formatted Printing

```
# print integer and float value
print("Enrollment: %2d, Average Score: %5.2f" %(52, 78.523))

# print integer and float variables
enroll = 52
score = 78.523
print("Enrollment: %2d, Average Score: %5.2f" %(enroll, score))

# print two integer value
print("Total students: %3d, Monday Class: %2d" %(52, 26))

# print exponential value
print("%10.3E"% (356.08977))
```

Floating-Point Formatting

```
import math

f = 0.123456789

# %f code is used with floating point variables
# %f defaults to rounding to 6 decimal places
# \n prints a newline character
print("f is about %f\n" %(f))

# Number of decimal places specified by .X
# Output is rounded to that number of places
print("PI is about %.1f\n" %(math.pi))
print("PI is about %.2f\n" %(math.pi))
print("PI is about %.3f\n" %(math.pi))
print("PI is about %.4f\n" %(math.pi))

# %e code outputs in scientific notation
# .X specifies number of significant figures
C = 299792458.0
print("C = %e\n" %(C))
print("C = %.3e\n" %(C))
```

```
f is about 0.123457
```

\n means line feed

```
PI is about 3.1

PI is about 3.14

PI is about 3.142

PI is about 3.1416
```

```
C = 2.997925e+08

C = 2.998e+08
```

Integer Formatting

```
# %d code is for integer values
# Multiple % codes can be used in a single print()
power = 1
for i in range(0, 8):
    print("%d = 2^%d" %(power, i))
    power = power * 2

# A r                                    um width
# Goo                                    values
power
for i in range(0, 8):
    print("%5d = 2^%d" %(power, i))
    power = power * 2
```

You can have multiple % codes that are filled in by a list of parameters to print()

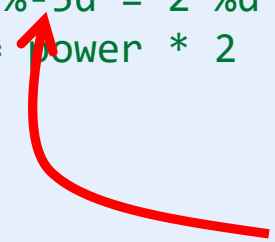
Minimum width of this field in the output. Python will pad with whitespace to reach this width (but can exceed this width if necessary).

```
1 = 2^0
2 = 2^1
4 = 2^2
8 = 2^3
16 = 2^4
32 = 2^5
64 = 2^6
128 = 2^7
```

```
    1 = 2^0
    2 = 2^1
    4 = 2^2
    8 = 2^3
   16 = 2^4
   32 = 2^5
   64 = 2^6
  128 = 2^7
```

Flags

```
# Same table, but left justify the first field
power = 1
for i in range(0, 8):
    print("%-5d = 2^%d" %(power, i))
    power = power * 2
```



- flag causes this field to be left justified

```
1     = 2^0
2     = 2^1
4     = 2^2
8     = 2^3
16    = 2^4
32    = 2^5
64    = 2^6
128   = 2^7
```

Text Formatting

```
# Characters can be output with %c, strings using %s
name = "Bill"
grade = 'B'
print("%s got a %c in the class." %(name, grade))
```

```
Bill got a B in the class.
```

```
# This prints the same thing without using printf
print(name + " got a " + grade + " in the class.")
```

An equivalent way to print the same thing out using good old concatenation.

```
# And so does this
str = name + " got a " + grade + " in the class."
print(str)
```

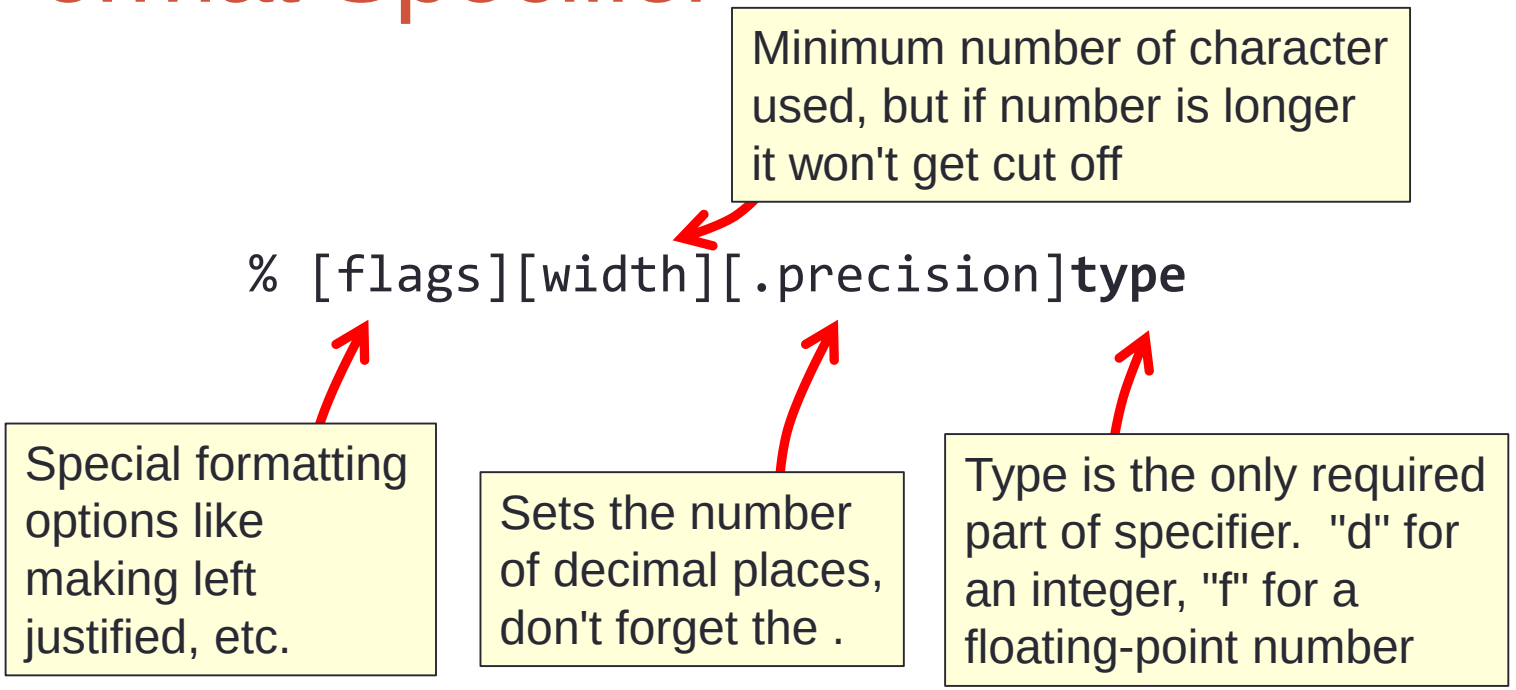
Creating Formatted Strings

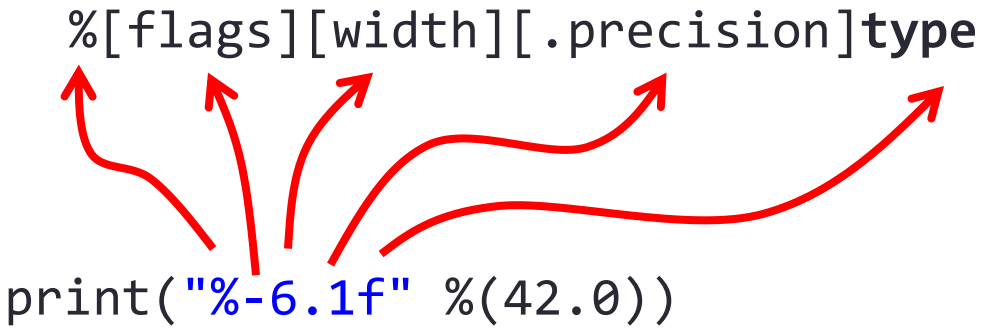
- Formatted `String` creation
  - You don't always want to immediately print formatted text to standard output
  - Save in a string variable for later use

```
# Formatted Strings can be created and added to
lines = ""
for i in range(0, 4):
   lines += "Random number %d = %.2f\n" %(i, random.random())
print(lines)
```

```
Random number 0 = 0.54
Random number 1 = 0.50
Random number 2 = 0.39
Random number 3 = 0.64
```

The Format Specifier





print Gone Bad

- Format string specifies:
  - Number of variables to fill in
  - Type of those variables
- With great power comes great responsibility
  - Format must agree with #/types of arguments
  - Runtime error otherwise

```
# Runtime error %f expects a number
print("crash %f\n" %("Hello"))

# Runtime error, %d expects a number
#print("crash %d\n" %("Hello"))

# Runtime error, not enough arguments
#print("crash %d %d\n" %(2))
```

print Puzzler

Code	Letter
`print("%f"  %(4242.00))`	E
`print("%d"  %(4242))`	A
`print("%.2f" %(4242.0))`	B
`print("%.3e" %(float(4242)))`	C
`print("%-d"  %(4242))`	A

Letter	Output
A	4242
B	4242.00
C	4.242e+03
D	4,242
E	4242.000000

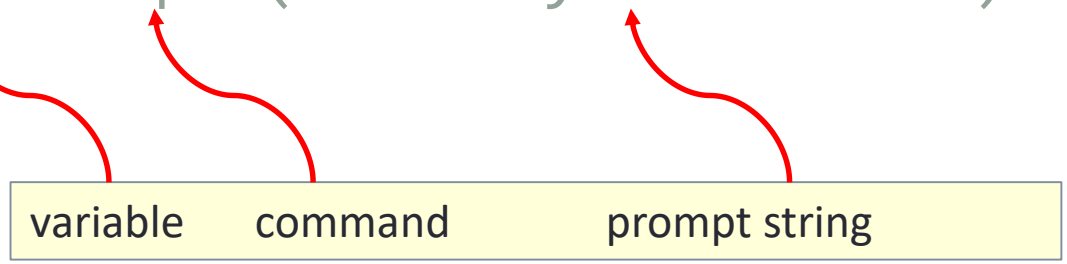
Code	#
`print("%d%d" %(42, 42))`	2
`print("%d+%d" %(42, 42))`	1
`print("%d %d" %(42))`	5
`print("%8d" %(42))`	3
`print("%-8d" %(42))`	4
`print("%d"  %(42.0))`	4

#	Output
1	42+42
2	4242
3	42
4	42
5	runtime error
6	4242.00

Interactive Keyboard Input

- Python has reasonable facilities for handling keyboard input.
  - Use the input() command
  - If you don't save to a variable, the input gets lost

```
name = input("Enter your name: ")
```



  - Whatever the user types before pressing <Enter> gets stored in the variable called `name`

Interactive Keyboard Input

- Input comes in as a string

```
number = input("Enter your favorite number: ")
```

  - If you want to use the incoming value as numeric, you must convert it

```
number = int(input("Enter your favorite number: "))
```

Or:

```
number = float(input("Enter your favorite number: "))
```

Command Line Input

- Input comes from the command line when you run the program
  - Run…customized in the Idle shell
  - From the Command window
- Input data comes into the list of strings, `sys.argv`
- If you don't save the data to variable(s), the data is lost
- You must `import sys` before you can access the list

```
import sys

program = sys.argv[0]
number = sys.argv[1]
```

Command Line Input

- Input comes in as a list of strings

```
import sys

program = sys.argv[0]
number = sys.argv[1]
```

  - If you want to use the incoming value as numeric, you must convert it

```
number = int(sys.argv[1])
```

Or:

```
number = float(sys.argv[1])
```

Input from Files

- What if..
  - There are too many values for a user to type interactively?
  - These values are stored in a text file?
- Can our program read these values from a file?
  - Yep! ☺

Python File Input

- Step 1: We need to open the file:

```
with open(fname, 'r') as f:
```

  - fname is a string for the file name
  - f is just any variable that you want to use
  - ‘r’ means we want to read the file (as opposed to writing it)

```
f = open(fname, 'r')
```

  - The first approach will help you walk through the whole file
  - The second approach just opens it and lets you figure things out
- Step 2: We need to read in data from the file (and save the data somehow)
- Step 3: Once we are done with the file, we need to close it:

```
f.close()
```

File Input

```
import sys

sum   = 0.0
count = 0

# Check if we need to print out command line help
if len(sys.argv) < 2:
    print("AvgNumsFile <filename>")
else:
    # Open up the text file for reading
    fname = sys.argv[1]
    with open(fname, 'r') as f:
        # Keep going as long as there is more text in the file
        for line in f:
            # Translate that line to a float
            sum += float(line)
            count += 1
    f.close()

    # Print out the final average
    print(sum / count)
```

As an example, here we are reading in a file containing many numbers and finding their average

Run this with squares.txt as the command line argument.

What's in squares.txt?

0
1
4
9
16
25
36
49
64
81
100
121
144
169 … 1000 entries of squared numbers

File Input

```
import sys

sum   = 0.0
count = 0

# Check if we need to print out command line help
if len(sys.argv) < 2:
    print("AvgNumsFile <filename>")
else:
    # Open up the text file for reading
    fname = sys.argv[1]
    f = open(fname, 'r')
    line = f.readline().strip()
    # Keep going as long as there is more text in the file
    while line != "":
        # Translate that line to a float
        sum += float(line)
        count += 1
        # Read the next line
        line = f.readline().strip()
    f.close()

    # Print out the final average
    print(sum / count)
```

Same functionality as last program but opening the file in a different way.

Run this with squares.txt as the command line argument.

Why the .strip() after f.readline()?

Whitespace characters are also in the file – newline characters that make each number be on a new line in the file. We need to get rid of these.

Let's Try Another Type of File

- Configuration files are often used to initialize software programs.
  - Might hold preferences for how you want your text editor to look
  - Maybe levels in a game
- These are all text files, so we can open them up and read them

Configuration File: hitchhiker.txt

```
stars.jpg
dont_panic_40.png 0.5 0.5 0.035 100
6
asteroid_small.png 0.1 0.1 0.018 -0.002 -0.003
asteroid_medium.png 0.2 0.2 0.030 0.002 -0.003
asteroid_large.png 0.3 0.3 0.065 -0.002 0.003
asteroid_small.png 0.4 0.4 0.018 -0.001 -0.004
asteroid_medium.png 0.6 0.6 0.030 0.002 -0.003
asteroid_large.png 0.7 0.7 0.065 -0.0035 0.0025
```

```
# Hitchhikers Guide to the Galaxy: Avoid a bunch of asteroids
#  <background image>
#  <player image> <player x-position> <player y-position> <player radius> <player speed factor>
#  <number enemies>
#  <enemy0 image> <enemy0 x-position> <enemy0 y-position> <enemy0 x-velocity> <enemy0 y-velocity>
#  <enemy1 image> <enemy1 x-position> <enemy1 y-position> <enemy1 x-velocity> <enemy1 y-velocity>
#  ...
```

Code to Read Configuration File

- To open and read **just** the first two lines of the file, we might use:

```
# Open up the text file for reading
fname = sys.argv[1]
with open(fname, 'r') as f:
    # Read in the first line of text
    line = f.readline().split()
    # Translate that line to a string
    bg = picture.Picture(line[0])
    line = f.readline().split()
    playerImg = line[0]
    player = picture.Picture(playerImg)
    playerX = float(line[1])
    playerY = float(line[2])
    playerRadius = float(line[3])
    playerSpeed = int(line[4])
f.close()
```

Why the .split() after f.readline()?

Many lines in the file contain more than one value. We can use split to build a list of strings (remember lists?) from each line and then pull items off the list to use in our program. Since .split() splits on whitespace, it also gets rid of those pesky newline characters so we don't need to use .strip().

Summary: Basic Input/Output

- Screen Output
- Keyboard Input
- Command Line Input
- File Input