

Brief Python

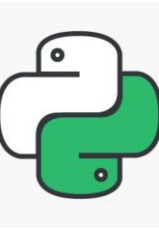
First Python Course for Beginners

Chapter 2: Conditionals

Dr. Eric Chou

IEEE Senior Member





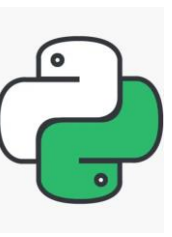
Topics

- Flow Chart
- Boolean Value
- Character and String
- Boolean Expression
- Conditional Structures



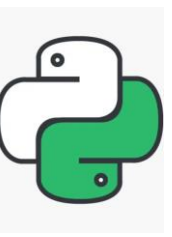
Flow Chart

LECTURE 1



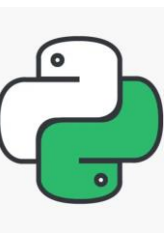
What is Flow Chart?

- Flowchart is a graphical representation of an algorithm. Programmers often use it as a program-planning tool to solve a problem. It makes use of symbols which are connected among them to indicate the flow of information and processing.
- The process of drawing a flowchart for an algorithm is known as “flowcharting”.



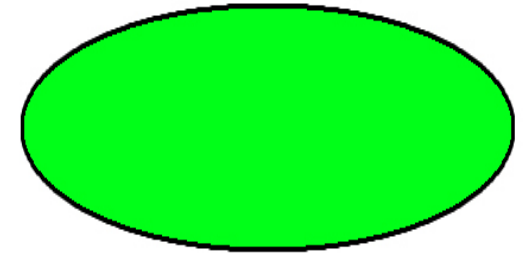
Basic Symbols used in Flowchart Designs

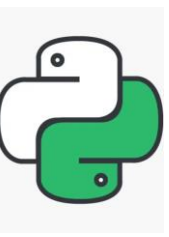
1. Terminals
2. Input/Output
3. Processing
4. Decision
5. Connector



Terminal

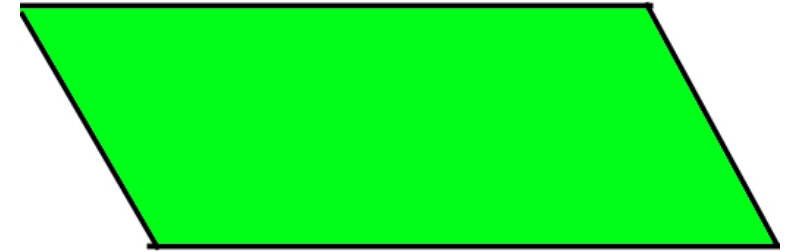
- The oval symbol indicates Start, Stop and Halt in a program's logic flow. A pause/halt is generally used in a program logic under some error conditions. Terminal is the first and last symbols in the flowchart.

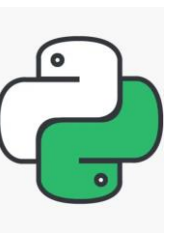




Input/Output

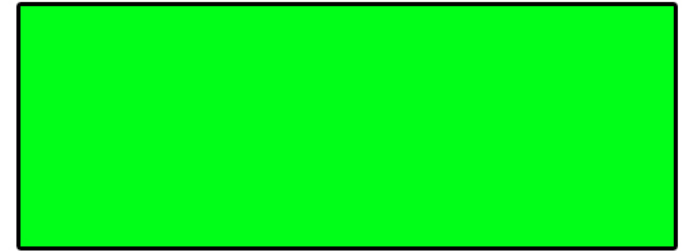
- A parallelogram denotes any function of input/output type. Program instructions that take input from input devices and display output on output devices are indicated with parallelogram in a flowchart.

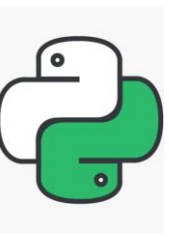




Processing

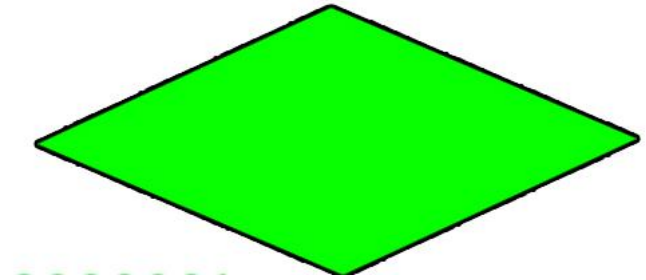
- A box represents arithmetic instructions. All arithmetic processes such as adding, subtracting, multiplication and division are indicated by action or process symbol.

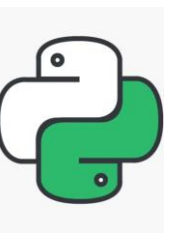




Decision

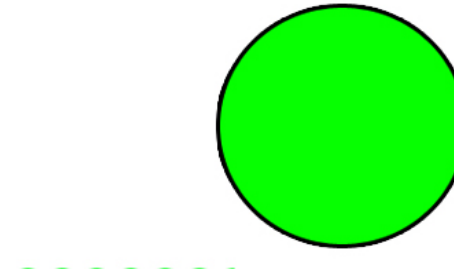
- Diamond symbol represents a decision point. Decision based operations such as yes/no question or true/false are indicated by diamond in flowchart.

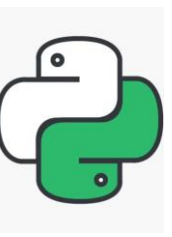




Connectors

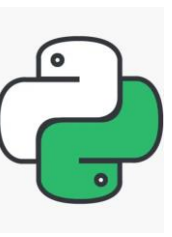
- Whenever flowchart becomes complex or it spreads over more than one page, it is useful to use connectors to avoid any confusions. It is represented by a circle.





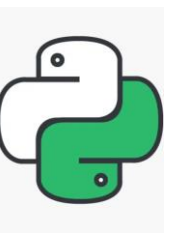
Flow lines

- Flow lines indicate the exact sequence in which instructions are executed. Arrows represent the direction of flow of control and relationship among different symbols of flowchart.



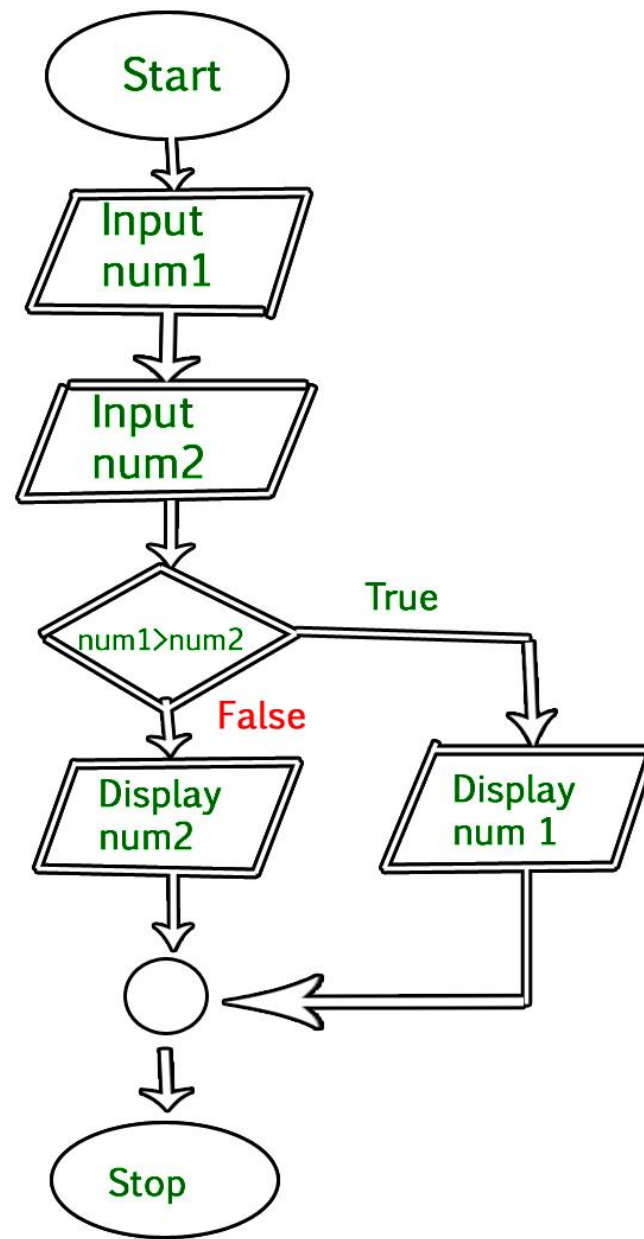
Exercise

- Draw a Flow Chart for your daily work schedule

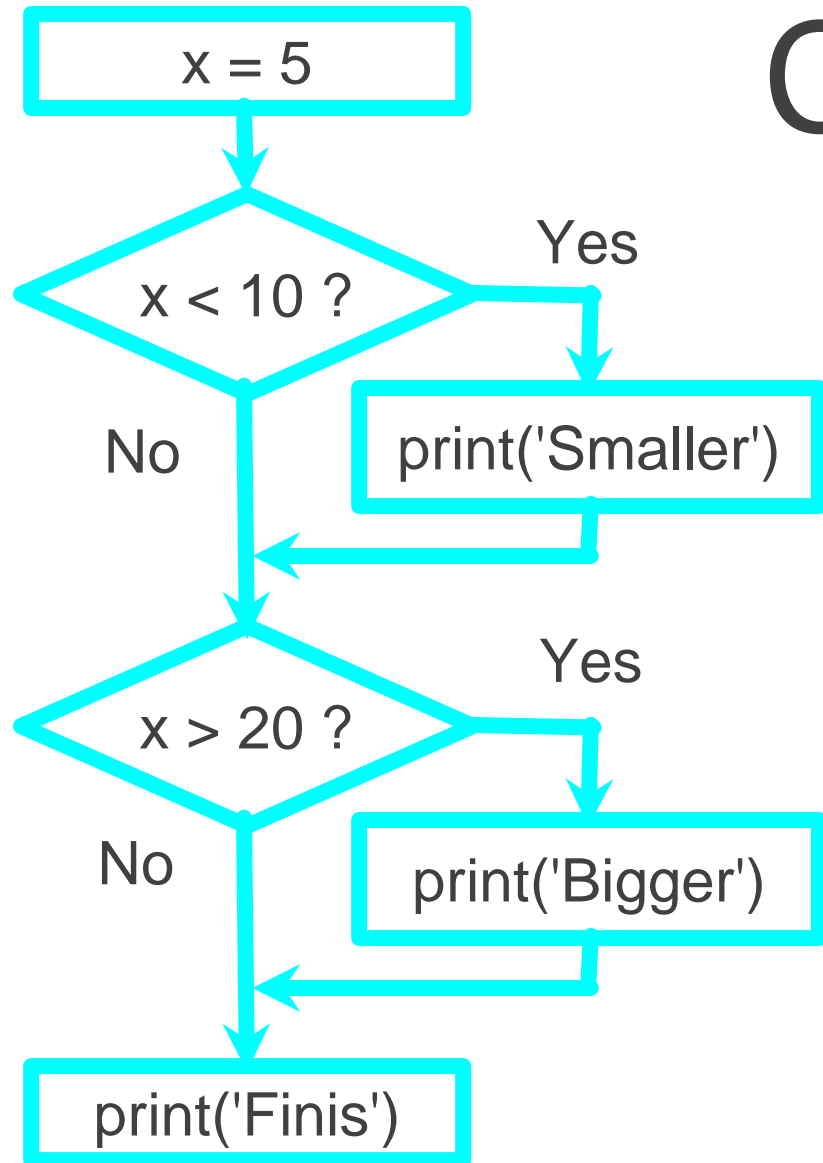


Exercise

- Draw a Flow Chart about what will happen if you go to the library to borrow a book. You search on a computer. If the book exists, it may be available or borrowed. Then what will you do? Also, if the book does not exist, then, what will you do.



Conditional Steps



Program:

```
x = 5
if x < 10:
    print('Smaller')
if x > 20:
    print('Bigger')

print('Finis')
```

Output:

Smaller
Finis



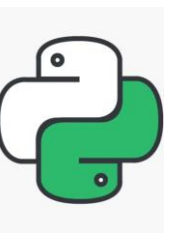
Boolean

LECTURE 2



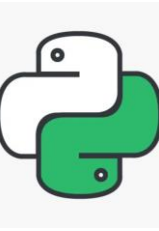
Boolean Value





Boolean Value

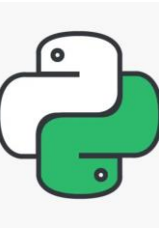
- Boolean expressions ask a question and produce a Yes or No result which we use to control program flow
- Boolean expressions using comparison operators evaluate to True / False or Yes / No
- Comparison operators look at variables but do not change the variables



Comparison Operators

Python	Meaning
<	Less than
<=	Less than or Equal to
==	Equal to
>=	Greater than or Equal to
>	Greater than
!=	Not equal

Remember: “=” is used for assignment.



Comparison Operators

```
x = 5
if x == 5 :
    print('Equals 5')
if x > 4 :
    print('Greater than 4')
if x >= 5 :
    print('Greater than or Equals 5')
if x < 6 : print('Less than 6')
if x <= 5 :
    print('Less than or Equals 5')
if x != 6 :
    print('Not equal 6')
```

Equals 5

Greater than 4

Greater than or Equals 5

Less than 6

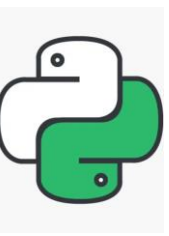
Less than or Equals 5

Not equal 6



Character

LECTURE 3



ASCII

American Standard Code for Information Interexchange

- Americans came up with (8-bit) ASCII representation with English only alphabets as a standard to exchange information.

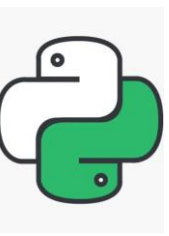
'A' – 65 – 0x41, 'a' – 97 – 0x61

```
>> ord('A')
```

```
65
```

```
>> chr(65)
```

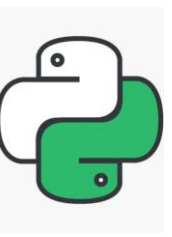
```
A
```



Unicode

A Unified Code Standard for International Symbols

- The rest of the world came up with their unaccented English characters in their own way (messed up). There is a need for international code standard.
- To exchange information in all languages, we got some more requirements:
 - Unique and single rule for the code standard.
 - Adoptable across all machines
 - Efficient storage as much as possible



Unicode

A Unified Code Standard for International Symbols

Python using 16-bit Unicode by default.

Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value
00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	\$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[6B	k	7B	{
0C	FF	1C	FS	2C	,	3C	<	4C	L	5C	\	6C	l	7C	
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

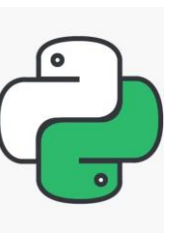
Graphic character symbol	Hexadecimal character value
--------------------------	-----------------------------

0020	0 0030	@ 0040	P 0050	` 0060	p 0070	00A0	° 00B0	À 00C0	Ð 00D0	à 00E0	ð 00F0
! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071	i 00A1	± 00B1	Á 00C1	Ñ 00D1	á 00E1	ñ 00F1
" 0022	2 0032	B 0042	R 0052	b 0062	r 0072	ç 00A2	² 00B2	Â 00C2	Ò 00D2	â 00E2	ò 00F2
# 0023	3 0033	C 0043	S 0053	c 0063	s 0073	£ 00A3	³ 00B3	Ã 00C3	Ó 00D3	ã 00E3	ó 00F3
\$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074	¤ 00A4	´ 00B4	Ä 00C4	Ô 00D4	ä 00E4	ô 00F4
% 0025	5 0035	E 0045	U 0055	e 0065	u 0075	¥ 00A5	µ 00B5	Å 00C5	Õ 00D5	å 00E5	õ 00F5
& 0026	6 0036	F 0046	V 0056	f 0066	v 0076	¦ 00A6	¶ 00B6	Æ 00C6	Ö 00D6	æ 00E6	ö 00F6
' 0027	7 0037	G 0047	W 0057	g 0067	w 0077	§ 00A7	· 00B7	Ç 00C7	× 00D7	ç 00E7	÷ 00F7
(0028	8 0038	H 0048	X 0058	h 0068	x 0078	¨ 00A8	¸ 00B8	È 00C8	Ø 00D8	è 00E8	ø 00F8
) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079	© 00A9	¹ 00B9	É 00C9	Ù 00D9	é 00E9	ù 00F9
* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A	ª 00AA	º 00BA	Ê 00CA	Ú 00DA	ê 00EA	ú 00FA
+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B	« 00AB	» 00BB	Ë 00CB	Û 00DB	ë 00EB	û 00FB
, 002C	< 003C	L 004C	\ 005C	l 006C	007C	¬ 00AC	¼ 00BC	Ì 00CC	Ü 00DC	ì 00EC	ü 00FC
- 002D	= 003D	M 004D] 005D	m 006D	} 007D	- 00AD	½ 00BD	Í 00CD	Ý 00DD	í 00ED	ý 00FD
. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E	® 00AE	¾ 00BE	Î 00CE	Þ 00DE	î 00EE	þ 00FE
/ 002F	? 003F	O 004F	_ 005F	o 006F	007F	¯ 00AF	¿ 00BF	Ï 00CF	ß 00DF	ï 00EF	ÿ 00FF



String

LECTURE 3



Strings

- Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes.
- Creating strings is as simple as assigning a value to a variable. For example:

```
var1 = 'Hello World!'
```

```
var2 = "Python Programming"
```

String Data Type

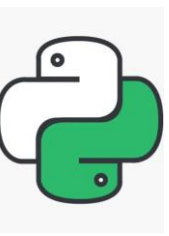
- A string is a sequence of characters
- A string literal uses quotes
'Hello' or "Hello"
- For strings, + means “concatenate”
- When a string contains numbers, it is still a string
- We can convert numbers in a string into a number using int()

```
>>> str1 = "Hello"
>>> str2 = 'there'
>>> bob = str1 + str2
>>> print(bob)
Hellothere
>>> str3 = '123'
>>> str3 = str3 + 1
Traceback (most recent call
last):  File "<stdin>", line 1,
in <module>
TypeError: cannot concatenate
'str' and 'int' objects
>>> x = int(str3) + 1
>>> print(x)
124
>>>
```

Reading and Converting

- We prefer to read data in using strings and then parse and convert the data as we need
- This gives us more control over error situations and/or bad user input
- Input numbers must be converted from strings

```
>>> name = input('Enter:')
Enter:Chuck
>>> print(name)
Chuck
>>> apple = input('Enter:')
Enter:100
>>> x = apple - 10
Traceback (most recent call
last):  File "<stdin>", line 1,
in <module>
TypeError: unsupported operand
type(s) for -: 'str' and 'int'
>>> x = int(apple) - 10
>>> print(x)
90
```

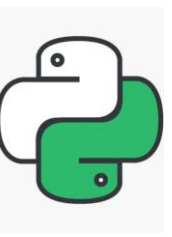


Accessing Values in Strings

- Python does not support a character type; these are treated as strings of length one, thus also considered a substring.
- To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring:

Example:

```
var1 = 'Hello World!'
var2 = "Python Programming"
print ("var1[0]: ", var1[0])
print ("var2[1:5]: ", var2[1:5])
```

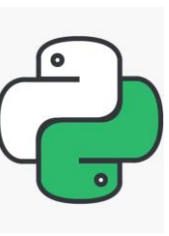


Accessing Values in Strings

- This will produce following result:

`var1[0]: H`

`var2[1:5]: ytho`

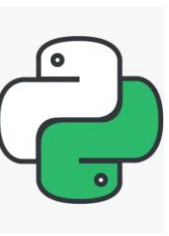


Looking Inside Strings

- We can get at any single character in a string using an index specified in square brackets
- The index value must be an integer and starts at zero
- The index value can be an expression that is computed

b	a	n	a	n	a
0	1	2	3	4	5

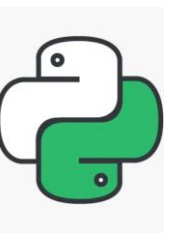
```
>>> fruit = 'banana'
>>> letter = fruit[1]
>>> print(letter)
a
>>> x = 3
>>> w = fruit[x - 1]
>>> print(w)
n
```



A Character Too Far

- You will get a python error if you attempt to index beyond the end of a string
- So be careful when constructing index values and slices

```
>>> zot = 'abc'
>>> print(zot[5])
Traceback (most recent call
last):  File "<stdin>", line
1, in <module>
IndexError: string index out
of range
>>>
```

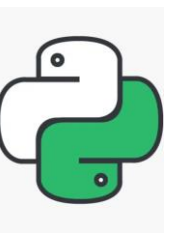


Strings Have Length

- The built-in function `len` gives us the length of a string

b	a	n	a	n	a
0	1	2	3	4	5

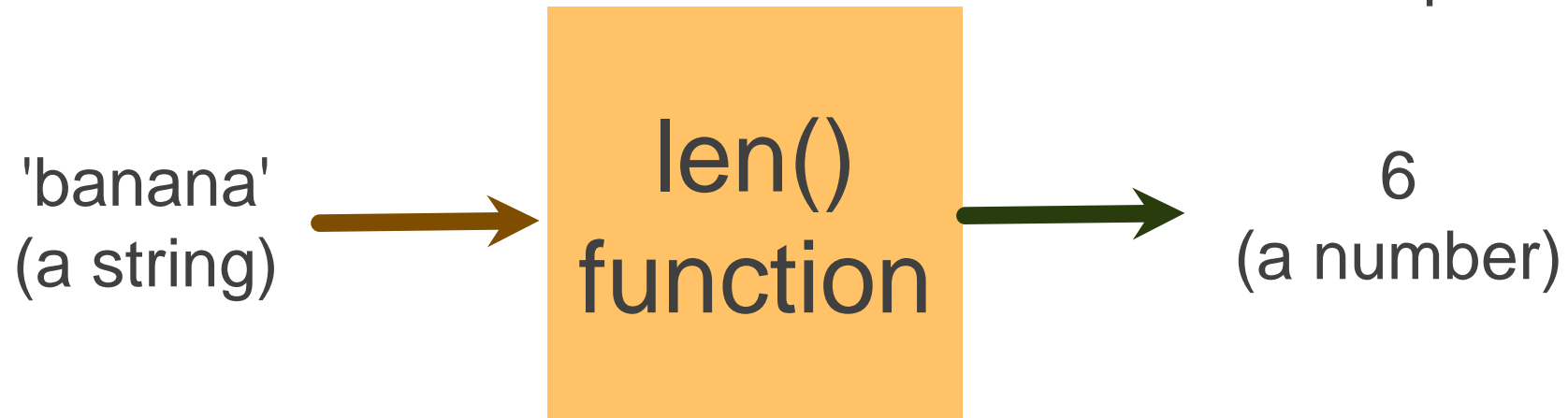
```
>>> fruit = 'banana'
>>> print(len(fruit))
6
```

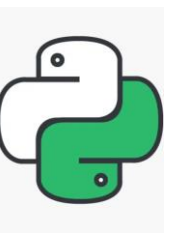


len Function

```
>>> fruit = 'banana'
>>> x = len(fruit)
>>> print(x)
6
```

A function is some stored code that we use. A function takes some input and produces an output.





len Function

```
>>> fruit = 'banana'
>>> x = len(fruit)
>>> print(x)
6
```

A function is some stored code that we use. A function takes some input and produces an output.

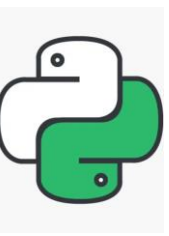
'banana'
(a string)



```
def len(inp):
    blah
    blah
    for x in y:
        blah
        blah
```



6
(a number)



Updating Strings

- You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether.

Example:

```
var1 = 'Hello World!'
print "Updated String :- ", var1[:6] + 'Python'
```

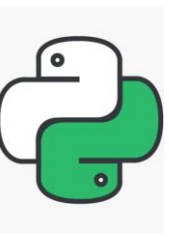
- This will produce following result:
Updated String :- Hello Python

Escape Characters

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x
\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

String Special Operators: Assume string variable a holds 'Hello' and variable b holds 'Python' then:

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppress actual meaning of Escape characters.	print r'\n' prints \n and print R'\n' prints \n
%	Format - Performs String formatting	See at next section

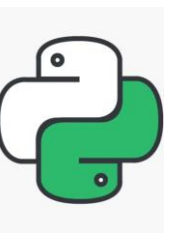


Looping Through Strings

- Using a while statement, an iteration variable, and the len function, we can construct a loop to look at each of the letters in a string individually

```
fruit = 'banana'
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(index, letter)
    index = index + 1
```

0 b
1 a
2 n
3 a
4 n
5 a

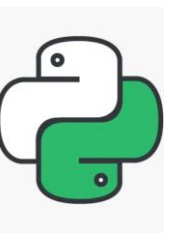


Looping Through Strings

- A definite loop using a for statement is much more elegant
- The iteration variable is completely taken care of by the for loop

```
fruit = 'banana'
for letter in fruit:
    print(letter)
```

b
a
n
a
n
a



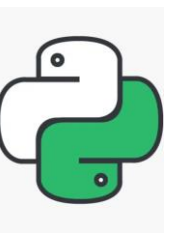
Looping Through Strings

- A definite loop using a for statement is much more elegant
- The iteration variable is completely taken care of by the for loop

```
fruit = 'banana'
for letter in fruit :
    print(letter)
```

b
a
n
a
n
a

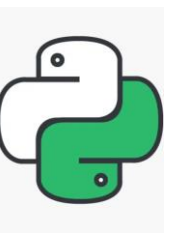
```
index = 0
while index < len(fruit) :
    letter = fruit[index]
    print(letter)
    index = index + 1
```



Looping and Counting

- This is a simple loop that loops through each letter in a string and counts the number of times the loop encounters the 'a' character


```
word = 'banana'
count = 0
for letter in word :
    if letter == 'a' :
        count = count + 1
print(count)
```



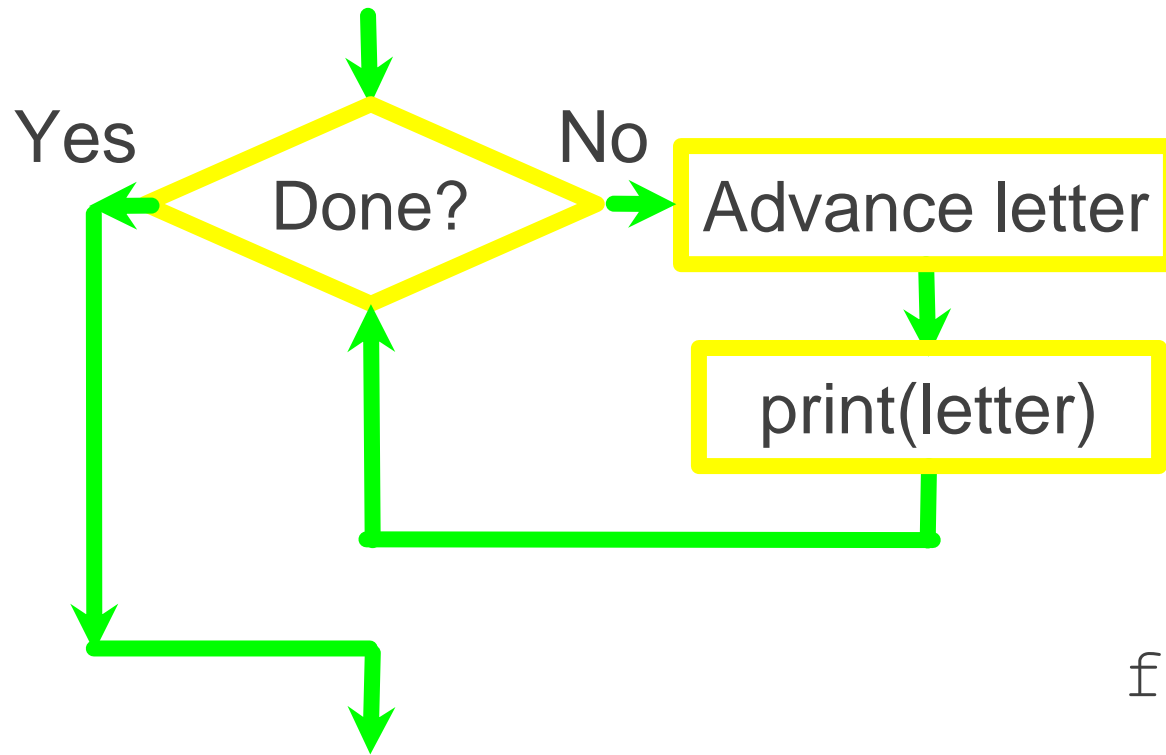
Looking Deeper into in

- The iteration variable “iterates” through the sequence (ordered set)
- The block (body) of code is executed once for each value in the sequence
- The iteration variable moves through all of the values in the sequence

Iteration variable Six-character string



```
for letter in 'banana' :  
    print(letter)
```



```
for letter in 'banana' :  
    print(letter)
```

The iteration variable “iterates” through the string and the block (body) of code is executed once for each value in the sequence



Formatted Print

```
>> print("%.4f" % 3.1415926)
```

```
3.1415
```

```
>> print("%d, %d" % (3, 4))
```

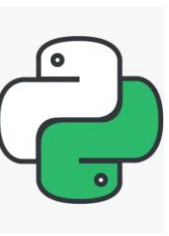
```
3, 4
```

String Formatting Operator:

Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')
%E	exponential notation (with UPPERcase 'E')
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

Other supported symbols and functionality are listed in the following table:

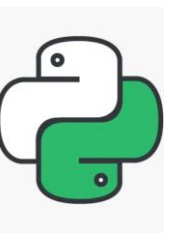
Symbol	Functionality
*	argument specifies width or precision
-	left justification
+	display the sign
<sp>	leave a blank space before a positive number
#	add the octal leading zero ('0') or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
0	pad from left with zeros (instead of spaces)
%	'%%' leaves you with a single literal '%'
(var)	mapping variable (dictionary arguments)
m.n.	m is the minimum total width and n is the number of digits to display after the decimal point (if appl.)



Triple Quotes:

- Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINEs, TABs, and any other special characters.
- The syntax for triple quotes consists of three consecutive **single or double** quotes.

```
para_str = """this is a long string that is made up of several
lines and non-printable characters such as TAB ( \t ) and they
will show up that way when displayed. NEWLINEs within the
string, whether explicitly given like this within the brackets
[ \n ], or just a NEWLINE within the variable assignment will
also show up. """
print(para_str)
```



Unicode String

- Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters, including special characters from most languages in the world. I'll restrict my treatment of Unicode strings to the following:

```
print(u'Hello, world!')
```

This would print following result:

```
Hello, world!
```



Advanced String Operations

ACTIVITY

Useful string functions & methods

Name	Purpose
<code>len(s)</code>	Calculate the length of the string <code>s</code>
<code>+</code>	Add two strings together
<code>*</code>	Repeat a string
<code>s.find(x)</code>	Find the first position of <code>x</code> in the string <code>s</code>
<code>s.count(x)</code>	Count the number of times <code>x</code> is in the string <code>s</code>
<code>s.upper()</code> <code>s.lower()</code>	Return a new string that is all uppercase or lowercase
<code>s.replace(x, y)</code>	Return a new string that has replaced the substring <code>x</code> with the new substring <code>y</code>
<code>s.strip()</code>	Return a new string with whitespace stripped from the ends
<code>s.format()</code>	Format a string's contents

Slicing Strings

M	o	n	t	y		P	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11

- We can also look at any continuous section of a string using a colon operator
The second number is one beyond the end of the slice - “up to but not including”
- If the second number is beyond the end of the string, it stops at the end

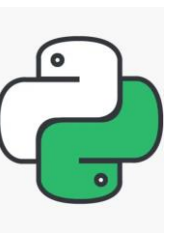
```
>>> s = 'Monty Python'  
>>> print(s[0:4])  
Mont  
>>> print(s[6:7])  
P  
>>> print(s[6:20])  
Python
```

Slicing Strings

M	o	n	t	y		P	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11

- If we leave off the first number or the last number of the slice, it is assumed to be the beginning or end of the string respectively

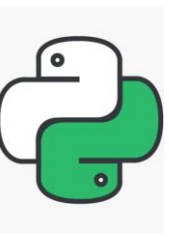
```
>>> s = 'Monty Python'
>>> print(s[:2])
Mo
>>> print(s[8:])
thon
>>> print(s[:])
Monty Python
```



String Concatenation

- When the + operator is applied to strings, it means “concatenation”

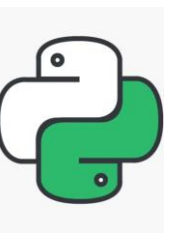
```
>>> a = 'Hello'
>>> b = a + 'There'
>>> print(b)
HelloThere
>>> c = a + ' ' + 'There'
>>> print(c)
Hello There
>>>
```

Using in as a Logical Operator

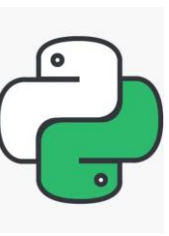
- The in keyword can also be used to check to see if one string is “in” another string
- The in expression is a logical expression that returns True or False and can be used in an if statement

```
>>> fruit = 'banana'
>>> 'n' in fruit
True
>>> 'm' in fruit
False
>>> 'nan' in fruit
True
>>> if 'a' in fruit :
...     print('Found it!')
...
Found it!
>>>
```



String Comparison

```
if word == 'banana':  
    print('All right, bananas.')  
if word < 'banana':  
    print('Your word, ' + word + ', comes before banana.')elif word > 'banana':  
    print('Your word, ' + word + ', comes after banana.')else:  
    print('All right, bananas.')
```



String Library

- Python has a number of string functions which are in the string library
- These functions are already built into every string - we invoke them by appending the function to the string variable
- These functions do not modify the original string, instead they return a new string that has been altered

```
>>> greet = 'Hello Bob'
>>> zap = greet.lower()
>>> print(zap)
hello bob
>>> print(greet)
Hello Bob
>>> print('Hi There'.lower())
hi there
>>>
```

```
>>> stuff = 'Hello world'
>>> type(stuff)
<class 'str'>
>>> dir(stuff)
['capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
```

<https://docs.python.org/3/library/stdtypes.html#string-methods>

str.replace(*old*, *new*[, *count*])

Return a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

str.rfind(*sub*[, *start*[, *end*]])

Return the highest index in the string where substring *sub* is found, such that *sub* is contained within `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` on failure.

str.rindex(*sub*[, *start*[, *end*]])

Like `rfind()` but raises `ValueError` when the substring *sub* is not found.

str.rjust(*width*[, *fillchar*])

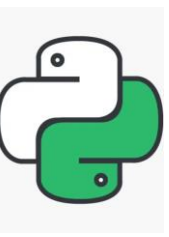
Return the string right justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

str.rpartition(*sep*)

Split the string at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

str.rsplit(*sep=None*, *maxsplit=-1*)**

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done, the *rightmost* ones. If *sep* is not specified or `None`, any whitespace string is a separator. Except for splitting from the right, `rsplit()` behaves like `split()` which is described in detail below.



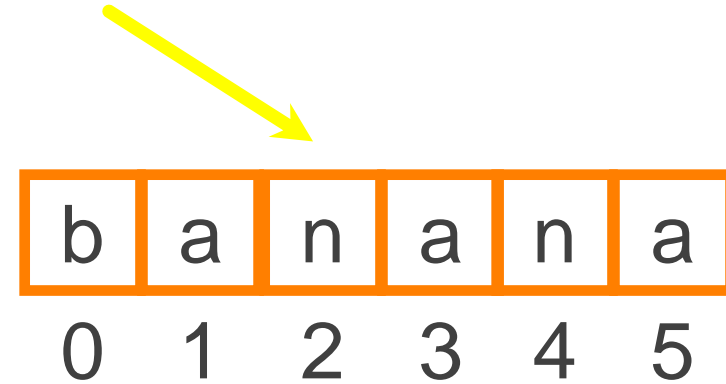
String Library

```
str.capitalize()  
str.center(width[, fillchar])  
str.endswith(suffix[, start[, end]])  
str.find(sub[, start[, end]])  
str.lstrip([chars])
```

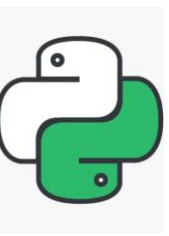
```
str.replace(old, new[, count])  
str.lower()  
str.rstrip([chars])  
str.strip([chars])  
str.upper()
```

Searching a String

- We use the find() function to search for a substring within another string
- find() finds the first occurrence of the substring
- If the substring is not found, find() returns -1
- Remember that string position starts at zero



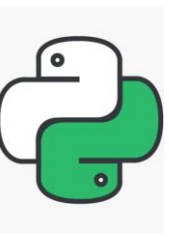
```
>>> fruit = 'banana'
>>> pos = fruit.find('na')
>>> print(pos)
2
>>> aa = fruit.find('z')
>>> print(aa)
-1
```



Making everything UPPER CASE

- You can make a copy of a string in lower case or upper case
- Often when we are searching for a string using `find()` we first convert the string to lower case so we can search a string regardless of case

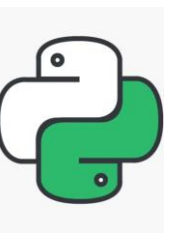
```
>>> greet = 'Hello Bob'
>>> nnn = greet.upper()
>>> print(nnn)
HELLO BOB
>>> www = greet.lower()
>>> print(www)
hello bob
>>>
```

Search and Replace

- The `replace()` function is like a “search and replace” operation in a word processor
- It replaces all occurrences of the search string with the replacement string

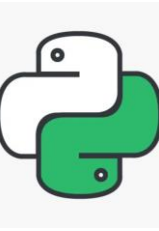
```
>>> greet = 'Hello Bob'
>>> nstr = greet.replace('Bob', 'Jane')
>>> print(nstr)
Hello Jane
>>> nstr = greet.replace('o', 'X')
>>> print(nstr)
HellX BXb
>>>
```



Stripping Whitespace

- Sometimes we want to take a string and remove whitespace at the beginning and/or end
- `lstrip()` and `rstrip()` remove whitespace at the left or right
- `strip()` removes both beginning and ending whitespace

```
>>> greet = '    Hello Bob    '  
>>> greet.lstrip()  
'Hello Bob '  
>>> greet.rstrip()  
'    Hello Bob'  
>>> greet.strip()  
'Hello Bob'  
>>>
```



Prefixes

```
>>> line = 'Please have a nice day'
>>> line.startswith('Please')
True
>>> line.startswith('p')
False
```

Parsing and Extracting

21

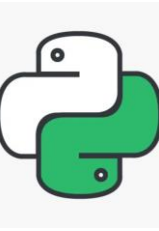


31



From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print(atpos)
21
>>> sppos = data.find(' ', atpos)
>>> print(sppos)
31
>>> host = data[atpos+1 : sppos]
>>> print(host)
uct.ac.za
```



Two Kinds of Strings

```
Python 2.7.10
>>> x = '이광춘'
>>> type(x)
<type 'str'>
>>> x = u'이광춘'
>>> type(x)
<type 'unicode'>
>>>
```

```
Python 3.5.1
>>> x = '이광춘'
>>> type(x)
<class 'str'>
>>> x = u'이광춘'
>>> type(x)
<class 'str'>
>>>
```

In Python 3, all strings are Unicode



One-way Selection

LECTURE 4

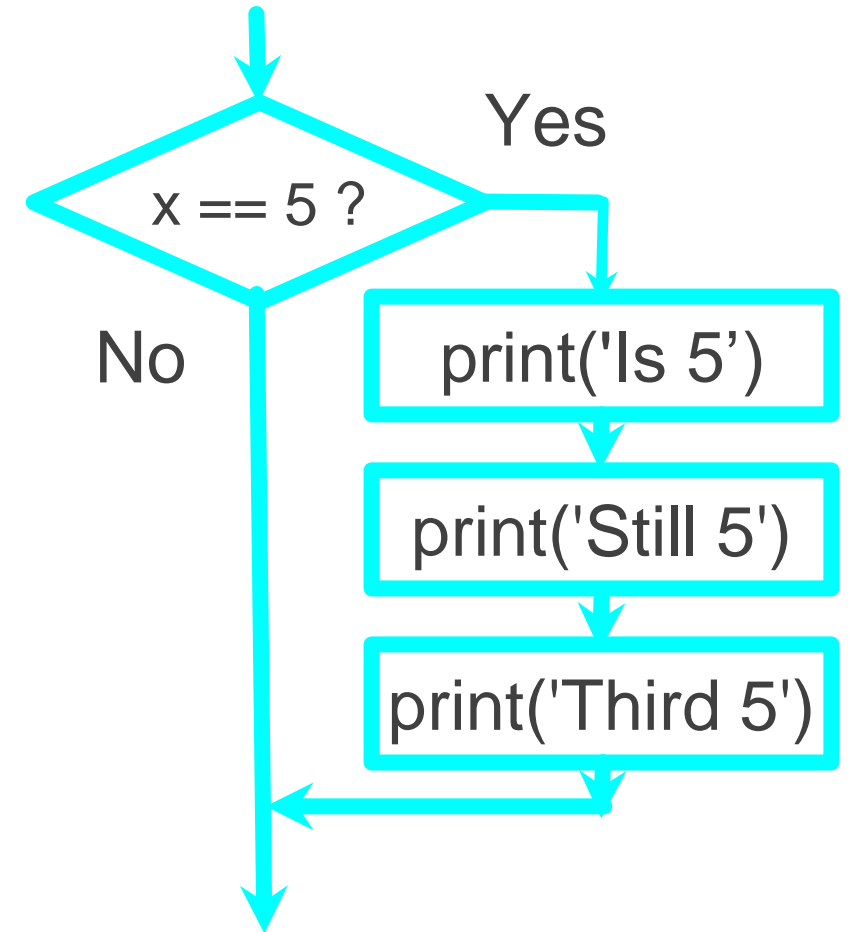
One-Way Decisions

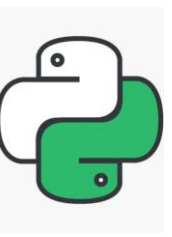
```
x = 5
print('Before 5')
if x == 5 :
    print('Is 5')
    print('Is Still 5')
    print('Third 5')
print('Afterwards 5')
print('Before 6')
if x == 6 :
    print('Is 6')
    print('Is Still 6')
    print('Third 6')
print('Afterwards 6')
```

Before 5

Is 5
Is Still 5
Third 5
Afterwards 5
Before 6

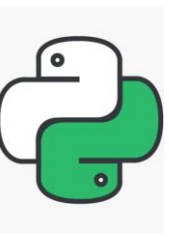
Afterwards 6





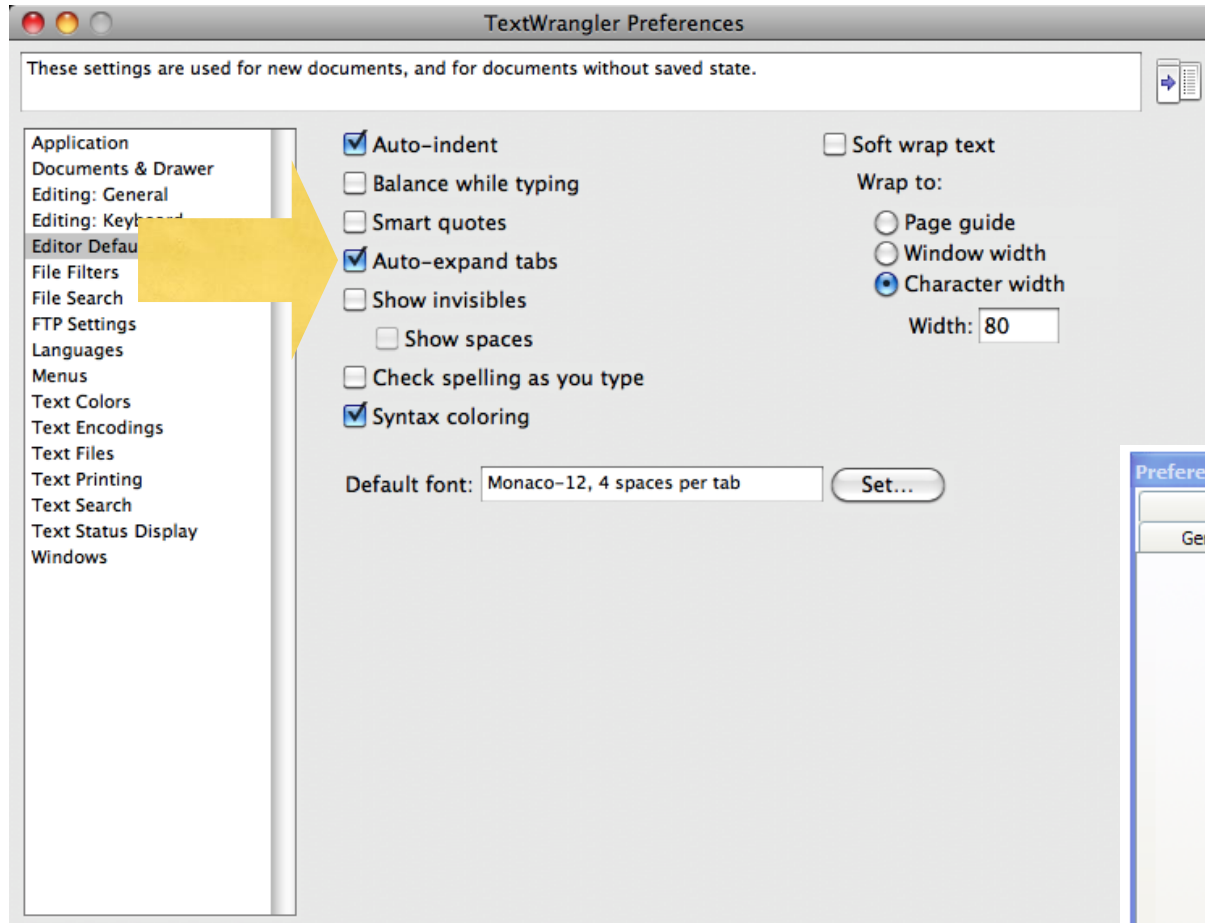
Indentation

- Increase indent after an if statement or for statement (after :)
- Maintain indent to indicate the scope of the block (which lines are affected by the if/for)
- Reduce indent back to the level of the if statement or for statement to indicate the end of the block
- Blank lines are ignored - they do not affect indentation
- Comments on a line by themselves are ignored with regard to indentation

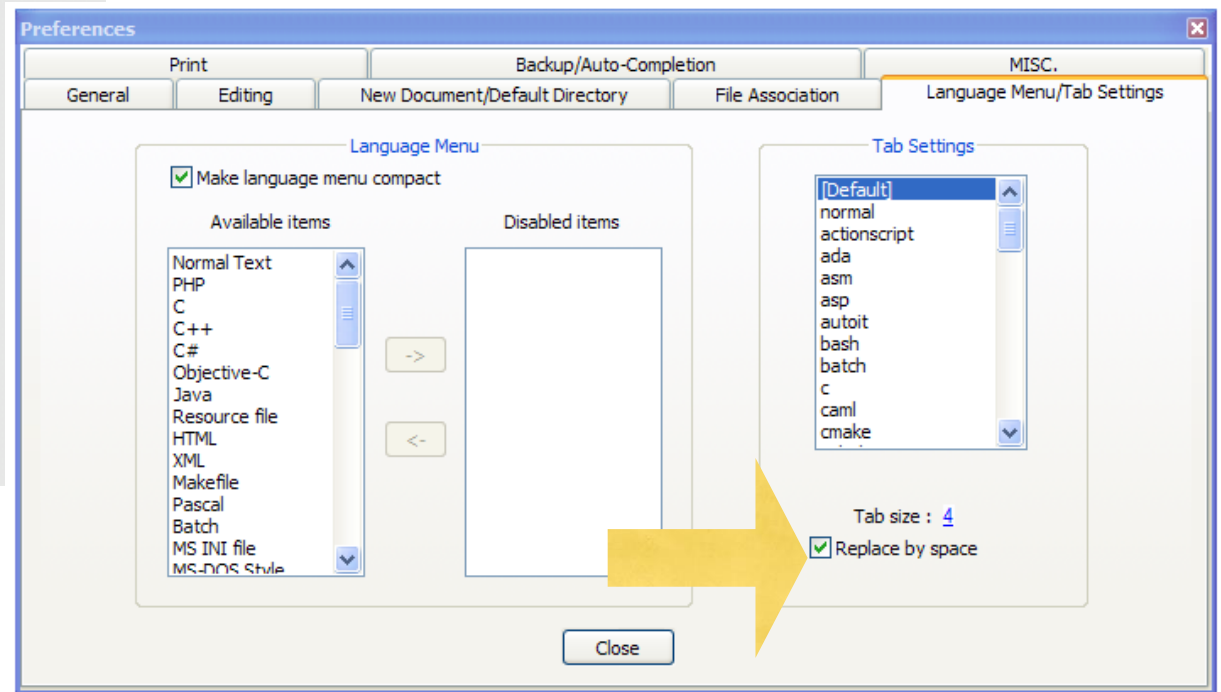


Warning: Turn Off Tabs!!

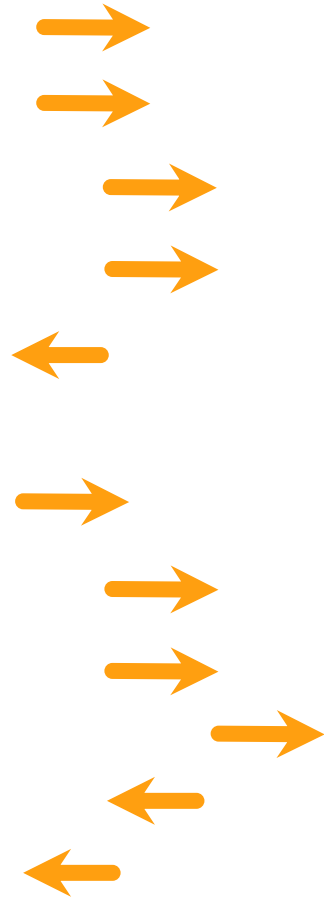
- Atom automatically uses spaces for files with ".py" extension (nice!)
- Most text editors can turn tabs into spaces - make sure to enable this feature
 - - Notepad++: Settings -> Preferences -> Language Menu/Tab Settings
 - - TextWrangler: TextWrangler -> Preferences -> Editor Defaults
- Python cares a *lot* about how far a line is indented. If you mix tabs and spaces, you may get "indentation errors" even if everything looks fine



This will save you
much unnecessary
pain.



increase / maintain after if or for
decrease to indicate end of block



```
x = 5
if x > 2 :
    print('Bigger than 2')
    print('Still bigger')
print('Done with 2')

for i in range(5) :
    print(i)
    if i > 2 :
        print('Bigger than 2')
    print('Done with i', i)
print('All Done')
```

Think About begin/end Blocks

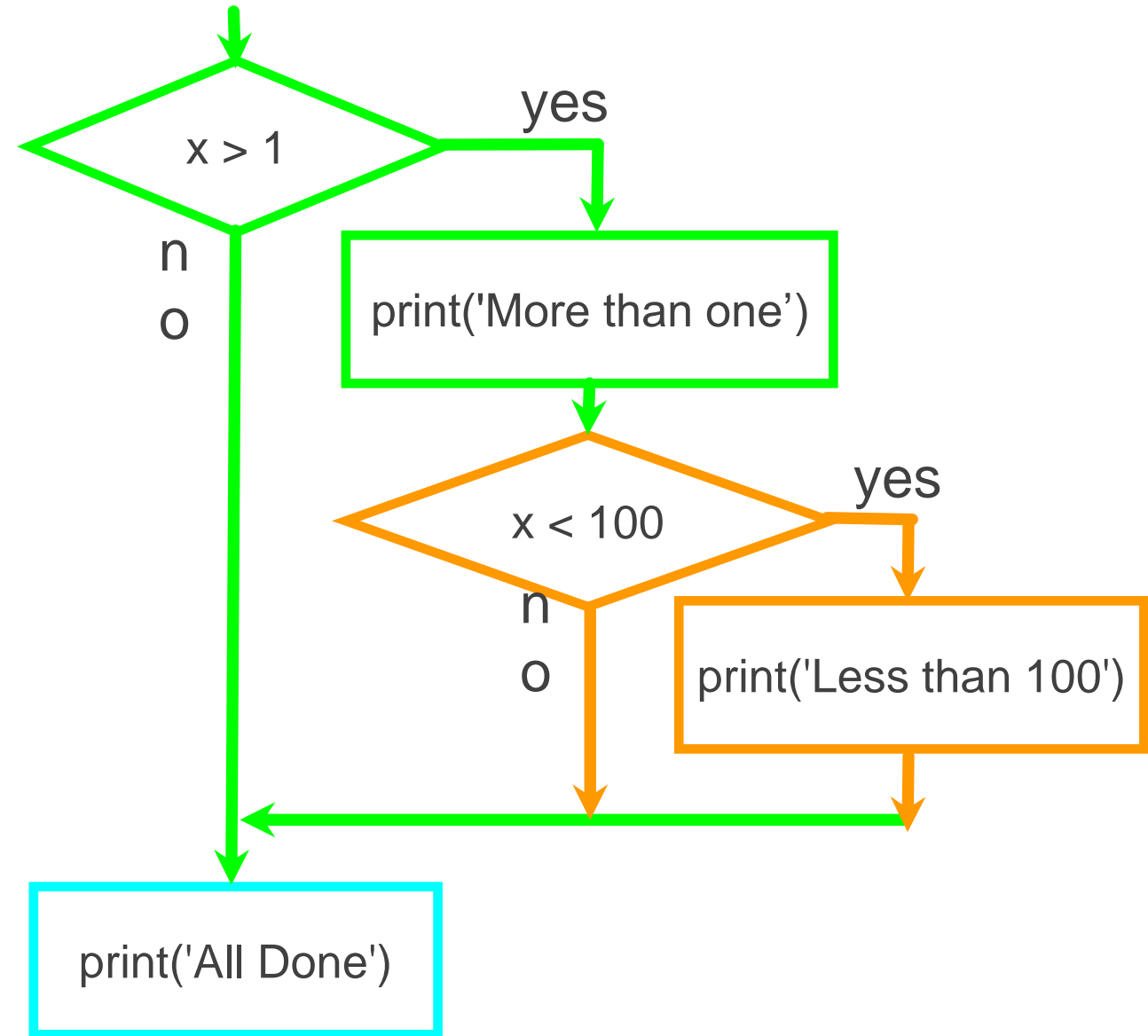
```
x = 5
```

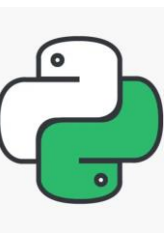
```
if x > 2 :  
    print('Bigger than 2')  
    print('Still bigger')  
print('Done with 2')
```

```
for i in range(5) :  
    print(i)  
    if i > 2 :  
        print('Bigger than 2')  
    print('Done with i', i)  
print('All Done')
```

Nested Decisions

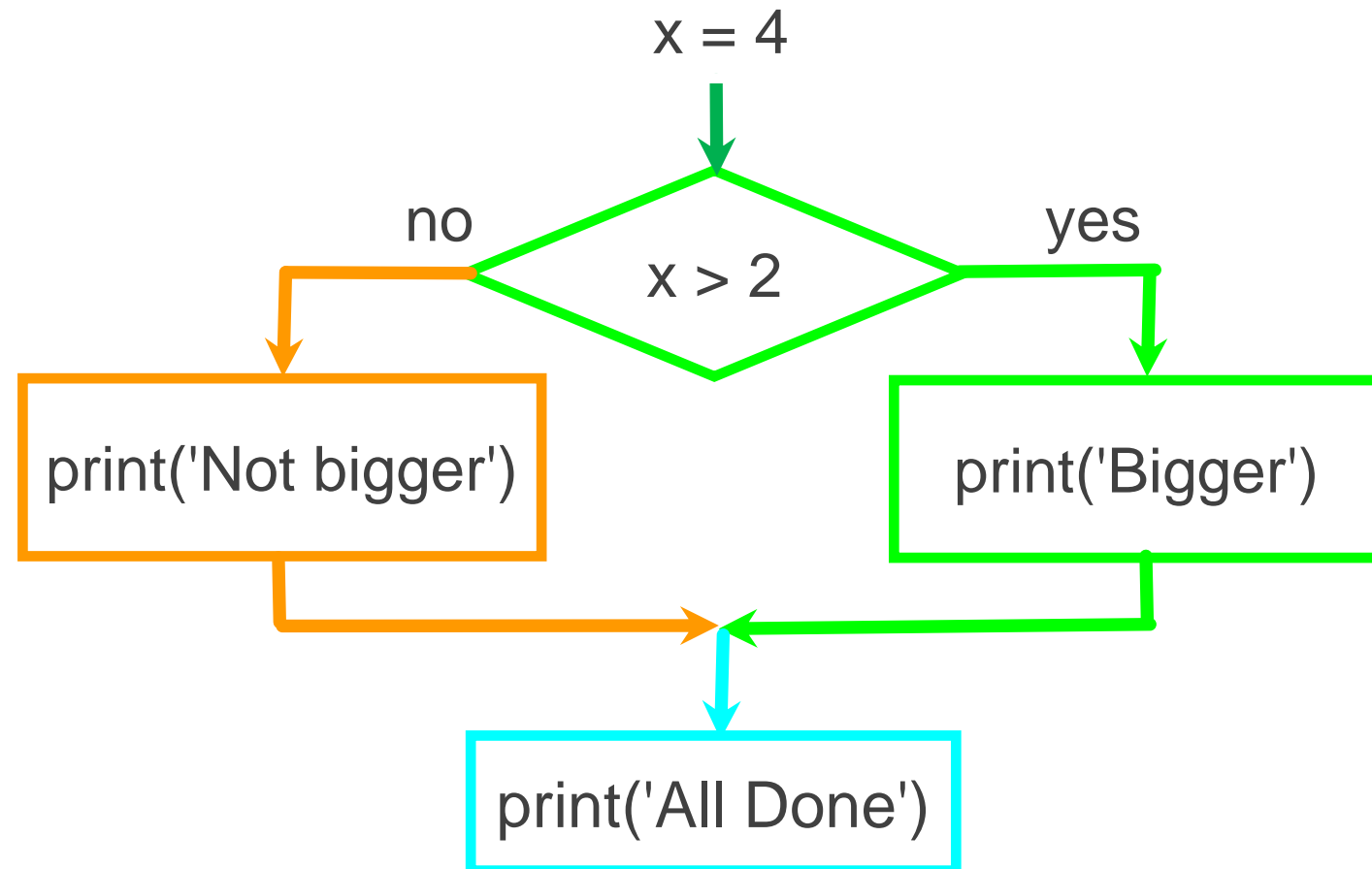
```
x = 42
if x > 1 :
    print('More than one')
    if x < 100 :
        print('Less than 100')
print('All done')
```

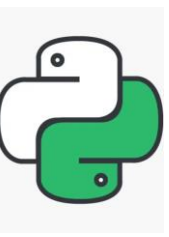




Two-way Decisions

- Sometimes we want to do one thing if a logical expression is true and something else if the expression is false
- It is like a fork in the road - we must choose one or the other path but not both

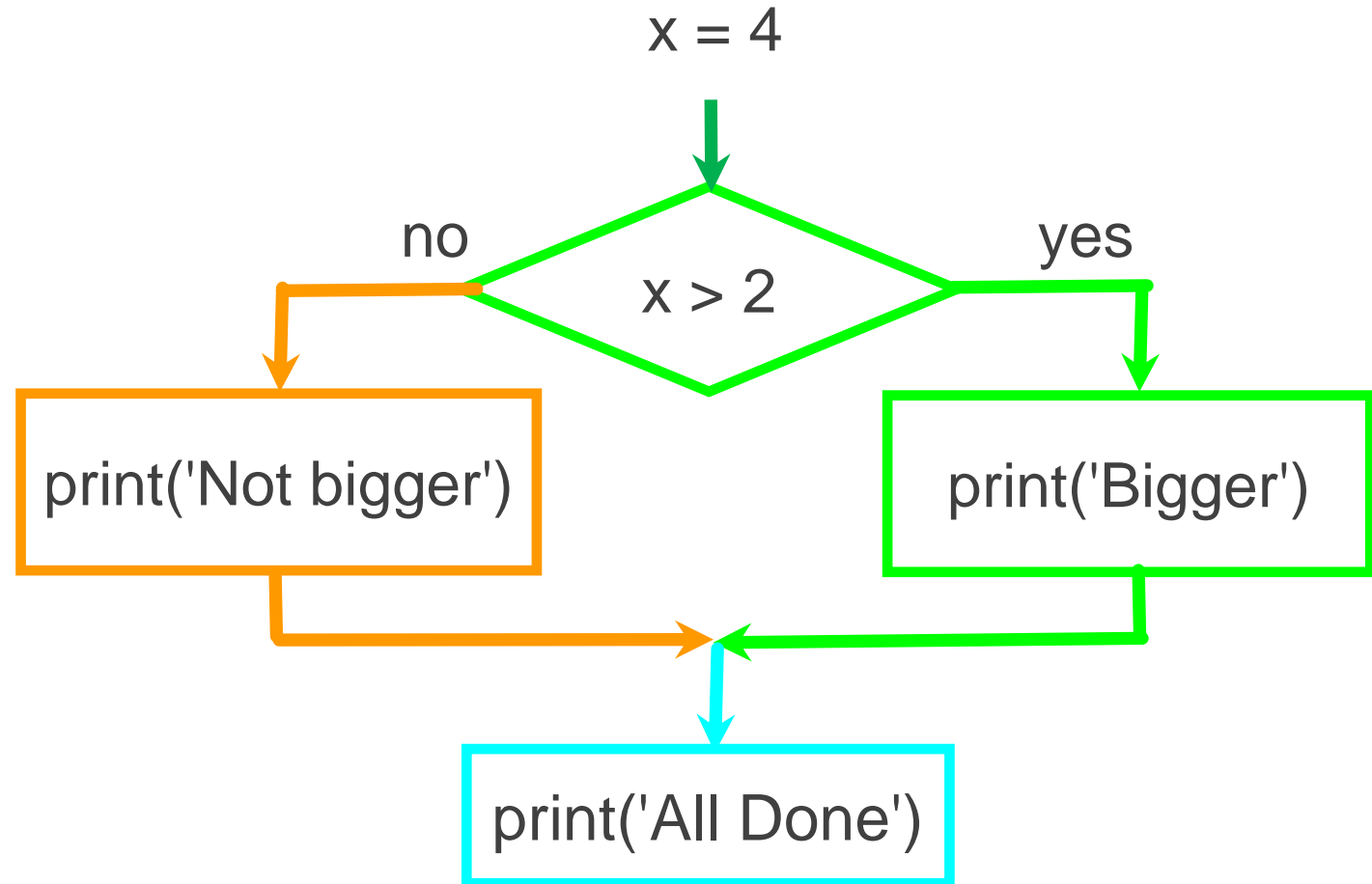




Two-way Decisions with else:

```
x = 4
```

```
if x > 2 :  
    print('Bigger')  
else :  
    print('Smaller')  
  
print('All done')
```



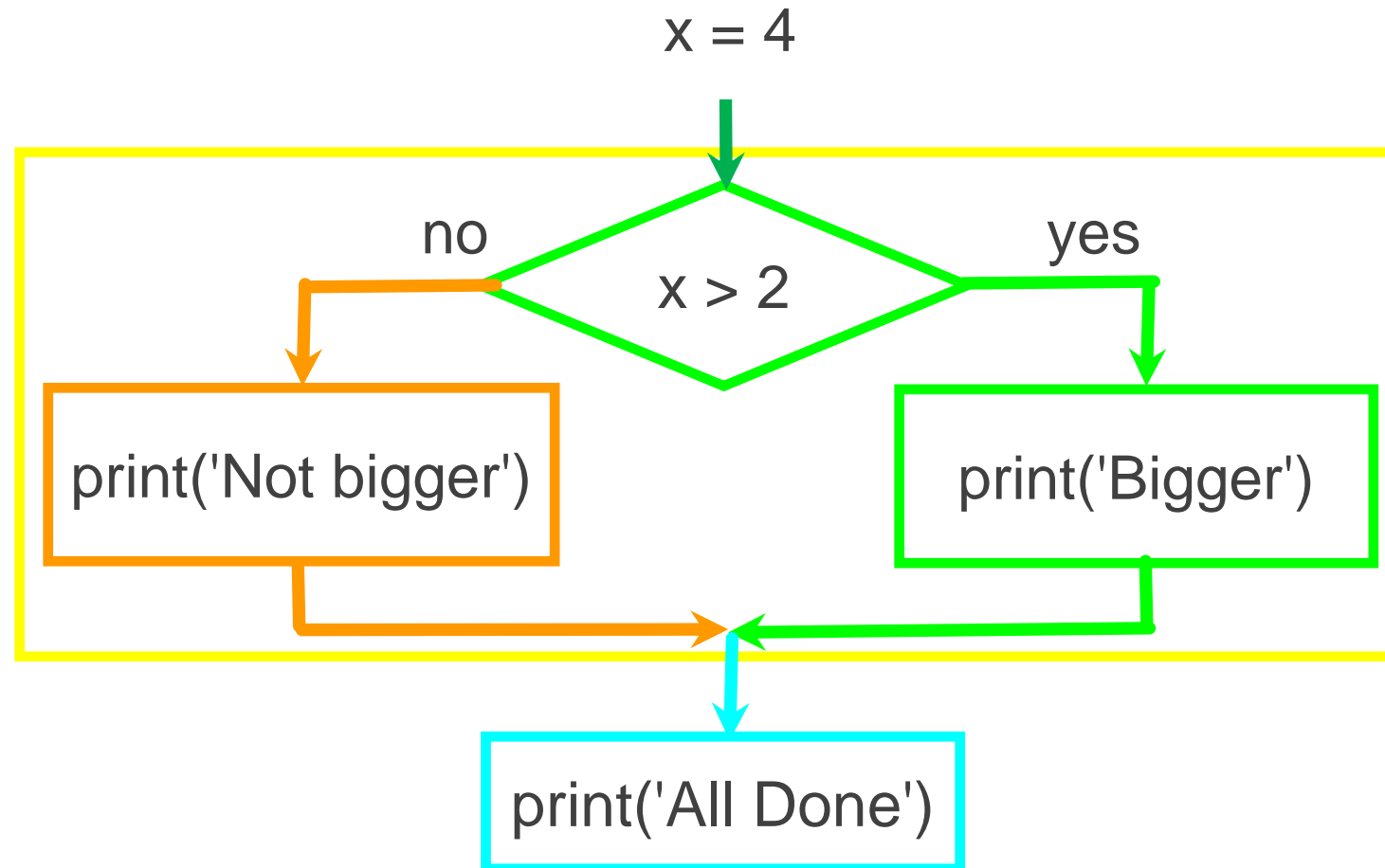


Visualize Blocks

```
x = 4
```

```
if x > 2 :  
    print('Bigger')  
else :  
    print('Smaller')
```

```
print('All done')
```



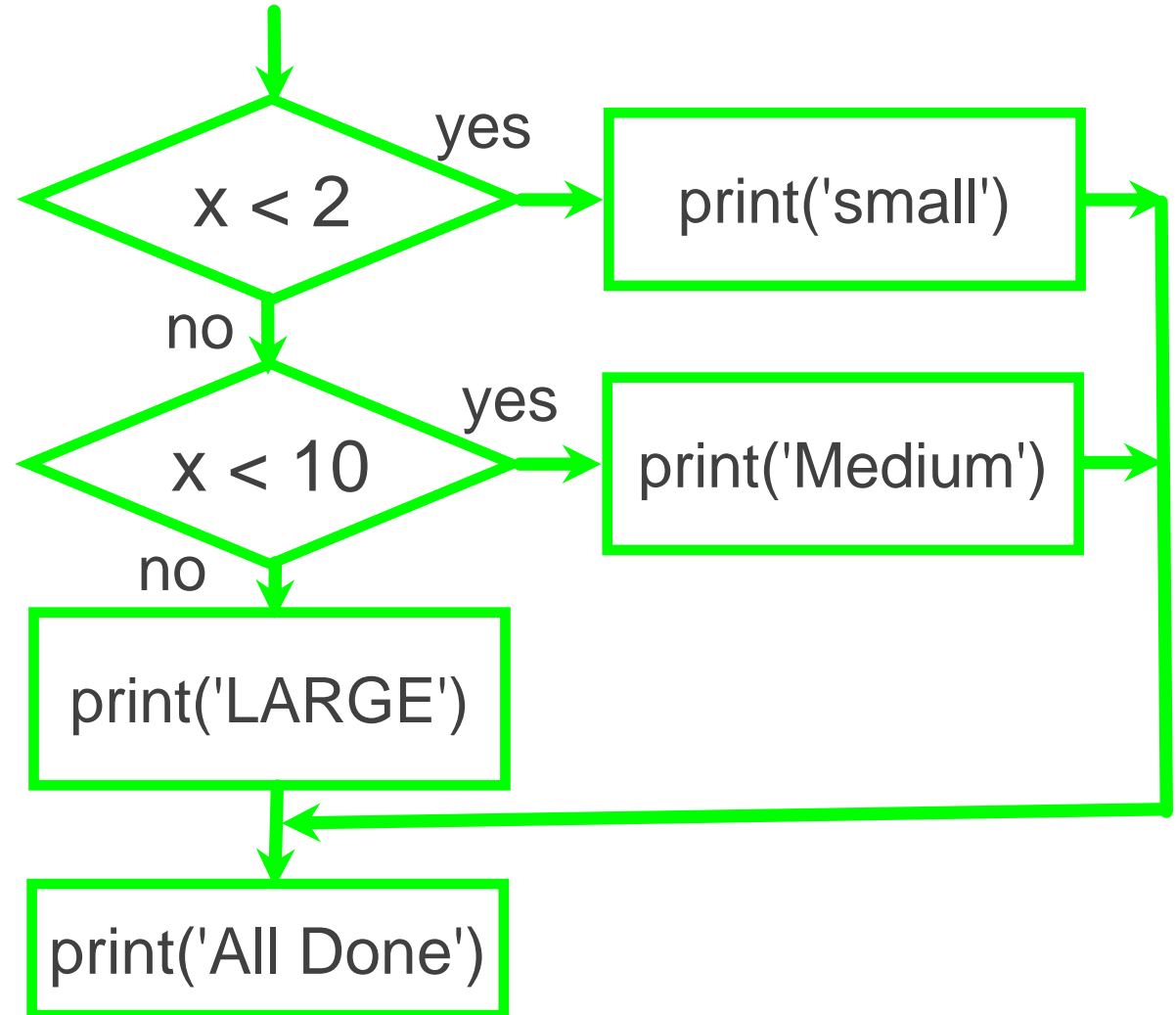


Multiple- Way Selection

LECTURE 4

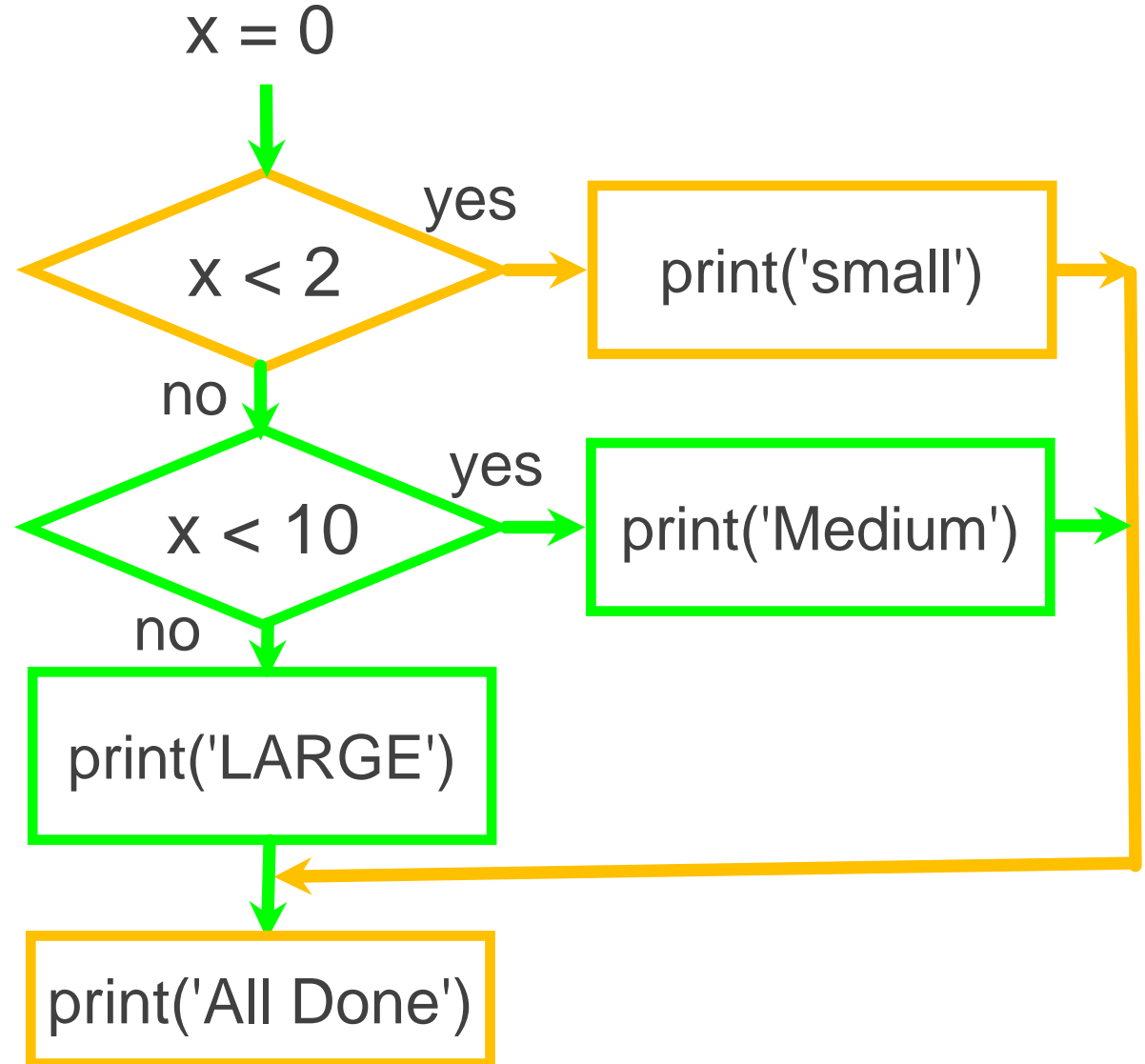
Multi-way

```
if x < 2 :  
    print('small')  
elif x < 10 :  
    print('Medium')  
else :  
    print('LARGE')  
print('All done')
```



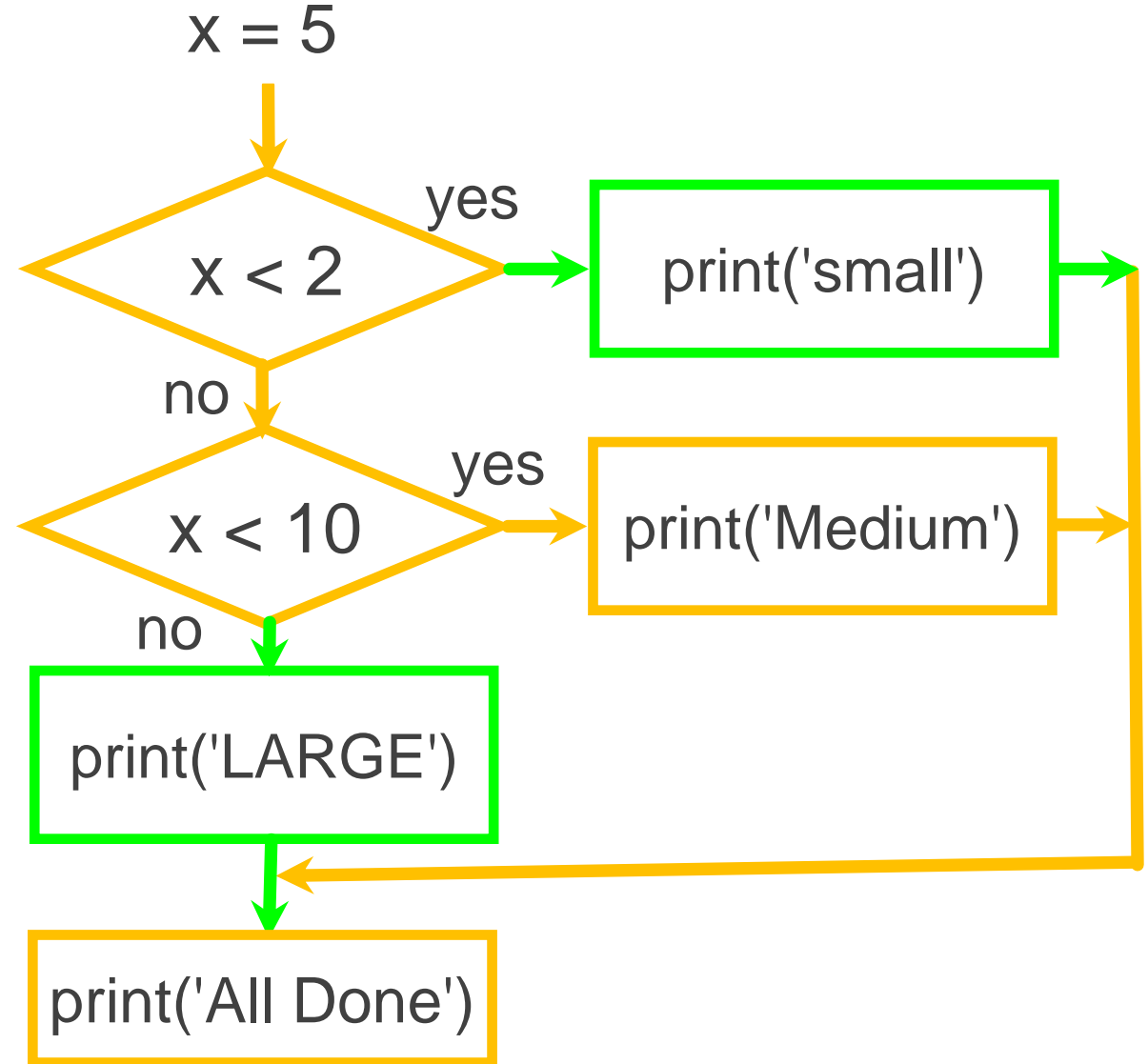
Multi-way

```
x = 0
if x < 2 :
    print('small')
elif x < 10 :
    print('Medium')
else :
    print('LARGE')
print('All done')
```



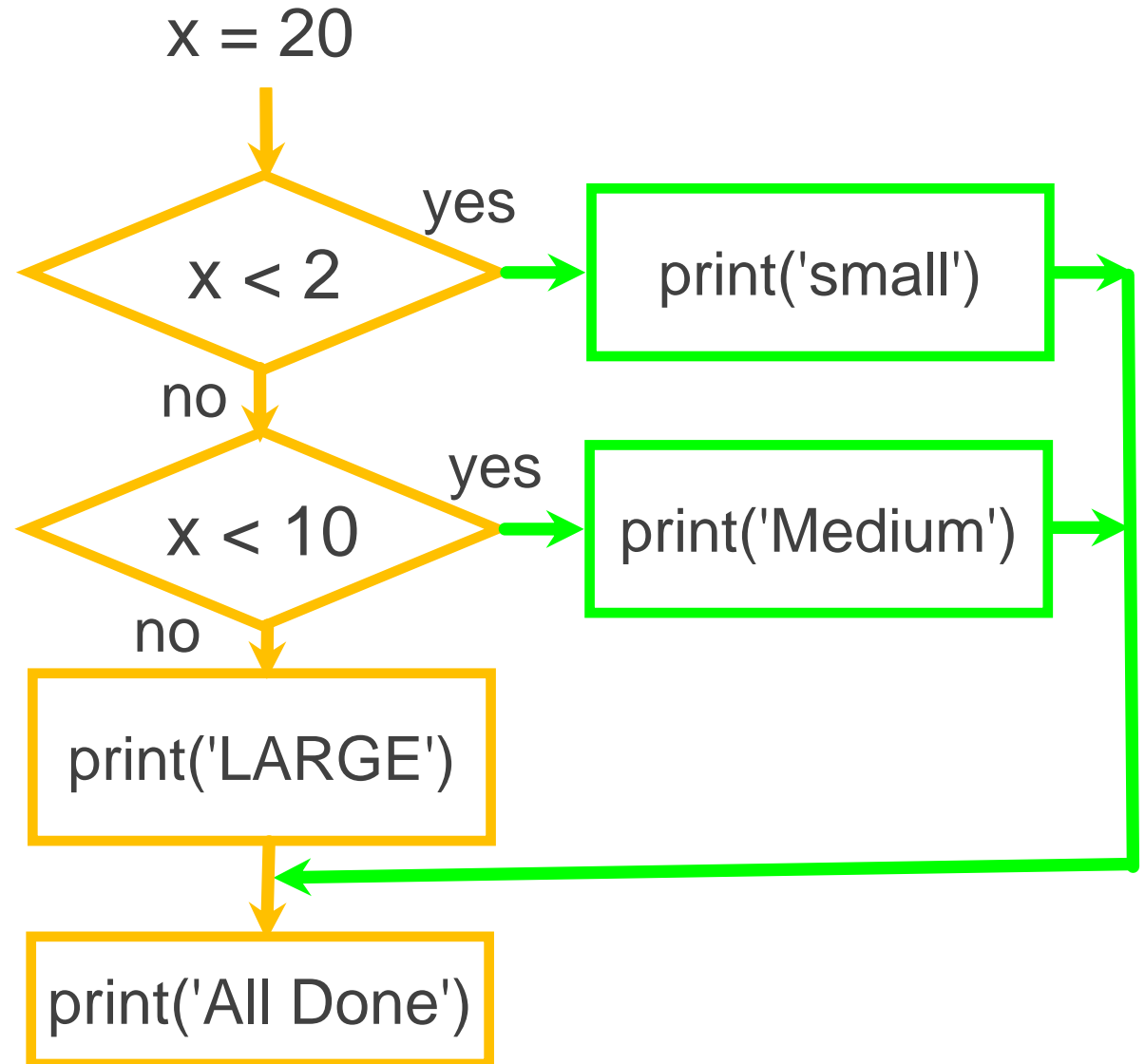
Multi-way

```
x = 5
if x < 2 :
    print('small')
elif x < 10 :
    print('Medium')
else :
    print('LARGE')
print('All done')
```



Multi-way

```
x = 20
if x < 2 :
    print('small')
elif x < 10 :
    print('Medium')
else :
    print('LARGE')
print('All done')
```



Multi-way

```
# No Else
x = 5
if x < 2 :
    print('Small')
elif x < 10 :
    print('Medium')

print('All done')
```

```
if x < 2 :
    print('Small')
elif x < 10 :
    print('Medium')
elif x < 20 :
    print('Big')
elif x < 40 :
    print('Large')
elif x < 100:
    print('Huge')
else :
    print('Ginormous')
```



Multi-way Puzzles

Which will never print regardless of the value for x?

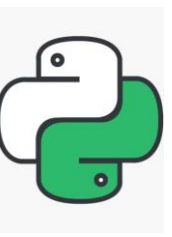
```
if x < 2 :  
    print('Below 2')  
elif x >= 2 :  
    print('Two or more')  
else :  
    print('Something else')
```

```
if x < 2 :  
    print('Below 2')  
elif x < 20 :  
    print('Below 20')  
elif x < 10 :  
    print('Below 10')  
else :  
    print('Something else')
```



Conditional Statement

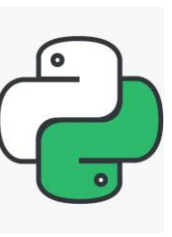
LECTURE 5



Conditional Statement

value_for_true **if** boolean_expression **else** value_for_false

```
a = 3 if x > 4 else 4    # a will get a value of 3 if x > 4,  
                        # otherwise, it will get 4
```



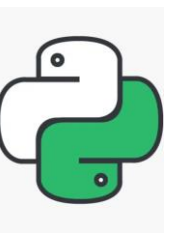
Conditional Statement

- Conditional Statement is often used for the pre-processing of data
- Conditional Statement will make a program shorter.



Lab

SALARY CALCULATION



Exercise

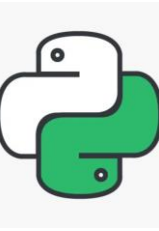
Rewrite your pay computation to give the employee 1.5 times the hourly rate for hours worked above 40 hours.

Enter Hours: 45

Enter Rate: 10

Pay: 475.0

$$475 = 40 * 10 + 5 * 15$$



Exercise

Rewrite your pay program using try and except so that your program handles non-numeric input gracefully.

Enter Hours: 20

Enter Rate: nine

Error, please enter numeric input

Enter Hours: forty

Error, please enter numeric input