Logistics

- Classes and exams
  - No class on 09/05 (labor day)
  - No class on 09/12 (instructor traveling)
  - Makeup class on 10/10? (10/10 – 10/11 is Fall Break)
  - Dec 5th, last class
  - Dec 12th, final exam
- Attending
  - Hybrid -- Zoom links are created and shared
  - Recording automatically, available via blackboard
- Office hour:
  - Mon 1pm – 2pm,
  - Thu 4pm – 5pm (virtual, zoom link shared)
- Please check google calendar for all classes/office hours

Lecture 2. Python Primer – Part 2

Chao Chen

Stony Brook University

Aug 29, 2022

Input

```
>>> name = input('What is your name?\n')
What is your name?
Chuck
>>> print(name)
Chuck
```

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
17
>>> int(speed)
17
>>> int(speed) + 5
22
```

What if user input something incorrect (not a number)?
How to improve?

Comments

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
percentage = (minute * 100) / 60      # percentage of an hour
```

Everything from the **\#** to the end of the line is ignored; it has no effect on the program.

Comments (cont'd)

This comment is redundant with the code and useless:

```
v = 5       # assign 5 to v
```

This comment contains useful information that is not in the code:

```
v = 5       # velocity in meters/second.
```

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a trade-off.

Choosing mnemonic variable names

- Mnemonic -- memory aid
- Mnemonic variable names help us remember why we created the variable in the first place.

```
a = 35.0
b = 12.50
c = a * b
print(c)
```

```
hours = 35.0
rate = 12.50
pay = hours * rate
print(pay)
```

```
x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

Debugging

```
>>> bad name = 5
SyntaxError: invalid syntax
```

- Wrong characters in name (‘ ‘, ‘-’, etc)
- Integers should not be led by digit 0
- Float can be led by digit 0

```
>>> month = 09
  File "<stdin>", line 1
    month = 09
              ^
SyntaxError: invalid token
```

- Wrong name leads to “undefined” error, sometimes hard to find out.

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

- No error message, but this could be an error if you meant 1/(2*pi) stealthy error, hard to find

```
>>> 1.0 / 2.0 * pi
```

Exercises

Exercise 2: Write a program that uses `input` to prompt a user for their name and then welcomes them.

```
Enter your name: Chuck
Hello Chuck
```

Exercise 3: Write a program to prompt the user for hours and rate per hour to compute gross pay.

```
Enter Hours: 35
Enter Rate: 2.75
Pay: 96.25
```

Exercises (cont'd)

Exercise 4: Assume that we execute the following assignment statements:

```
width = 17
height = 12.0
```

For each of the following expressions, write the value of the expression and the type (of the value of the expression).

1. `width//2`
2. `width/2.0`
3. `height/3`
4. `1 + 2 \* 5`

Exercises (cont'd)

Exercise 5: Write a program which prompts the user for a Celsius temperature, convert the temperature to Fahrenheit, and print out the converted temperature.

Comparison operators

- Equality operators

==	equivalent
!=	not equivalent

```
>>> 5 == 5
True
>>> 5 == 6
False
```

- **More Comparison operators:**

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

what if 5<‘hello’ ?

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Expressions and Operators

- Logic operators:

not	unary negation
and	conditional and
or	conditional or

```
x > 0 and x < 10
```

is true only if `x` is greater than 0 *and* less than 10.

`n%2 == 0 or n%3 == 0` is true if *either* of the conditions is true, that is, if the number is divisible by 2 *or* 3.

`not (x > y)` is true if `x > y` is false; that is, if `x` is less than or equal to `y`.

Expressions and Operators

- Logic operators:

not	unary negation
and	conditional and
or	conditional or

Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as "true."

```
>>> 17 and True
True
```

This flexibility can be useful, but there are some subtleties to it that might be confusing. You might want to avoid it until you are sure you know what you are doing.

Conditional Statements

- Check condition, execute if true

```
if x > 0 :
    print('x is positive')
```

- ":" -- start of the block of statements
- Indentation determines the end of the block (demo)
- This type of code block is called *compound statements* (not sure if the name has any use though)
- Sometimes, only a placeholder (pass)

```
if x < 0 :
    pass                # need to handle negative values!
```

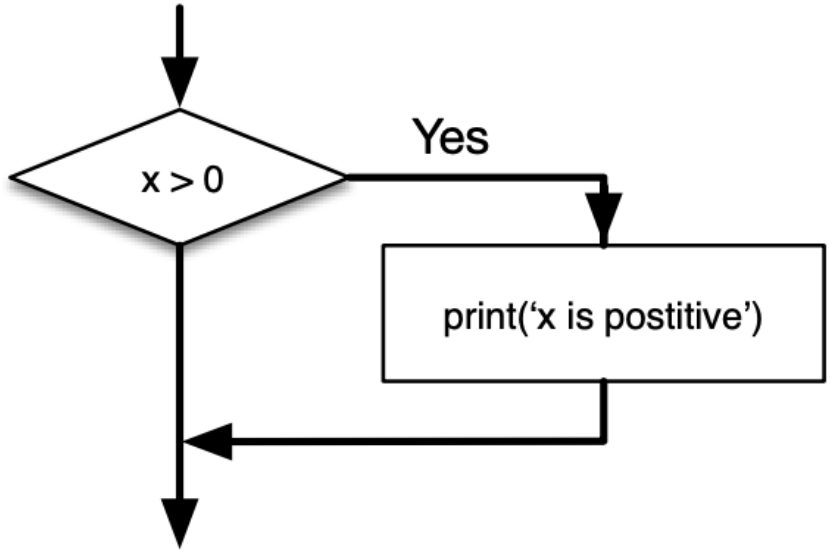


Figure 3.1: If Logic

If statement in the interpreter

- Once typed an if condition, the interpreter would not execute until the whole block is finished
- Indentation determines the end of the if statement

```
>>> x = 3
>>> if x < 10:
...     print('Small')
...
Small
>>>
```

Alternative Execution

- else + ":"

```
if x%2 == 0 :
    print('x is even')
else :
    print('x is odd')
```

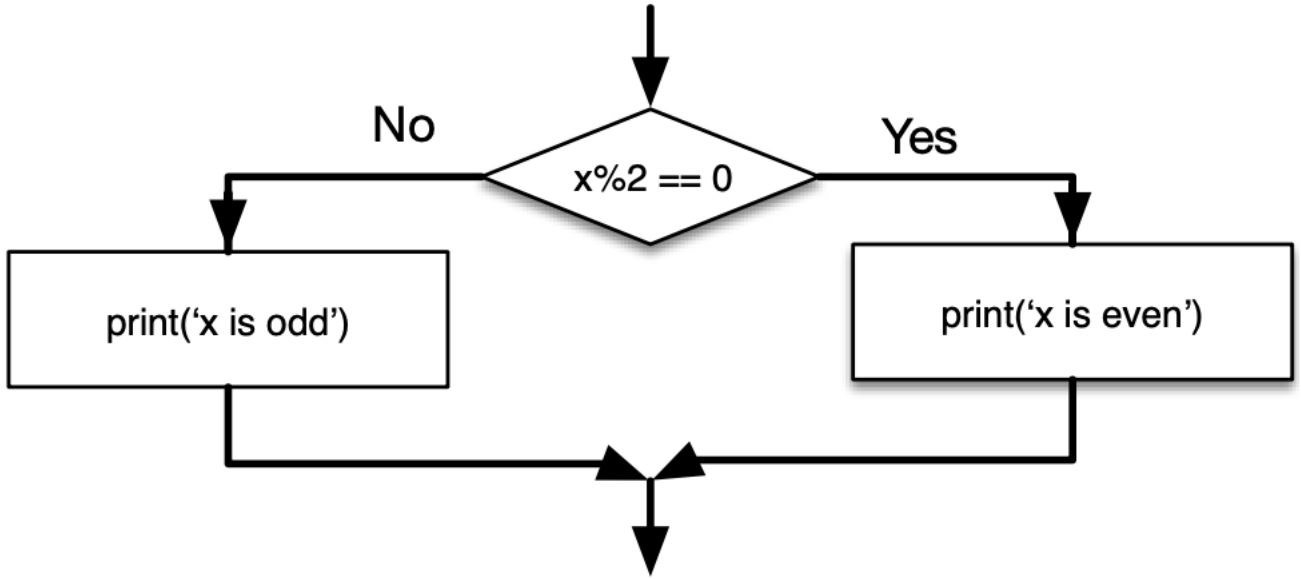


Figure 3.2: If-Then-Else Logic

- Same thing about indentation
- But indentation within else can be different from indentation within if (demo)
- General rule of thumb:
  indentations can be arbitrary, but have to be the same within each block

Chained Conditionals

- elif = else if

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

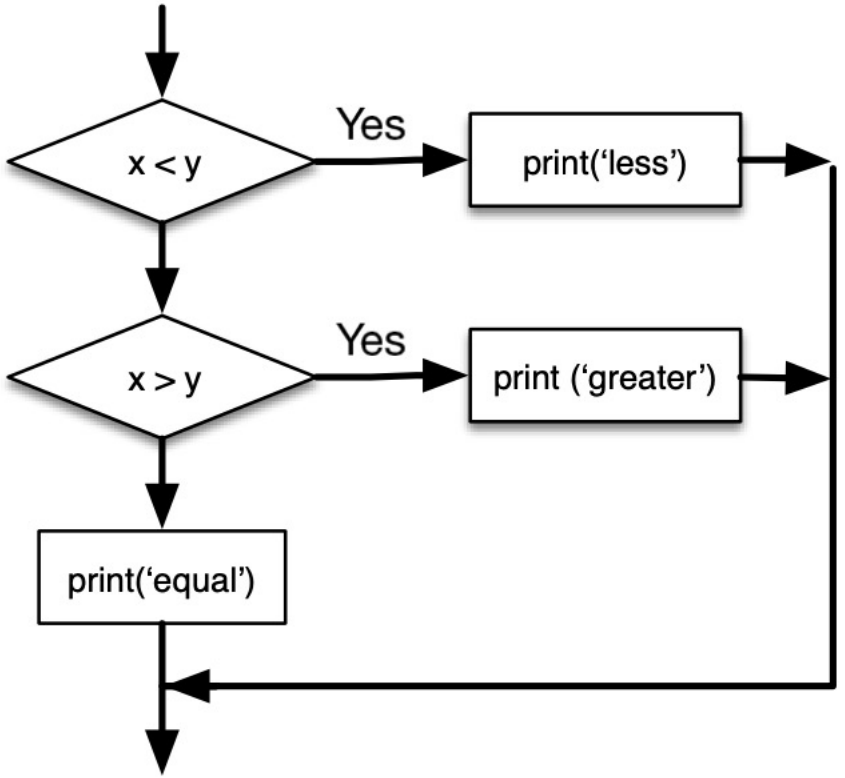


Figure 3.3: If-Then-ElseIf Logic

- Same old same old with ":" and indentation

Chained Conditionals (cont'd)

- It does not have to end with an "else"

```
if choice == 'a':
    print('Bad guess')
elif choice == 'b':
    print('Good guess')
elif choice == 'c':
    print('Close, but not correct')
```

- What if choice == 'd'?

Chained Conditionals (cont'd)

- What if conditions overlap?

```
1  x = 33
2  if x > 20:
3      print("x is bigger than 20")
4  elif x > 10:
5      print("x is bigger than 10 and no greater than 20")
6  else:
7      print("x is no greater than 10")
```

Nested Conditionals

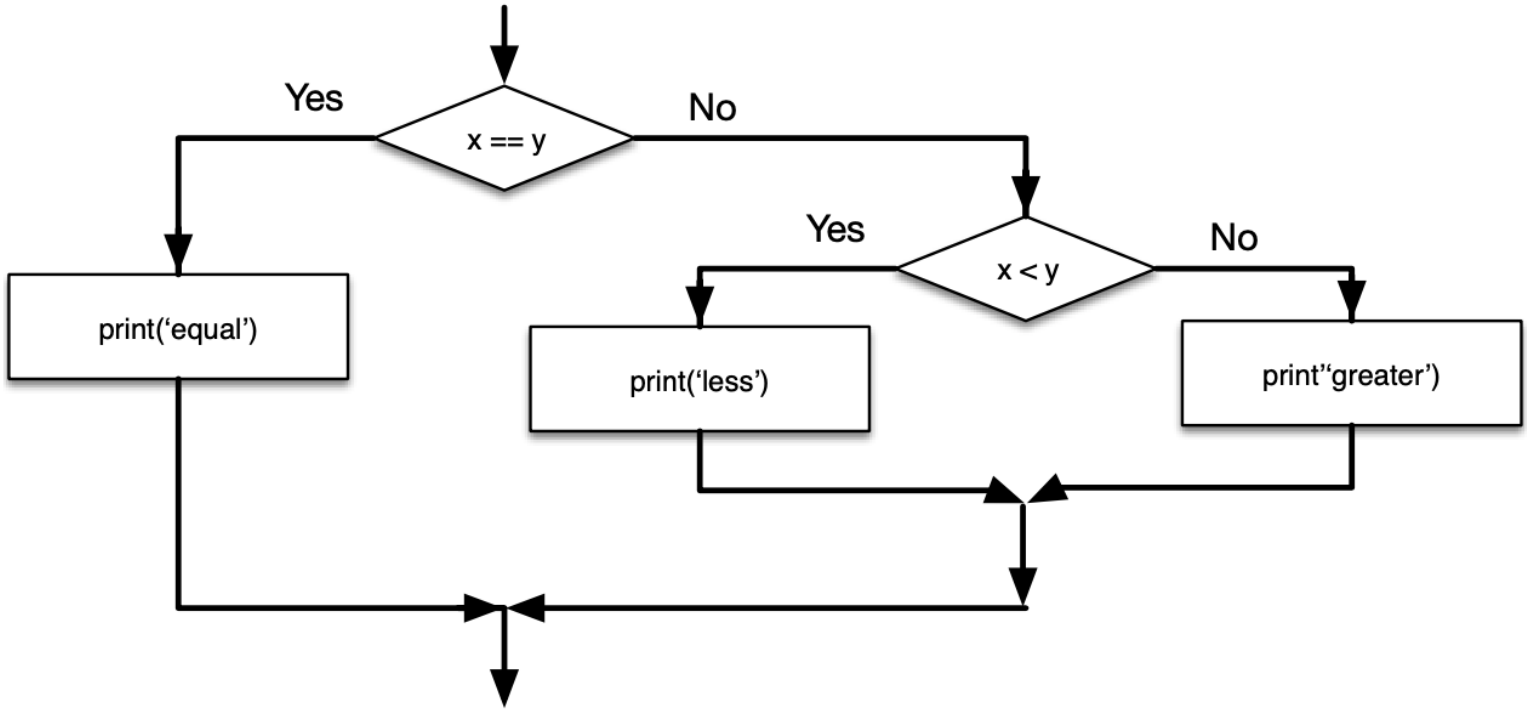


Figure 3.4: Nested If Statements

- Multiple conditions can be implemented with nested statements

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

Nested Conditionals – simplify with logical operators

- Nested conditionals can be too complex and get messy
- Simplify with logical operators

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

- Simplify to

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

Short-circuit evaluation of logical expressions

- For a logical expression, the evaluation is finished as soon as a conclusion is reached
- Examples:
  - if A and B: -- when A is False, B will not need to be evaluated
  - if A or B: -- when A is True, B will not need to be evaluated

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
```

```
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
```

```
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Guard Evaluation

- y != 0 plays as a guard of (x/2) > 2

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

Guard Evaluation (cont'd)

```
1  s = [0,1,2,3,4,5]
2  idx = 8
3  if s[idx] % 2 == 1:
4      print("s[idx] is odd")
```

```
---------------------------------------
IndexError
/var/folders/m2/ngsq8rt93tb2zy4dk71x3sr
      1 s = [0,1,2,3,4,5]
      2 idx = 8
----> 3 if s[idx] % 2 == 1:
      4     print("s[idx] is odd")

IndexError: list index out of range
```

- Fix: check if idx is in range

```
1  s = [0,1,2,3,4,5]
2  idx = 8
3  if idx < len(s) and s[idx] % 2 == 1:
4      print("s[idx] is odd")
```

Catching exceptions

- What if the error is unexpected?

```
>>> prompt = "What...is the airspeed velocity of an unladen swallow?\n"
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
>>>
```

Try/catch

- Another example: converting Fahrenheit to Celsius temperature

```
inp = input('Enter Fahrenheit Temperature: ')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)

python fahren.py
Enter Fahrenheit Temperature:72
22.2222222222222


python fahren.py
Enter Fahrenheit Temperature:fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(inp)
ValueError: could not convert string to float: 'fred'
```

Try/catch

- Insurance policy:
  - try – code block that may have unexpected errors
  - except – catching and handling the error case, skip if no error occurs

```
inp = input('Enter Fahrenheit Temperature:')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print(cel)
except:
    print('Please enter a number')

# Code: http://www.py4e.com/code3/fahren2.py
```

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.22222222222222

python fahren2.py
Enter Fahrenheit Temperature:fred
Please enter a number
```

Exercises

Exercise 1: 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
```

Exercises

Exercise 2: Rewrite your pay program using `try` and `except` so that your program handles non-numeric input gracefully by printing a message and exiting the program. The following shows two executions of the program:

```
Enter Hours: 20
Enter Rate: nine
Error, please enter numeric input

Enter Hours: forty
Error, please enter numeric input
```

Exercises

Exercise 3: Write a program to prompt for a score between 0.0 and 1.0. If the score is out of range, print an error message. If the score is between 0.0 and 1.0, print a grade using the following table:

```
Score   Grade
>= 0.9     A
>= 0.8     B
>= 0.7     C
>= 0.6     D
< 0.6     F
~~~
```

```
Enter score: perfect
Bad score

Enter score: 10.0
Bad score

Enter score: 0.75
C

Enter score: 0.5
F
```

Functions

- Function: a named sequence of statements that perform a computation.
- Every function has a name

```
>>> len('Hello world')
11
>>>
```

- Name: len
- Argument(s): what is inside the parenthesis, can be multiple (separated by “,”)
  ‘hello world’
- Return value: result – type of the result is type of the function
  11 – type is int

Built-in Functions

- Type
```
>>> type(32)
<class 'int'>
```

- type-conversion

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int() with base 10: 'Hello'
```

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

Random

- Generating random numbers between [0.0, 1.0) -- half close half open
- import random – import a module containing predefined functions and variables
- random.random() – calling the random() function which is within the module random

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

Output

```
0.1113286792115235 6
0.595094922789024 1
0.0482026588499687 7
0.841003109276478
0.997914947094958
0.0484233080336811 1
0.741629594820840 5
0.510535245390327
0.2744704017197814 3
0.0285118054727858 67
```

Random (cont'd)

- randint(low, high) -- returns a random integer between [low, high] – including both ends
- choice(S) – randomly select an element from the list S

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

- Both still require importing the random module
- Exercise: choice can be rewritten using randint
- Exercise: randint can be rewritten using random

Math Functions

- Basic math functions are implemented in the math module

```
>>> import math

>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

- dot notation – connecting a module and a function/variable defined in the module
- math.log10(…) computes base 10 logarithm of the input argument
- math.log(…) – log with base e, also other bases
- math.sin(…) – trigonometric functions: sin, cos, tan, etc.

Math Functions (cont'd)

- math.pi – π, a variable, not a function, thus no parenthesis
- math.sqrt(…) – taking a square root

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865476

>>> math.sqrt(2) / 2.0
0.7071067811865476
```

Existing Modules

Existing Modules	
Module Name	**Description**
array	Provides compact array storage for primitive types.
collections	Defines additional data structures and abstract base classes involving collections of objects.
copy	Defines general functions for making copies of objects.
heapq	Provides heap-based priority queue functions (see Section 9.3.7).
math	Defines common mathematical constants and functions.
os	Provides support for interactions with the operating system.
random	Provides random number generation.
re	Provides support for processing regular expressions.
sys	Provides additional level of interaction with the Python interpreter.
time	Provides support for measuring time, or delaying a program.

Table 1.7: Some existing Python modules relevant to data structures and algorithms.

Existing modules (cont'd)

- random
  - https://docs.python.org/3/library/random.html
- math
  - https://docs.python.org/3/library/math.html
- sys – interacting with python interpretor
  - https://docs.python.org/3/library/sys.html
  - sys.float_info, sys.int_info
  - sys.argv, sys.exit(…)
- os – interacting with operating system within the python code
  - https://docs.python.org/3/library/os.html
  - E.g.: executing an external command – os.system(command)
  - Can write script with python
- time – get system time
  - https://docs.python.org/3/library/time.html
  - time.time() – return a float number counting seconds since Jan 1st, 1970, 00:00:00 UTC
  - time.gmtime(), time.gmtime(given_time) – utc time
  - time.localtime(), time.localtime(given_time) – local time

Self-defined functions

- Function definition: a named sequence of statements which are executed when the name is called
- First line: header – def, name, parenthesis (arguments inside), colon
- Body: sequence of statements
- Same rule with colon and indentation
  no indentation = end of the definition.

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')
```

Self-defined functions (cont'd)

Defining a function creates a variable with the same name.

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> print(type(print_lyrics))
<class 'function'>
```

The value of `print_lyrics` is a *function object*, which has type "function".

Self-defined functions (cont'd)

- Calling:
  - Inside normal code
  - Inside another definition of function

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Self-defined functions (cont'd)

- Putting all together

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')


def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()

# Code: http://www.py4e.com/code3/lyrics.py
```

Execution order
– code definition is only executed when the function is called.

- Move repeat_lyrics() to the beginning?
- repeat_lyrics() before print_lyrics()?

Self-defined functions (cont'd)

- Argument(s) – bruce is just a name of the argument,
- it can be anything, depending how the function is called.

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

- String, int, float, a variable

```
>>> print_twice('Spam')
Spam
Spam
```

```
>>> print_twice(17)
17
17
```

```
>>> import math
>>> print_twice(math.pi)
3.141592653589793
3.141592653589793
```

Self-defined functions (cont'd)

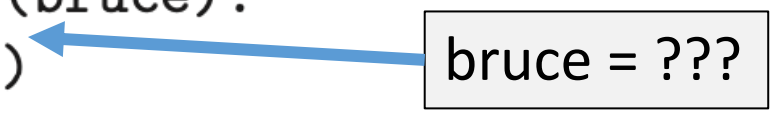
- Expression, variable

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

- For each call, assign the argument (evaluated expression) to bruce, then execute

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```



Self-defined functions (cont'd)

- Some function returns something

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2

def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)
```

- Some function returns nothing (void function, returns None)

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

- None – a special value with a special NoneType

```
>>> print(type(None))
<class 'NoneType'>
```

Why functions?

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read, understand, and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Debugging functions can be tricky.

Exercises

Exercise 4: What is the purpose of the “def” keyword in Python?

a) It is slang that means “the following code is really cool”
b) It indicates the start of a function
c) It indicates that the following indented section of code is to be stored for later
d) b and c are both true
e) None of the above

Exercises

Exercise 5, What is the output of the following program?

```
def fred():
    print("Zap")

def jane():
    print("ABC")

jane()
fred()
jane()
```

a) Zap ABC jane fred jane
b) Zap ABC Zap
c) ABC Zap jane
d) ABC Zap ABC
e) Zap Zap Zap

Exercises

Exercise 6: Rewrite your pay computation with time-and-a-half for overtime and create a function called **computepay** which takes two parameters (**hours** and **rate**).

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

For overtime (hours after 40), the rate is 1.5 * rate

Exercises

Exercise 7: Rewrite the grade program from the previous chapter using a function called **computegrade** that takes a score as its parameter and returns a grade as a string.

Score	Grade
> 0.9	A
> 0.8	B
> 0.7	C
> 0.6	D
<= 0.6	F

Program Execution:

```
Enter score: 0.95
A

Enter score: perfect
Bad score

Enter score: 10.0
Bad score

Enter score: 0.75
C

Enter score: 0.5
F
```

The end