Python Programming Essentials

Unit 2: Structured Python

CHAPTER 5: SEQUENCES: STRINGS, LISTS, AND FILES

DR. ERIC CHOU IEEE SENIOR MEMBER

Objectives

LECTURE 1

Objectives

- To understand the string data type and how strings are represented in the computer.
- To become familiar with various operations that can be performed on strings through built-in functions and string methods.

Objectives

- To understand the basic idea of sequences and indexing as they apply to Python strings and lists.
- To be able to apply string formatting to produce attractive, informative program output.
- To understand basic file processing concepts and techniques for reading and writing text files in Python.

Objectives

- To understand basic concepts of cryptography.
- To be able to understand and write programs that process textual information.

The String Data Type

LECTURE 2

The String Data Type

- The most common use of personal computers is word processing.
- Text is represented in programs by the string data type.
- A string is a sequence of characters enclosed within quotation marks (") or apostrophes (').

The String Data Type

```
>>> str1="Hello"
>>> str2='spam'
>>> print(str1, str2)
Hello spam
>>> type(str1)
<class 'str'>
>>> type(str2)
<class 'str'>
```

The String Data Type

- Getting a string as input

```
>>> firstName = input("Please enter your name: ")
Please enter your name: John
>>> print("Hello", firstName)
Hello John
```

- Notice that the input is not evaluated. We want to store the typed characters, not to evaluate them as a Python expression.

The String Data Type

- We can access the individual characters in a string through indexing.
- The positions in a string are numbered from the left, starting with 0.
- The general form is <string>[<expr>], where the value of expr determines which character is selected from the string.

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet = "Hello Bob"
>>> greet[0]
'H'
>>> print(greet[0], greet[2], greet[4])
H l o
>>> x = 8
>>> print(greet[x - 2])
B
```

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

- In a string of n characters, the last character is at position n-1 since we start counting with 0.
- We can index from the right side using negative indexes.

```
>>> greet[-1]
'b'
>>> greet[-3]
'B'
```

The String Data Type

- Indexing returns a string containing a single character from a larger string.
- We can also access a contiguous sequence of characters, called a **substring**, through a process called **slicing**.

The String Data Type

Slicing:

- `<string>[<start>:<end>]`
- `start` and `end` should both be ints
- The slice contains the substring beginning at position `start` and runs up to but **doesn't include** the position `end`.

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

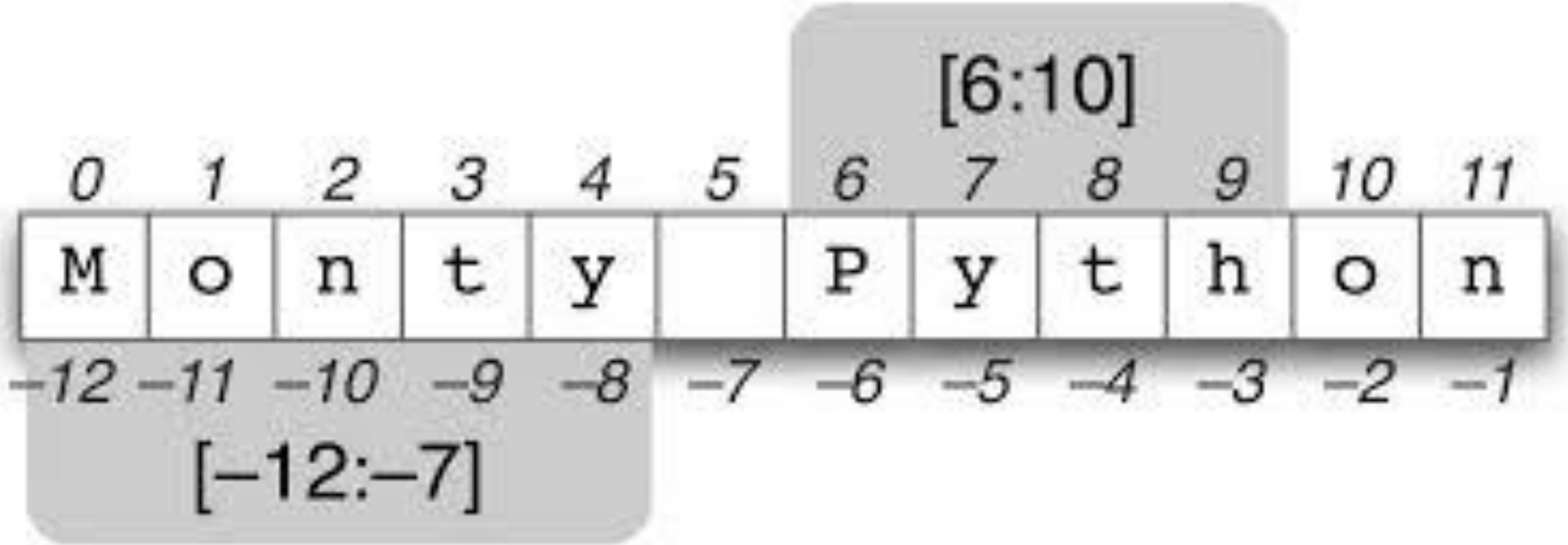
```
>>> greet[0:3]
'Hel'
>>> greet[5:9]
' Bob'
>>> greet[:5]
'Hello'
>>> greet[5:]
' Bob'
>>> greet[:]
'Hello Bob'
```

The String Data Type

- If either expression is missing, then the start or the end of the string are used.
- Can we put two strings together into a longer string?
- Concatenation “glues” two strings together (+)
- Repetition builds up a string by multiple concatenations of a string with itself (*)

The String Data Type

- Text in a program is represented by the string data type.
- String can also be viewed as an linear (1-D) array of characters.
- String can also be viewed as a sequence of characters.



The String Data Type

- The function len will return the length of a string.

```
>>> "spam" + "eggs"
'spameggs'
>>> "Spam" + "And" + "Eggs"
'SpamAndEggs'
>>> 3 * "spam"
'spamspamspam'
>>> "spam" * 5
'spamspamspamspamspam'
>>> (3 * "spam") + ("eggs" * 5)
'spamspamspameggseggseggseggseggs'
```

The String Data Type

```
>>> len("spam")
4
>>> for ch in "Spam!":
           print (ch, end=" ")

S p a m !
```

The String Data Type

Operator	Meaning
+	Concatenation
*	Repetition
<string>[]	Indexing
<string>[:]	Slicing
len(<string>)	Length
for <var> in <string>	Iteration through characters

Simple String Processing

LECTURE 3

Simple String Processing

- Usernames on a computer system
  - First initial, first seven characters of last name

```
# get user’s first and last names
first = input("Please enter your first name (all lowercase): ")
last = input("Please enter your last name (all lowercase): ")

# concatenate first initial with 7 chars of last name
uname = first[0] + last[:7]
```

Simple String Processing

```
>>>
Please enter your first name (all lowercase): john
Please enter your last name (all lowercase): doe
uname =  jdoe

>>>
Please enter your first name (all lowercase): donna
Please enter your last name (all lowercase): rostenkowski
uname =  drostenk
```

Case Study 1: username.py

```
def main():
    print(“This program generates computer usernames.\n”)
    first = input(“Please enter your first name (all lowercase):”)
    last = input(“Please enter your last name (all lowercase):”)
    uname = first[0] + last[:7]
    print(“Your username is: ”, uname)


Main()
```

Simple String Processing

- Another use – converting an int that stands for the month into the three letter abbreviation for that month.
- Store all the names in one big string:

  ```
  "JanFebMarAprMayJunJulAugSepOctNovDec"
  ```
- Use the month number as an index for slicing this string:

  ```
  monthAbbrev = months[pos:pos+3]
  ```

Simple String Processing

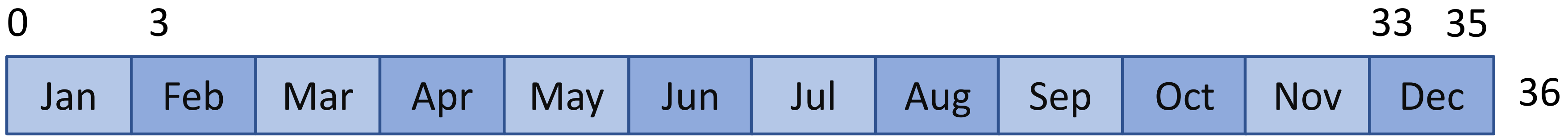
Month	Number	Position
Jan	1	0
Feb	2	3
Mar	3	6
Apr	4	9

- To get the correct position, subtract one from the month number and multiply by three

Simple String Processing

```
# month.py
#  A program to print the abbreviation of a month, given its number

def main():

    # months is used as a lookup table
    months = "JanFebMarAprMayJunJulAugSepOctNovDec"

    n = int(input("Enter a month number (1-12): "))

    # compute starting position of month n in months
    pos = (n-1) * 3
    # Grab the appropriate slice from months
    monthAbbrev = months[pos:pos+3]

    # print the result
    print ("The month abbreviation is", monthAbbrev + ".")
```

Case study 2: month.py



```
# month.py
def main():
      months = “JanFebMarAprMayJunJulAugSepOctNovDec”
      n = eval(input(“Enter a month number (1-12): ”))
      pos = (n-1) * 3
      monthAbbrev = months[pos:pos+3]
      print(“The month abbreviation is ”, monthAbbrev + “.”)

main()
```

Simple String Processing

```
>>> main()
Enter a month number (1-12): 1
The month abbreviation is Jan.
>>> main()
Enter a month number (1-12): 12
The month abbreviation is Dec.
```

- One weakness – this method only works where the potential outputs all have the same length.
- How could you handle spelling out the months?

List of Sequences

LECTURE 4

Lists as Sequences

- It turns out that strings are really a special kind of sequence, so these operations also apply to sequences!

```
>>> [1,2] + [3,4]
[1, 2, 3, 4]
>>> [1,2]*3
[1, 2, 1, 2, 1, 2]
>>> grades = ['A', 'B', 'C', 'D', 'F']
>>> grades[0]
'A'
>>> grades[2:4]
['C', 'D']
>>> len(grades)
5
```

Lists as Sequences

- Strings are always sequences of characters, but lists can be sequences of arbitrary values.
- Lists can have numbers, strings, or both!

```
myList = [1, "Spam ", 4, "U"]
```

Lists as Sequences

- We can use the idea of a list to make our previous month program even simpler!
- We change the lookup table for months to a list:

```
months = ["Jan", "Feb", "Mar", "Apr", "May",
"Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]
```

Lists as Sequences

- To get the months out of the sequence, do this:

```
monthAbbrev = months[n-1]
```

- Rather than this:

```
monthAbbrev = months[pos:pos+3]
```

Lists as Sequences

```
# month2.py
#  A program to print the month name, given it's number.
#  This version uses a list as a lookup table.

def main():

    # months is a list used as a lookup table
    months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
              "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

    n = int(input("Enter a month number (1-12): "))

    print ("The month abbreviation is", months[n-1] + ".")
```

- Note that the months line overlaps a line. Python knows that the expression isn’t complete until the closing ‘]’ is encountered.

Lists as Sequences

```
# month2.py
#  A program to print the month name, given it's number.
#  This version uses a list as a lookup table.

def main():

    # months is a list used as a lookup table
    months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
              "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

    n = int(input("Enter a month number (1-12): "))

    print ("The month abbreviation is", months[n-1] + ".")
```

- Since the list is indexed starting from 0, the n-1 calculation is straight-forward enough to put in the print statement without needing a separate step.

Lists as Sequences

- This version of the program is easy to extend to print out the whole month name rather than an abbreviation!

```
months = ["January", "February", "March",
            "April", "May", "June", "July",
            "August", "September", "October",
            "November", "December"]
```

Lists as Sequences

- Lists are **mutable**, meaning they can be changed. Strings can **not** be changed.

```
>>> myList = [34, 26, 15, 10]
>>> myList[2]
15
>>> myList[2] = 0
>>> myList
[34, 26, 0, 10]
>>> myString = "Hello World"
>>> myString[2]
'l'
>>> myString[2] = "p"

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    myString[2] = "p"
TypeError: object doesn't support item assignment
```

String Representation

LECTURE 5

String Representation

- Inside the computer, strings are represented as sequences of 1’s and 0’s, just like numbers.
- A string is stored as a sequence of binary numbers, one number per character.
- It doesn’t matter what value is assigned as long as it’s done consistently.

String Representation

- In the early days of computers, each manufacturer used their own encoding of numbers for characters.
- ASCII system (American Standard Code for Information Interchange) uses 127 bit codes
- Python supports Unicode (100,000+ characters)

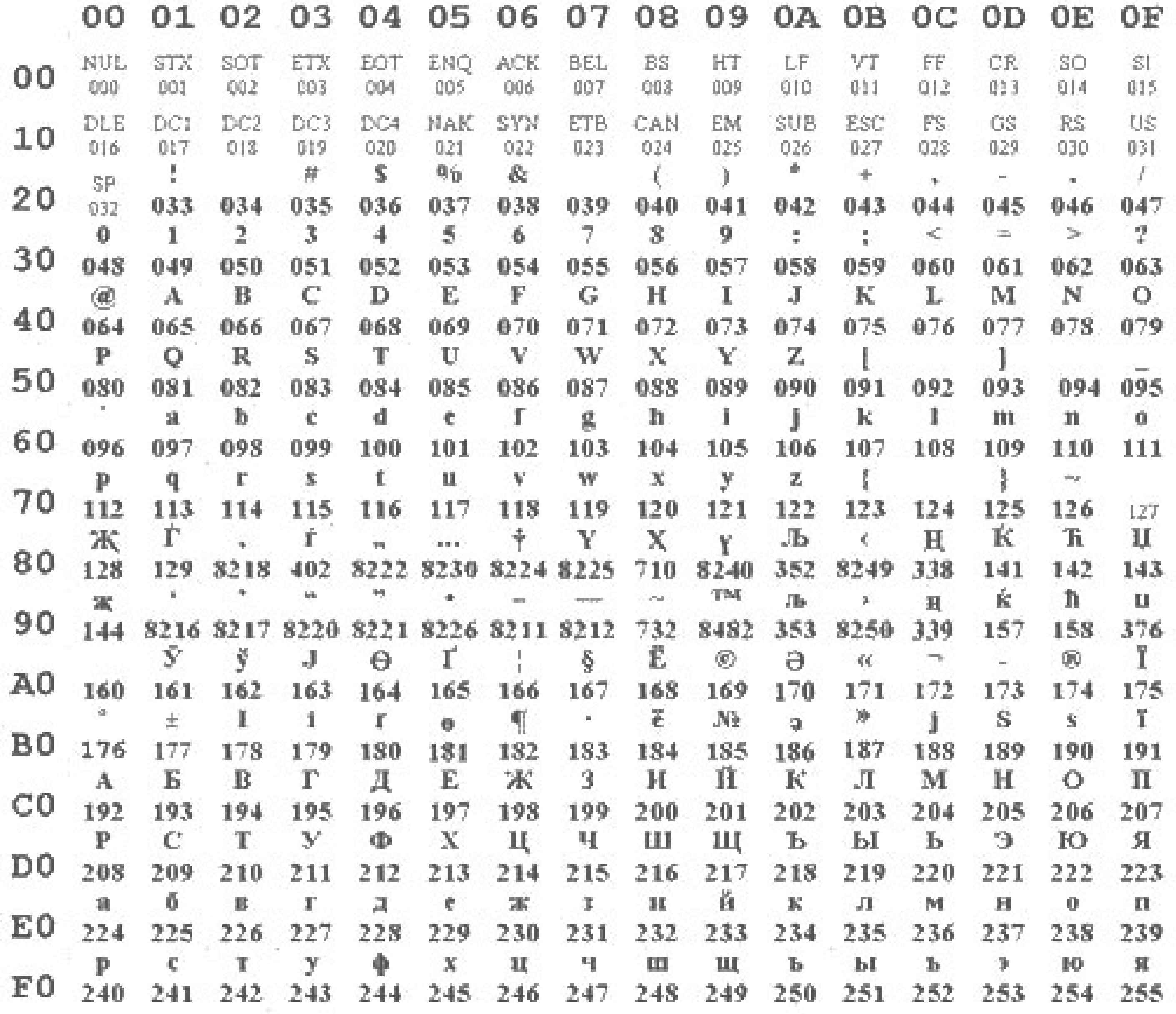
String Representation

- The **ord** function returns the numeric (ordinal) code of a single character.
- The **chr** function converts a numeric code to the corresponding character.

```
>>> ord("A")
65
>>> ord("a")
97
>>> chr(97)
'a'
>>> chr(65)
'A'
```

Unicode

ASCII



	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	NUL 000	STX 001	SOT 002	ETX 003	EOT 004	ENQ 005	ACK 006	BEL 007	BS 008	HT 009	LF 010	VT 011	FF 012	CR 013	SO 014	SI 015
10	DLE 016	DC1 017	DC2 018	DC3 019	DC4 020	NAK 021	SYN 022	ETB 023	CAN 024	EM 025	SUB 026	ESC 027	FS 028	GS 029	RS 030	US 031
20	SP 032	! 033	034	# 035	$ 036	% 037	& 038	039	(040	) 041	* 042	+ 043	, 044	- 045	. 046	/ 047
30	0 048	1 049	2 050	3 051	4 052	5 053	6 054	7 055	8 056	9 057	: 058	; 059	< 060	= 061	> 062	? 063
40	@ 064	A 065	B 066	C 067	D 068	E 069	F 070	G 071	H 072	I 073	J 074	K 075	L 076	M 077	N 078	O 079
50	P 080	Q 081	R 082	S 083	T 084	U 085	V 086	W 087	X 088	Y 089	Z 090	[091	092	] 093	094	_ 095
60	` 096	a 097	b 098	c 099	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
70	p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119	x 120	y 121	z 122	{ 123	124	} 125	~ 126	127
80	Җ 128	Ғ 129	‚ 8218	ƒ 402	„ 8222	… 8230	† 8224	Ү 8225	Х 710	ү 8240	Љ 352	‹ 8249	Ң 338	Қ 141	Һ 142	Ц 143
90	җ 144	‘ 8216	’ 8217	“ 8220	” 8221	• 8226	– 8211	— 8212	˜ 732	™ 8482	љ 353	› 8250	ң 339	қ 157	һ 158	ц 376
A0	160	Ў 161	ў 162	Ј 163	Ө 164	Ґ 165	¦ 166	§ 167	Ё 168	© 169	Ә 170	« 171	¬ 172	- 173	® 174	Ї 175
B0	° 176	± 177	І 178	і 179	ґ 180	ө 181	¶ 182	· 183	ё 184	№ 185	ә 186	» 187	ј 188	Ѕ 189	ѕ 190	ї 191
C0	А 192	Б 193	В 194	Г 195	Д 196	Е 197	Ж 198	З 199	И 200	Й 201	К 202	Л 203	М 204	Н 205	О 206	П 207
D0	Р 208	С 209	Т 210	У 211	Ф 212	Х 213	Ц 214	Ч 215	Ш 216	Щ 217	Ъ 218	Ы 219	Ь 220	Э 221	Ю 222	Я 223
E0	а 224	б 225	в 226	г 227	д 228	е 229	ж 230	з 231	и 232	й 233	к 234	л 235	м 236	н 237	о 238	п 239
F0	р 240	с 241	т 242	у 243	ф 244	х 245	ц 246	ч 247	ш 248	щ 249	ъ 250	ы 251	ь 252	э 253	ю 254	я 255

Programming an Encoder

LECTURE 6

Programming an Encoder

- Using ord and char we can convert a string into and out of numeric form.
- The encoding algorithm is simple:

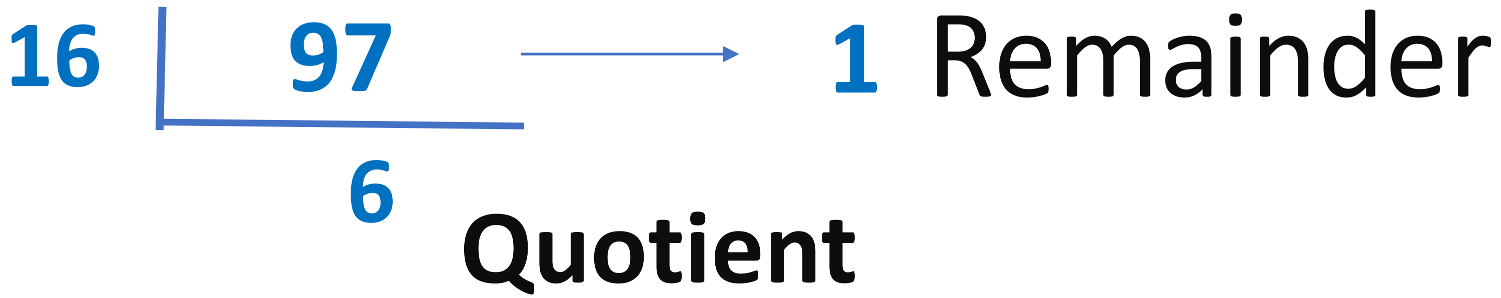
```
get the message to encode
for each character in the message:
   print the letter number of the character
```

- A `for` loop iterates over a sequence of objects, so the for loop looks like:

```
for ch in <string>
```

Conversion between Hexadecimal and Decimal

From decimal to hexadecimal



od97 = ox61   /* decimal 97 = hexadecimal 61 */

Hexadecimal Digit (aka. Half Byte)

0d0 = 0x0

0d1 = 0x1

0d2 = 0x2

0d3 = 0x3

0d4 = 0x4

0d5 = 0x5

0d6 = 0x6

0d7 = 0x7

0d8 = 0x8

0d9 = 0x9

0d10 = 0xA

0d11 = 0xB

0d12 = 0xC

0d13 = 0xD

0d14 = 0xE

0d15 = 0xF

Hexadecimal Convert to Decimal

0x61 = 6 * 16 + 1 = 6×16^1+ 1 = 0d97

0x5E83 = $5 \times 16^3 + 14 \times 16^2 + 8 \times 16^1$+ 3×16^0= 0d24195

- All strings and character are stored in computer as binary data byte (1 byte) or binary data words (2 bytes)
- 1 byte is 8 bits, 1 word is 16 bits, 1 half-byte is 4 bits.
- Every 4 bits of binary data equal 1 hexadecimal half byte

Programming an Encoder

```
# text2numbers.py
#     A program to convert a textual message into a sequence of
#         numbers, utlilizing the underlying Unicode encoding.
def main():
    print("This program converts a textual message into a sequence")
    print ("of numbers representing the Unicode encoding of the message.\n")

    # Get the message to encode
    message = input("Please enter the message to encode: ")
    print("\nHere are the Unicode codes:")

    # Loop through the message and print out the Unicode values
    for ch in message:
        print(ord(ch),  end=" ")

    print()  # blank line before prompt
```

Programming a Decoder

LECTURE 7

Programming a Decoder

- We now have a program to convert messages into a type of “code”, but it would be nice to have a program that could decode the message!
- The outline for a decoder:

```
get the sequence of numbers to decode
message = “”
for each number in the input:
   convert the number to the appropriate character
   add the character to the end of the message
print the message
```

Programming a Decoder

- The variable **message** is an accumulator variable, initially set to the empty string, the string with no characters ("").
- Each time through the loop, a number from the input is converted to the appropriate character and appended to the end of the accumulator.

Programming a Decoder

- How do we get the sequence of numbers to decode?
- Read the input as a single string, then split it apart into substrings, each of which represents one number.

Programming a Decoder

- The new algorithm

```
get the sequence of numbers as a string, inString
split inString into a sequence of smaller strings
message = ""
for each of the smaller strings:
   change the string of digits into the number it represents
   append the ASCII character for that number to message
print message
```

- Strings are objects and have useful methods associated with them

Programming a Decoder

- One of these methods is split. This will split a string into substrings based on spaces.

```
>>> "Hello string methods!".split()
['Hello', 'string', 'methods!']
```

Programming a Decoder

- Split can be used on characters other than space, by supplying the character as a parameter.

```
>>> "32,24,25,57".split(",")
['32', '24', '25', '57']
```

Programming a Decoder

- We could get the x and y values of a point in a single input string by…
  - Turning it into a list using the split method
  - Indexing the individual component strings
  - Convert these strings into their corresponding numbers using `int` or `float`

```
coords = input("Enter the point coordinates (x,y): ").split(",")
x,y = float(coords[0]), float(coords[1])
```

Programming a Decoder

```
# numbers2text.py
#     A program to convert a sequence of Unicode numbers into
#         a string of text.

def main():
    print ("This program converts a sequence of Unicode numbers into")
    print ("the string of text that it represents.\n")

    # Get the message to encode
    inString = input("Please enter the Unicode-encoded message: ")

    # Loop through each substring and build Unicde message
    message = ""
    for numStr in inString.split():
        # convert the (sub)string to a number
        codeNum = int(numStr)
        # append character to message
        message = message + chr(codeNum)

    print("\nThe decoded message is:", message)
```

Programming a Decoder

- The `split` function produces a sequence of strings. `numString` gets each successive substring.
- Each time through the loop, the next substring is converted to the appropriate Unicode character and appended to the end of message.

Programming a Decoder

```
-------------------------------------------------------------------------
This program converts a textual message into a sequence
of numbers representing the Unicode encoding of the message.

Please enter the message to encode: CS120 is fun!

Here are the Unicode codes:
67 83 49 50 48 32 105 115 32 102 117 110 33

--------------------------------------------------------------------------
This program converts a sequence of Unicode numbers into
the string of text that it represents.

Please enter the ASCII-encoded message: 67 83 49 50 48 32 105 115 32 102 117 110 33
The decoded message is: CS120 is fun!
```

More String Methods

LECTURE 8

String Methods

Function	Meaning
s.capitalize()	Copy of s with only the first character capitalized
s.center(width)	Copy of s centered in a field of given width
s.count(sub)	Count the number of occurrences of sub in s
s.find(sub)	Find the first position where sub occurs in s
s.join(list)	Concatenate list into a string, using s as separator
s.ljust(width)	Like center, but s is left-justified
s.lower()	Copy of s in all lowercase characters

String Methods

Function	Meaning
s.lstrip()	Copy of s with leading white space removed
s.replace(oldsub, newsub)	Replace all occurrences of oldsub in s with newsub
s.rfind(sub)	Like find, but returns the rightmost position
s.rjust(width)	Like center, but s is right-justified
s.rstrip()	Copy of s with trailing white space removed
s.split()	Split s into a list of substrings (see text)
s.title()	Copy of s with first character of each word capitalized
s.upper()	Copy of s with all characters converted to uppercase.

More String Methods

- There are a number of other string methods. Try them all!
  - `s.capitalize()` – Copy of s with only the first character capitalized
  - `s.title()` – Copy of s; first character of each word capitalized
  - `s.center(width)` – Center s in a field of given width

More String Methods

- `s.count(sub)` – Count the number of occurrences of sub in s
- `s.find(sub)` – Find the first position where sub occurs in s
- `s.join(list)` – Concatenate list of strings into one large string using s as separator.
- `s.ljust(width)` – Like center, but s is left-justified

More String Methods

- `s.lower()` – Copy of s in all lowercase letters
- `s.lstrip()` – Copy of s with leading whitespace removed
- `s.replace(oldsub, newsub)` – Replace occurrences of oldsub in s with newsub
- `s.rfind(sub)` – Like find, but returns the right-most position
- `s.rjust(width)` – Like center, but s is right-justified

More String Methods

- `s.rstrip()` – Copy of s with trailing whitespace removed
- `s.split()` – Split s into a list of substrings
- `s.upper()` – Copy of s; all characters converted to uppercase

List Methods

LECTURE 9

Lists Have Methods, Too

- The `append` method can be used to add an item at the end of a list.

```
squares = []
for x in range(1,101):
    squares.append(x*x)
```

- We start with an empty list ([]) and each number from 1 to 100 is squared and appended to it (`[1,4,9,…,10000]`).

Lists Have Methods, Too

- We can use an alternative approach in the decoder program.
- The statement

```
message = message + chr(codeNum)
```

  - essentially creates a copy of the message so far and tacks one character on the end.
  - As we build up the message, we keep recopying a longer and longer string just to add a single character at the end!

Lists Have Methods, Too

- We can avoid this recopying by using lists of characters where each new character is `append`ed to the end of the existing list.
- Since lists are mutable, the list is changed “in place” without having to copy the content over to a new object.

Lists Have Methods, Too

- When done, we can use `join` to concatenate the characters into a string.

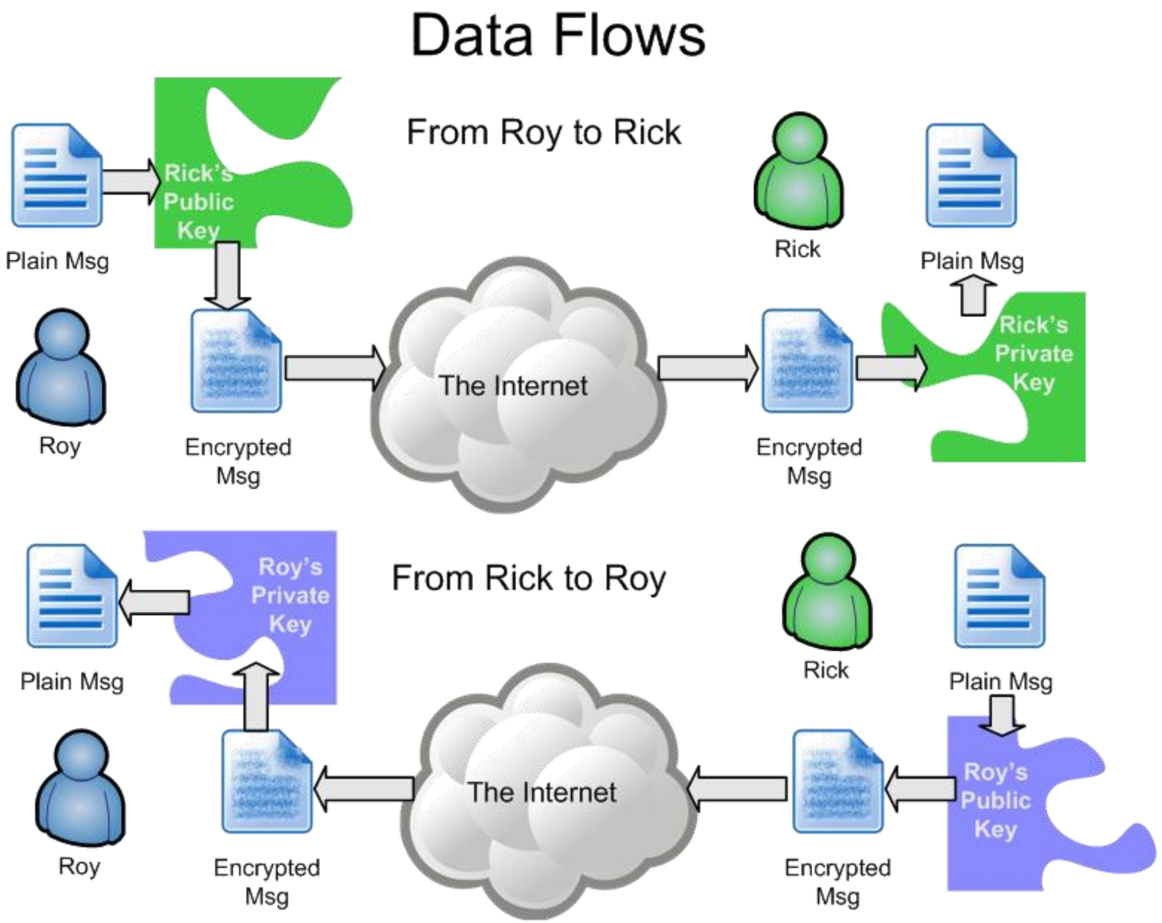
Lists Have Methods, Too

```
# numbers2text2.py
#     A program to convert a sequence of Unicode numbers into
#         a string of text. Efficient version using a list accumulator.
def main():
    print("This program converts a sequence of Unicode numbers into")
    print("the string of text that it represents.\n")

    # Get the message to encode
    inString = input("Please enter the Unicode-encoded message: ")
    # Loop through each substring and build Unicode message
    chars = []
    for numStr in inString.split():
        codeNum = int(numStr)             # convert digits to a number
        chars.append(chr(codeNum))         # accumulate new character

    message = "".join(chars)
    print("\nThe decoded message is:", message)
```

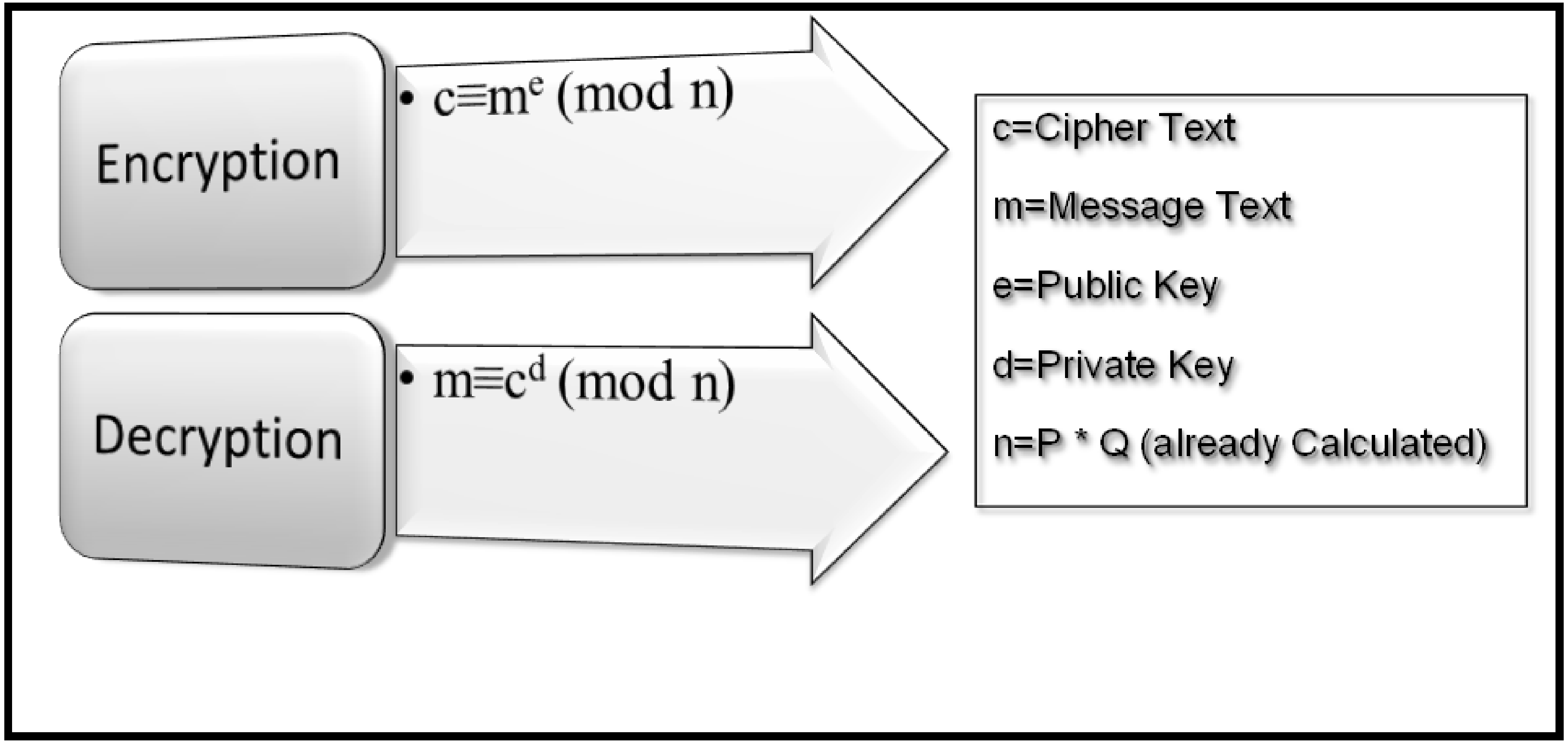
Encryption

LECTURE 10

Public Key System



RSA Algorithm (Rivest-Shamir-Aldleman)



Encryption

- $c \equiv m^e \pmod{n}$

Decryption

- $m \equiv c^d \pmod{n}$

c=Cipher Text

m=Message Text

e=Public Key

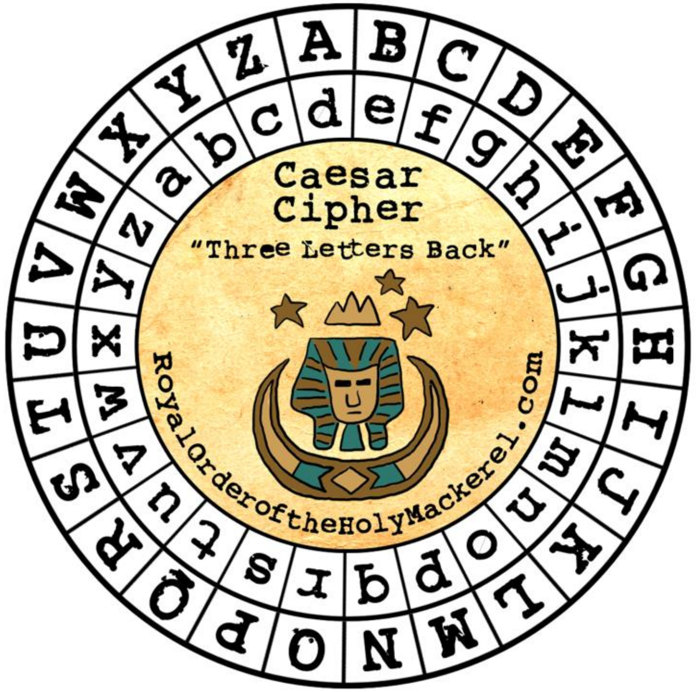
d=Private Key

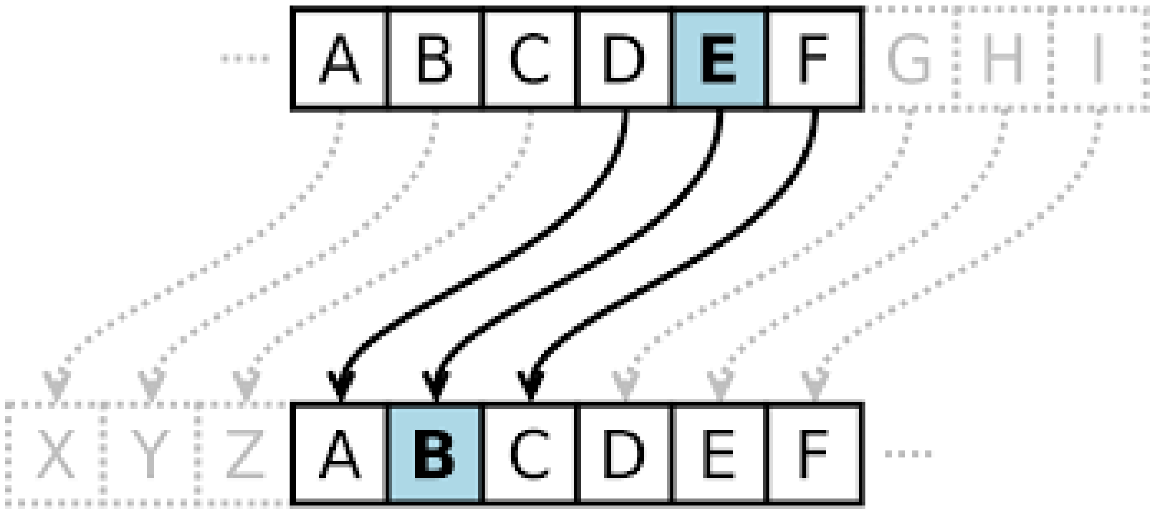
n=P * Q (already Calculated)

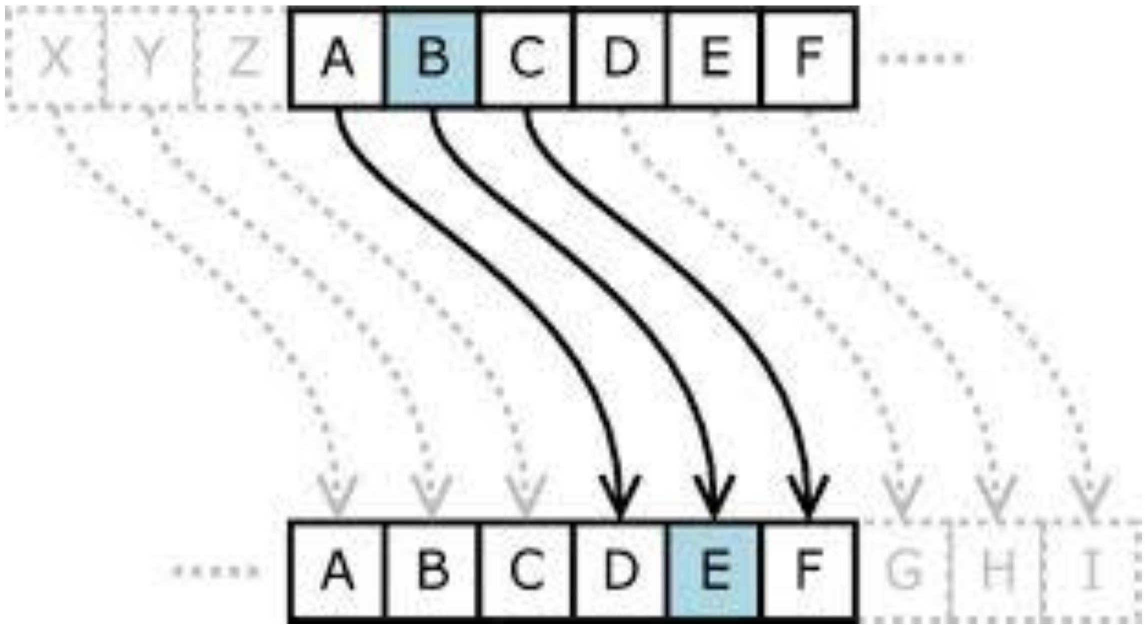
The Royal Order
of the
Holy Mackerel

Caesar Cipher Disk

In the first season of Gravity Falls, Alex Hirsch used the Caesar Cipher (shifted three letters back) to encode secret messages hidden throughout the first 6 episodes (most notably in the end credits) and beyond. Similarly, the first two challenges of the Royal Order of the Holy Mackerel are also encoded with this same cipher! Use decoder disk below to reveal the secret messages!


A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
d e f g h i j k l m n o p q r s t u v w x y z a b c
Caesar
Cipher
"Three Letters Back"
RoyalOrderoftheHolyMackerel.com


A B C D E F G H I
X Y Z A B C D E F


X Y Z A B C D E F
A B C D E F G H I

From Encoding to Encryption

- The process of encoding information for the purpose of keeping it secret or transmitting it privately is called `encryption`.
- `Cryptography` is the study of encryption methods.
- Encryption is used when transmitting credit card and other personal information to a web site.

From Encoding to Encryption

- Strings are represented as a sort of encoding problem, where each character in the string is represented as a number that’s stored in the computer.
- The code that is the mapping between character and number is an industry standard, so it’s not “secret”.

From Encoding to Encryption

- The encoding/decoding programs we wrote use a **substitution** cipher, where each character of the original message, known as the **plaintext**, is replaced by a corresponding symbol in the cipher alphabet.
- The resulting code is known as the **ciphertext**.

From Encoding to Encryption

- This type of code is relatively easy to break.
- Each letter is always encoded with the same symbol, so using statistical analysis on the frequency of the letters and trial and error, the original message can be determined.

From Encoding to Encryption

- Modern encryption converts messages into numbers.
- Sophisticated mathematical formulas convert these numbers into new numbers – usually this transformation consists of combining the message with another value called the “**key**”

From Encoding to Encryption

- To decrypt the message, the receiving end needs an appropriate key so the encoding can be reversed.
- In a **private key** system the same key is used for encrypting and decrypting messages. Everyone you know would need a copy of this key to communicate with you, but it needs to be kept a secret.

From Encoding to Encryption

- In `public key` encryption, there are separate keys for encrypting and decrypting the message.
- In public key systems, the encryption key is made publicly available, while the decryption key is kept private.
- Anyone with the public key can send a message, but only the person who holds the private key (decryption key) can decrypt it.

I/O as String Manipulation

LECTURE 11

Input/Output as String Manipulation

- Often we will need to do some string operations to prepare our string data for output (“pretty it up”)
- Let’s say we want to enter a date in the format “05/24/2015” and output
- “May 24, 2015.” How could we do that?

Input/Output as String Manipulation

```
Input the date in mm/dd/yyyy format (dateStr)
Split dateStr into month, day, and year strings
Convert the month string into a month number
Use the month number to lookup the month name
Create a new date string in the form "Month Day, Year"
Output the new date string
```

Input/Output as String Manipulation

- The first two lines are easily implemented!

```
dateStr = input("Enter a date (mm/dd/yyyy): ")
monthStr, dayStr, yearStr = dateStr.split("/")
```

- The date is input as a string, and then “unpacked” into the three variables by splitting it at the slashes and using simultaneous assignment.

Input/Output as String Manipulation

- Next step: Convert monthStr into a number
- We can use the ***int*** function on monthStr to convert "05", for example, into the integer 5. (`int("05") = 5`)

Input/Output as String Manipulation

- Note: eval would work, but for the leading 0

```
>>> int("05")
5
>>> eval("05")
 Traceback (most recent call last):
File "<pyshell#9>", line 1, in <module>
eval("05")
File "<string>", line 1
  05
       ^
  SyntaxError: invalid token
```

- This is historical baggage. A leading 0 used to be used for base 8 (octal) literals in Python.

Input/Output as String Manipulation

```
months = ["January", "February", …,
"December"]

monthStr = months[int(monthStr) - 1]
```

- Remember that since we start counting at 0, we need to subtract one from the month.
- Now let’s concatenate the output string together!

Input/Output as String Manipulation

```
print ("The converted date is:", monthStr,\
  dayStr+",", yearStr)
```

- Notice how the comma is appended to dayStr with concatenation!

```
>>> main()
Enter a date (mm/dd/yyyy): 01/23/2010
The converted date is: January 23, 2010
```

Input/Output as String Manipulation

- Sometimes we want to convert a number into a string.
- We can use the str function.

```
>>> str(500)
'500'
>>> value = 3.14
>>> str(value)
'3.14'
>>> print("The value is", str(value) + ".")
The value is 3.14.
```

Input/Output as String Manipulation

- If value is a string, we can concatenate a period onto the end of it.

- If value is an int, what happens?

```
>>> value = 3.14
>>> print("The value is", value + ".")
The value is

Traceback (most recent call last):
  File "<pyshell#10>", line 1, in -toplevel-
    print "The value is", value + "."
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

Input/Output as String Manipulation

- We now have a complete set of type conversion operations:

Function	Meaning
float(<expr>)	Convert expr to a floating point value
int(<expr>)	Convert expr to an integer value
str(<expr>)	Return a string representation of expr
eval(<string>)	Evaluate string as an expression

Case Study: dateconvert.py

```
# dateconvert.py
def main():
  # get the date
  dateStr =input("Enter ad date (mm/dd/yyyy):")
  # split into components
  monthStr, dayStr, yearStr = dateStr.split("/")
  # Convert monthStr to the month name
  months = ["January", "February", "March", "April",
            "May","June", "July", "August",
            "September", "October", "November", "December"]
  monthStr = months[int(monthStr)-1]
  # Output result in month day, year format
  print("The converted date is:", monthStr, dayStr+",", yearStr)
main()
```

Case Study: dateconvert2.py

```python
# dateconvert2.py
# input in the following format: 1,2,1968
def main():
    # get the date
    day, month, year =eval(input("Enter day. month, and year numbers:"))
    date1 = str(month)+"/"+str(day)+"/"+str(year)
    # Convert monthStr to the month name
    months = ["January", "February", "March", "April",
              "May","June", "July", "August",
              "September", "October", "November", "December"]
    monthStr = months[month-1]
    date2 = monthStr+" "+str(day)+", "+str(year)
    print("the date is", date1, "or", date2+".")
main()
```

String Formatting

LECTURE 12

String Formatting

- String formatting is an easy way to get beautiful output!

```
Change Counter

Please enter the count of each coin type.
Quarters: 6
Dimes: 0
Nickels: 0
Pennies: 0

The total value of your change is 1.5
```

- Shouldn’t that be more like $1.50??

String Formatting

We can format our output by modifying the print statement as follows:

```
print("The total value of your change is ${0:0.2f}".format(total))
```

- Now we get something like:

```
The total value of your change is $1.50
```

- Key is the string format method.

String Formatting

```
<template-string>.format(<values>)
```

- {} within the template-string mark “slots” into which the values are inserted.
- Each slot has description that includes format specifier telling Python how the value for the slot should appear.

String Formatting

```
print("The total value of your change is ${0:0.2f}".format(total)
```

- The template contains a single slot with the description: 0:0.2f
- Form of description:

```
<index>:<format-specifier>
```

- Index tells which parameter to insert into the slot. In this case, total.

String Formatting

- The formatting specifier has the form:

```
<width>.<precision><type>
```

- f means "fixed point" number
- <width> tells us how many spaces to use to display the value. 0 means to use as much space as necessary.
- <precision> is the number of decimal places.

String Formatting

```
>>> "Hello {0} {1}, you may have won ${2}" .format("Mr.", "Smith", 10000)
'Hello Mr. Smith, you may have won $10000'

>>> 'This int, {0:5}, was placed in a field of width 5'.format(7)
'This int,     7, was placed in a field of width 5'

>>> 'This int, {0:10}, was placed in a field of witdh 10'.format(10)
'This int,         10, was placed in a field of witdh 10'

>>> 'This float, {0:10.5}, has width 10 and precision 5.'.format(3.1415926)
'This float,    3.1416, has width 10 and precision 5.'

>>> 'This float, {0:10.5f},  is fixed at 5 decimal places.'.format(3.1415926)
'This float,   3.14159, has width 0 and precision 5.'

>>> "Compare {0} and {0:0.20}".format(3.14)
'Compare 3.14 and 3.1400000000000001243'
```

String Formatting

- Numeric values are right-justified and strings are left-justified, by default.
- You can also specify a justification before the width.

```
>>> "left justification: {0:<5}.format("Hi!")
'left justification: Hi!  '
>>> "right justification: {0:>5}.format("Hi!")
'right justification:   Hi!'
>>> "centered: {0:^5}".format("Hi!")
'centered:  Hi! '
```

Format Specification

{<index>:<format-specifier>}

- `<index>:` The index number in the format function values
- `<format-specifier>: <width>.<precision><type>`
- `width:` specifies how many spaces the value should take up. If the value is less than the specified width, it is padded. If the value is more than the specified width, it will take as much as is required to show the value. 0 is used to use as much as you need.
- `precision:` The decimal places to show a float value.
- `type:` output number format (see next)

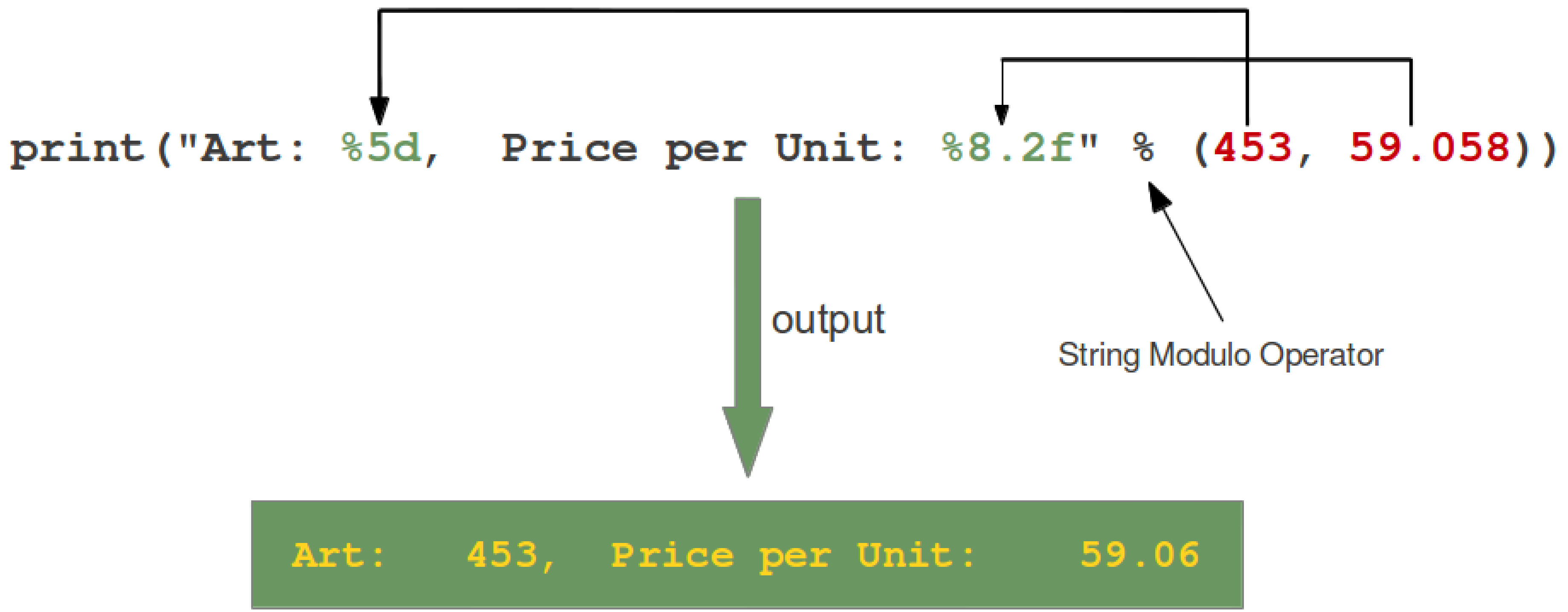
Python Format Types

Conversion	Meaning	Notes
d	Signed integer decimal.	
i	Signed integer decimal.	
o	Unsigned octal.	(1)
u	Unsigned decimal.	
x	Unsigned hexadecimal (lowercase).	(2)
X	Unsigned hexadecimal (uppercase).	(2)
e	Floating point exponential format (lowercase).	
E	Floating point exponential format (uppercase).	
f	Floating point decimal format.	
F	Floating point decimal format.	
g	Same as "e" if exponent is greater than -4 or less than precision, "f" otherwise.	
G	Same as "E" if exponent is greater than -4 or less than precision, "F" otherwise.	
c	Single character (accepts integer or single character string).	
r	String (converts any python object using repr()).	(3)
s	String (converts any python object using str()).	(4)
%	No argument is converted, results in a "%" character in the result.	

String Indentation

Formatting Symbol	Indentation
<	Left
>	Right
^	Center

Python Formatted String by Position



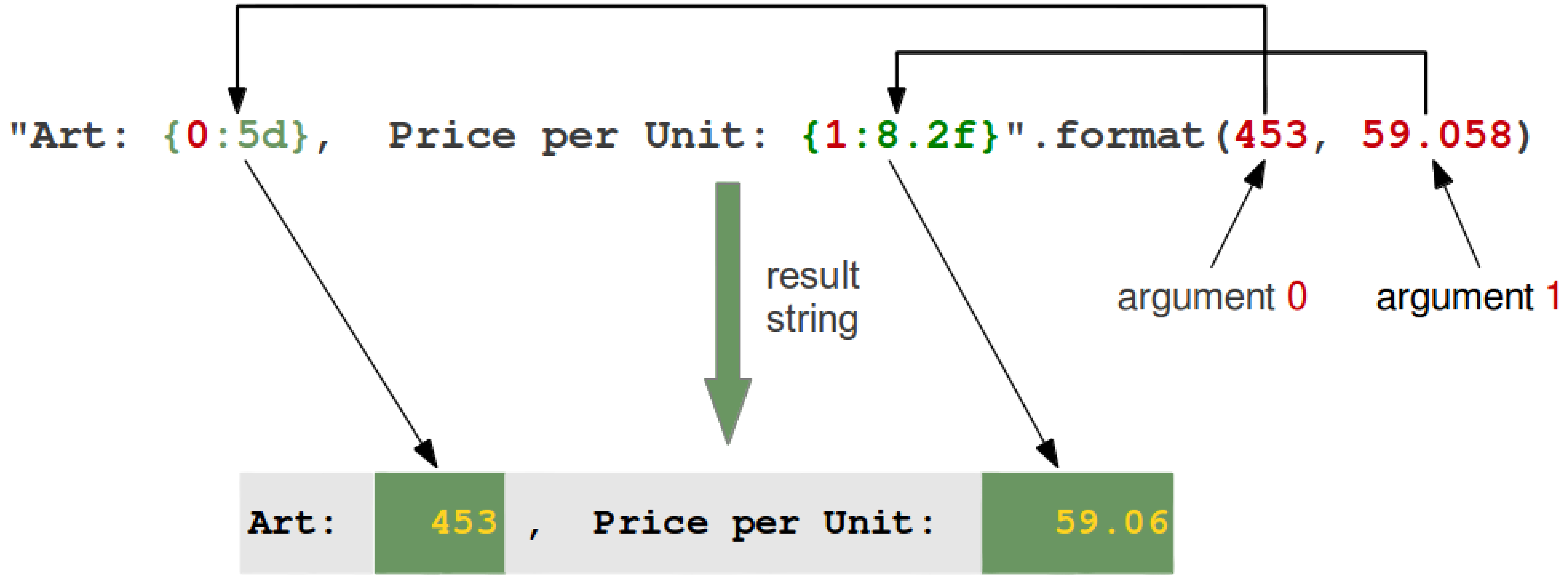
```
print("Art: %5d,  Price per Unit: %8.2f" % (453, 59.058))
```

output

String Modulo Operator

```
Art:   453,  Price per Unit:    59.06
```

Python Formatted String by Index



```
"Art: {0:5d},  Price per Unit: {1:8.2f}".format(453, 59.058)
```

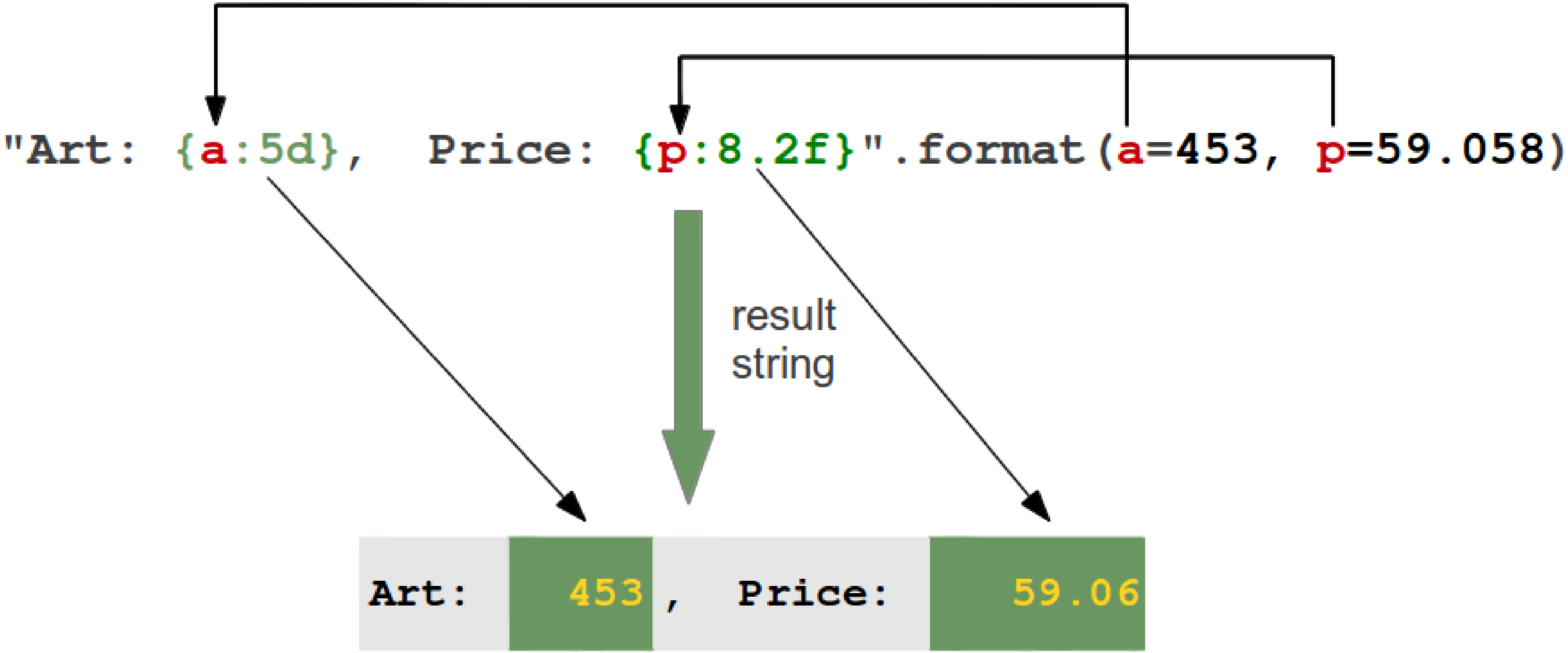
result string

argument 0

argument 1

```
Art:   453 ,  Price per Unit:    59.06
```

Python Formatted String by Parameter



```
"Art: {a:5d}, Price: {p:8.2f}".format(a=453, p=59.058)
```

result string

```
Art:   453, Price:    59.06
```

Standard String Formatting

LECTURE 13

Format Codes (Builtins)

- For builtins, there are standard format codes

Old Format	New Format	Description
"%d"	"d"	Decimal Integer
"%f"	"f"	Floating point
"%s"	"s"	String
"%e"	"e"	Scientific notation
"%x"	"x"	Hexadecimal

- Plus there are some brand new codes

	"o"	Octal
	"b"	Binary
	"%"	Percent

Formatted String

The %3d specifier means a minimum width of three spaces, which, by default, will be right-justified:

"%3d“ % 0	0
"%3d“ % 123456789	123456789
"%3d“ % -10	-10
"%3d“ % -123456789	-123456789

Left-justifying Formatted String

To left-justify integer output with printf, just add a minus sign (-) after the % symbol, like this:

"%-3d" % 0	0
"%-3d" % 123456789	123456789
"%-3d" % -10	-10
"%-3d" % -123456789	-123456789

The Formatted String zero-fill option

To zero-fill your printf integer output, just add a zero (0) after the % symbol, like this:

"%03d" % 0	000
"%03d" % 1	001
"%03d" % 123456789	123456789
"%03d" % -10	-10
"%03d" % -123456789	-123456789

Integer Formatted String

As a summary of printf integer formatting, here’s a little collection of integer formatting examples. Several different options are shown, including a minimum width specification, left-justified, zero-filled, and also a plus sign for positive numbers.

Description	Code	Result
At least five wide	"'%5d'" % 10	' 10'
At least five-wide, left-justified	"'%-5d'" % 10	'10 '
At least five-wide, zero-filled	"'%05d'" % 10	'00010'
At least five-wide, with a plus sign	"'%+5d'" % 10	' +10'
Five-wide, plus sign, left-justified	"'%-+5d'" % 10	'+10 '

Formatted String - floating point numbers

Here are several examples showing how to format floating-point numbers with printf:

Description	Code	Result
Print one position after the decimal	"'%.1f'" % 10.3456	`'10.3'`
Two positions after the decimal	"'%.2f'" % 10.3456	`'10.35'`
Eight-wide, two positions after the decimal	"'%8.2f'" % 10.3456	`'   10.35'`
Eight-wide, four positions after the decimal	"'%8.4f'" % 10.3456	`' 10.3456'`
Eight-wide, two positions after the decimal, zero-filled	"'%08.2f'" % 10.3456	`'00010.35'`
Eight-wide, two positions after the decimal, left-justified	"'%-8.2f'" % 10.3456	`'10.35   '`
Printing a much larger number with that same format	"'%-8.2f'" % 101234567.3456	`'101234567.35'`

Formatted String

Here are several examples that show how to format string output with printf:

Description	Code	Result
A simple string	"'%s’” % "Hello"	'Hello'
A string with a minimum length	"'%10s’” % "Hello"	' Hello'
Minimum length, left-justified	"'%-10s’” % "Hello"	'Hello '

Practice Question:

1. I have 7 bags of candy. They totally worth $100. Then, show me how much money each bag worth on the average down to the cents.

(Note: cents.py)

2. Write the result for the following functions: (round to integers)

- int(12.3)
- floor(4.0)
- ceil(5.2)
- round(6.45)
- round(3.58)

Better Change Counter

LECTURE 14

Better Change Counter

- With what we know now about floating point numbers, we might be uneasy about using them in a money situation.
- One way around this problem is to keep track of money in cents using an int or long int, and convert it into dollars and cents when output.

Better Change Counter

- If total is a value in cents (an int),

```
dollars = total//100
cents = total%100
```

- Cents is printed using width 0>2 to right-justify it with leading 0s (if necessary) into a field of width 2.
- Thus 5 cents becomes '05'

Better Change Counter

```python
# change2.py
#   A program to calculate the value of some change in dollars.
#   This version represents the total cash in cents.

def main():
    print ("Change Counter\n")
    print ("Please enter the count of each coin type.")
    quarters = int(input("Quarters: "))
    dimes = int(input("Dimes: "))
    nickels = int(input("Nickels: "))
    pennies = int(input("Pennies: "))
    total = quarters * 25 + dimes * 10 + nickels * 5 + pennies

    print ("The total value of your change is ${0}.{1:0>2}"
         .format(total//100, total%100))
```

Better Change Counter

```
>>> main()
Change Counter

Please enter the count of each coin type.
Quarters: 0
Dimes: 0
Nickels: 0
Pennies: 1

The total value of your change is $0.01
```

```
>>> main()
Change Counter

Please enter the count of each coin type.
Quarters: 12
Dimes: 1
Nickels: 0
Pennies: 4

The total value of your change is $3.14
```

Files: Multi-line Strings

LECTURE 15

Files: Multi-line Strings

- A **file** is a sequence of data that is stored in secondary memory (disk drive).
- Files can contain any data type, but the easiest to work with are text.
- A file usually contains more than one line of text.
- Python uses the standard newline character (`\n`) to mark line breaks.

Multi-Line Strings

```
Hello
World
```

- `Goodbye 32`
- **When stored in a file:**
- `Hello\nWorld\n\nGoodbye 32\n`

Multi-Line Strings

- This is exactly the same thing as embedding \n in print statements.
- Remember, these special characters only affect things when printed. They don’t do anything during evaluation.

File Processing I

LECTURE 16

File Processing

- The process of **opening** a file involves associating a file on disk with an object in memory.
- We can manipulate the file by manipulating this object.
  - Read from the file
  - Write to the file

File Processing

- When done with the file, it needs to be **closed**. Closing the file causes any outstanding operations and other bookkeeping for the file to be completed.
- In some cases, not properly closing a file could result in data loss.

File Processing

- Reading a file into a word processor
  - File opened
  - Contents read into RAM
  - File closed
  - Changes to the file are made to the copy stored in memory, not on the disk.

File Processing

- Saving a word processing file
  - The original file on the disk is reopened in a mode that will allow writing (this actually erases the old contents)
  - File writing operations copy the version of the document in memory to the disk
  - The file is closed

File Processing

- Working with text files in Python
- Associate a disk file with a file object using the open function
  - `<filevar> = open(<name>, <mode>)`
  - name is a string with the actual file name on the disk. The mode is either ‘r’ or ‘w’ depending on whether we are reading or writing the file.
  - `infile = open("numbers.dat", "r")`

File Methods

LECTURE 17

File Methods

- `<file>.read()` – returns the entire remaining contents of the file as a single (possibly large, multi-line) string
- `<file>.readline()` – returns the next line of the file. This is all text up to and **including** the next newline character
- `<file>.readlines()` – returns a list of the remaining lines in the file. Each list item is a single line including the newline characters.

File Processing II

LECTURE 18

File Processing

```
# printfile.py
#     Prints a file to the screen.

def main():
    fname = input("Enter filename: ")
    infile = open(fname,'r')
    data = infile.read()
    print(data)
```

- First, prompt the user for a file name
- Open the file for reading
- The file is read as one string and stored in the variable data

File Processing

- readline can be used to read the next line from a file, including the trailing newline character

```
infile = open(someFile, "r")
for i in range(5):
 line = infile.readline()
   print(line[:-1])
```

- This reads the first 5 lines of a file
- Slicing is used to strip out the newline characters at the ends of the lines

File Processing

- Another way to loop through the contents of a file is to read it in with `readlines` and then loop through the resulting list.

```
infile = open(someFile, "r")
for line in infile.readlines():
 # Line processing here
  infile.close()
```

File Processing

- Python treats the file itself as a sequence of lines!

```
infile = open(someFile, "r")
for line in infile:
 # process the line here
  infile.close()
```

File Processing

- Opening a file for writing prepares the file to receive data
- If you open an existing file for writing, you wipe out the file's contents. If the named file does not exist, a new one is created.

```
outfile = open("mydata.out", "w")
print(<expressions>, file=outfile)
```

Case Study: Batch Usernames

LECTURE 19

Example Program: Batch Usernames

- **Batch** mode processing is where program input and output are done through files (the program is not designed to be interactive)
- Let’s create usernames for a computer system where the first and last names come from an input file.

Example Program: Batch Usernames

```
# userfile.py
#    Program to create a file of usernames in batch mode.

def main():
    print ("This program creates a file of usernames from a")
    print ("file of names.")

    # get the file names
    infileName = input("What file are the names in? ")
    outfileName = input("What file should the usernames go in? ")

    # open the files
    infile = open(infileName, 'r')
    outfile = open(outfileName, 'w')
```

Example Program: Batch Usernames

```
# process each line of the input file
    for line in infile:
        # get the first and last names from line
        first, last = line.split()
        # create a username
        uname = (first[0]+last[:7]).lower()
        # write it to the output file
        print(uname, file=outfile)

    # close both files
    infile.close()
    outfile.close()

    print("Usernames have been written to", outfileName)
```

Example Program: Batch Usernames

- Things to note:
  - It’s not unusual for programs to have multiple files open for reading and writing at the same time.
  - The lower method is used to convert the names into all lower case, in the event the names are mixed upper and lower case.

File Dialogs

LECTURE 20

File Dialogs

- A common problem with file manipulation programs is figuring out exactly how to specify the file that you want to use.
- With no additional information, Python will look in the “current” directory for files.
- Most modern operating systems use file names having a form like <name>.<type> where type is a short indicator of what the file contains, e.g. txt (text file).

File Dialogs

- One problem: some operating systems (Windows and MacOS) by default only show the part of the name preceeding the period, so it can be hard to figure out the complete file name.
- It’s even harder when the file is located somewhere other than the current directory in your secondary memory! Then we will need the complete path in addition to the file name.

File Dialogs

On Windows, the complete file name may look like

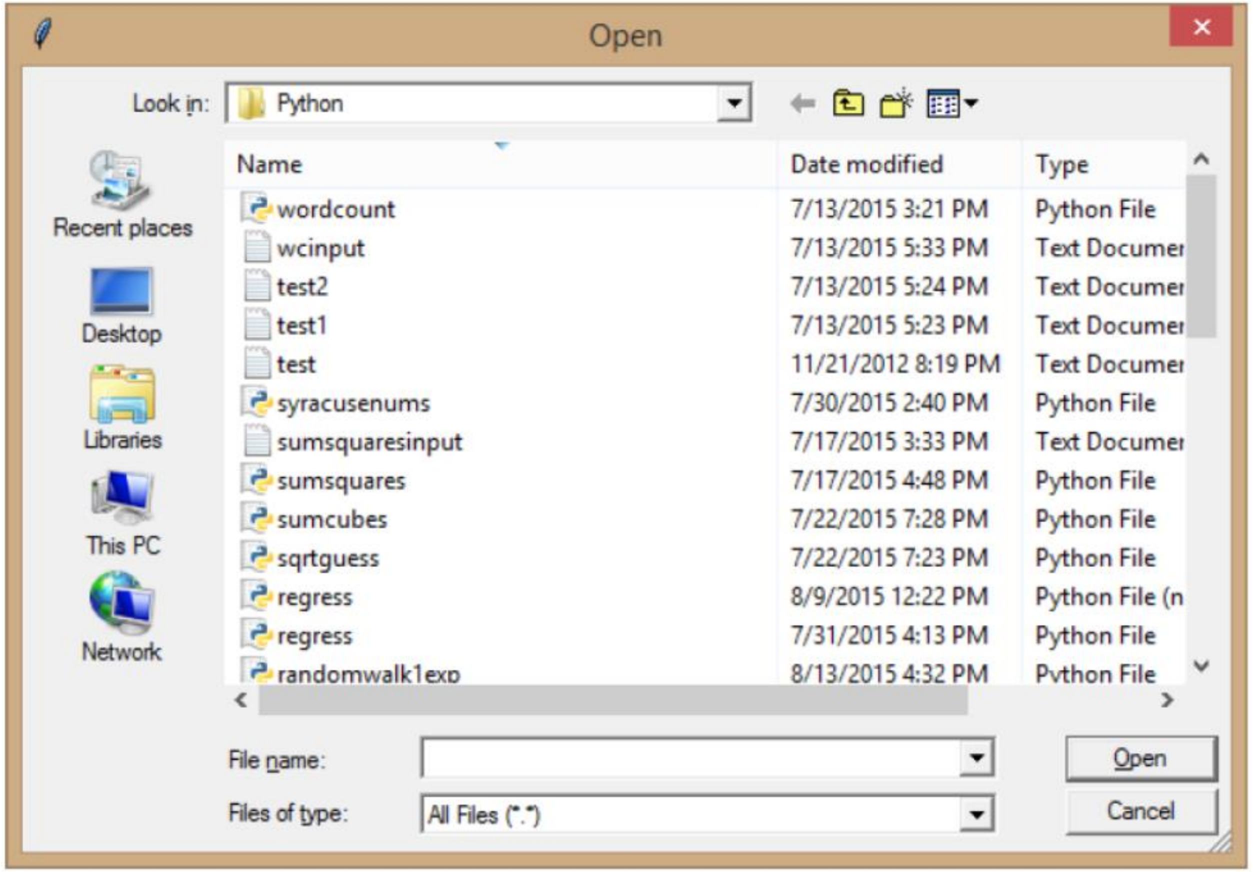
- `C:/users/susan/Documents/Python_Programs/users.txt`
- The solution? Allow the users to browse the file system visually and navigate to the file.
- This is a common enough operation that most operating systems provide a standard way to do this, usually incorporating a dialog box.

File Dialogs

- To ask the user for the name of a file to open, you can use askopenfilename from tkinter.filedialog.

```
from tkinter.filedialog import
askopenfilename
…
infileName = aksopenfilename()
infile = open(infileName, "r")
```

File Dialogs



Look in: Python

Name	Date modified	Type
wordcount	7/13/2015 3:21 PM	Python File
wcinput	7/13/2015 5:33 PM	Text Documer
test2	7/13/2015 5:24 PM	Text Documer
test1	7/13/2015 5:23 PM	Text Documer
test	11/21/2012 8:19 PM	Text Documer
syracusenums	7/30/2015 2:40 PM	Python File
sumsquaresinput	7/17/2015 3:33 PM	Text Documer
sumsquares	7/17/2015 4:48 PM	Python File
sumcubes	7/22/2015 7:28 PM	Python File
sqrtguess	7/22/2015 7:23 PM	Python File
regress	8/9/2015 12:22 PM	Python File (n
regress	7/31/2015 4:13 PM	Python File
randomwalk1exp	8/13/2015 4:32 PM	Python File

File name:

Files of type: All Files (*.*)

File Dialogs

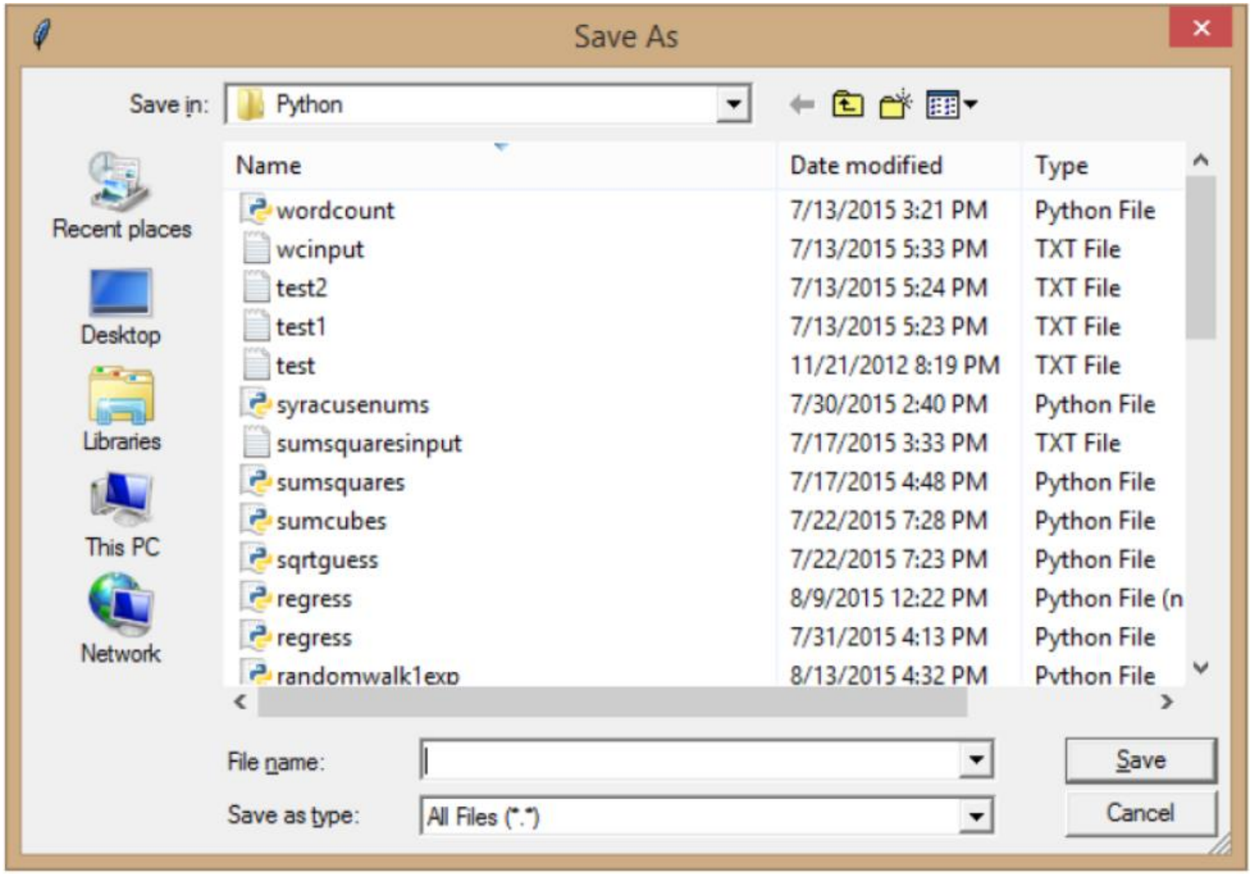
- When the user clicks the “Open” button, the complete path name of the file is returned as a string and saved into the variable `infileName`.
- If the user clicks “Cancel”, the function returns an empty string.

File Dialogs

- To ask the user for the name of a file to save, you can use `asksaveasfilename` from `tkinter.filedialog`.

```
from tkinter.filedialog import asksaveasfilename
…
outfileName = asksaveasfilename()
outfile = open(outfileName, "w")
```

File Dialogs



Save in: Python

Name	Date modified	Type
wordcount	7/13/2015 3:21 PM	Python File
wcinput	7/13/2015 5:33 PM	TXT File
test2	7/13/2015 5:24 PM	TXT File
test1	7/13/2015 5:23 PM	TXT File
test	11/21/2012 8:19 PM	TXT File
syracusenums	7/30/2015 2:40 PM	Python File
sumsquaresinput	7/17/2015 3:33 PM	TXT File
sumsquares	7/17/2015 4:48 PM	Python File
sumcubes	7/22/2015 7:28 PM	Python File
sqrtguess	7/22/2015 7:23 PM	Python File
regress	8/9/2015 12:22 PM	Python File (n
regress	7/31/2015 4:13 PM	Python File
randomwalk1exp	8/13/2015 4:32 PM	Python File

File name:

Save as type: All Files (*.*)

Homework

LECTURE 21

Homework 5

1. Read Chapter Summary
2. Do True/False Questions, Multiple Choice Questions, Discussion in Chapter 5
3. Work on programming exercises 2, 3, 7/8(same program)