CONCEPTS OF PROGRAMMING

MODULE 2
MPCS 50101

THE UNIVERSITY OF CHICAGO

VARIABLES, EXPRESSIONS, AND STATEMENTS

MODULE 2
MPCS 50101

THE UNIVERSITY OF CHICAGO

VARIABLES, EXPRESSIONS AND STATEMENTS

VALUES AND TYPES

- Python has built-in data types to represent different values
  - Numeric: int, long, float, complex
  - Sequences: str, bytes, tuple
  - Collections: dict, set, list

```
>>> print(123)
123

>>> print(98.6)
98.6

>>> print('Hello world')
Hello world
```

VARIABLES, EXPRESSIONS AND STATEMENTS

VALUES AND TYPES

- Fixed values such as numbers, letters, and strings are called “constants” because their value does not change
- String constants
  - Single quotes (')
  - Double quotes (")
  - Triple """ for multiline

```
>>> print(123)
123

>>> print(98.6)
98.6

>>> print('Hello world')
Hello world

>>> print("""
Hello
world!
""")
Hello
world!
```

VARIABLES, EXPRESSIONS AND STATEMENTS

VALUES AND TYPES

- Python can tell you what kind of type a value is

```
>>> type('Hello, World!')
<type 'str'>

>>> type(17)
<type 'int'>

>>> type(3.2)
<type 'float'>

>>> type('17')
<type 'str'>
```

VARIABLES, EXPRESSIONS AND STATEMENTS

VARIABLES

- A variable is a name that refers to a value stored in memory
- Notice
  - You do not have to tell Python what the type is
  - You can change the value of a variable after declaration

```
>>> x = 5

>>> y = 17.2

>>> pi = 3.1415926535897932

>>> name = "Johnny"

>>> name = 'Ada'

>>> type(name)
<type 'str'>
```

VARIABLES, EXPRESSIONS AND STATEMENTS

VARIABLES

- Choose names that are meaningful
- Name rules
  - Start with letter or "_"
  - Contains letters, number, "_"
  - Case sensitive

```
>>> x = 'Ada'

>>> name = 'Ada'

>>> Name = "Ada'

>>> 1st_name = 'Ada'
  File "<stdin>", line 1
    1st_name = 'Ada'
             ^
SyntaxError: invalid syntax
```

VARIABLES, EXPRESSIONS AND STATEMENTS

VARIABLES

- Reserved words in Python language
  - Can not use as variable names

```
and   del   for   is   raise
assert   elif   from   lambda
return   break   else
global   not   try   class
except   if   or   while
continue   exec   import
pass   yield   def   finally
in   print   as   with
```

VARIABLES, EXPRESSIONS AND STATEMENTS

ASSIGNMENT STATEMENTS

- Expression - a combination of values, variables, and operators
- Assignment - assign value of a variable

```
5         # expression
y         # expression
y + 5     # expression


x = 10    # assignment
x = y + 5 # assignment with
          # expression
```

VARIABLES, EXPRESSIONS AND STATEMENTS

OPERATORS AND OPERANDS

- Operators symbols represent computations
  - + (addition),
  - - (subtraction)
  - * (multiplication)
  - / (division)
  - ** (power)
  - % (modulo, remainder)

```
>>> 2 + 3
5

>>> 4 ** 2
16

>>> x = 2
>>> x + 3
5

>>> 4 + (3*4)
16

>>> 12 % 2
0
```

VARIABLES, EXPRESSIONS AND STATEMENTS

OPERATORS AND OPERANDS

- Notice that you can mix variables and constant values in expressions
- Modulo
  - 12 / 2 = 6 R 0
  - 11 / 2 = 5 R 1
- How can you tell if a number is odd or even using modulo?

```
>>> x = 2
>>> y = 5
>>> x + y
7

>>> 12 % 2
0

>>> 11 % 2
1
```

VARIABLES, EXPRESSIONS AND STATEMENTS

OPERATORS AND OPERANDS

- Python division behavior
  - Integer division converts to float
  - Floating point division produces floating point numbers
  - Integer divided by float produces float

```
>>> 10 / 2
5.0

>>> 9 / 2
4.5

>>> 9 / 2.0
4.5

>>> 10.0 / 2.0
5.0

>>> 2/9.0
0.2222222222222222
```

VARIABLES, EXPRESSIONS AND STATEMENTS

OPERATORS AND OPERANDS

```
# Python 2 vs. 3 Warning
#
# Division operator defaults to floating point value


# Python 2
print 7 / 5  # 1

# Python 3
print 7 / 5. # 1.4
```

VARIABLES, EXPRESSIONS AND STATEMENTS

OPERATORS AND OPERANDS

- You can convert between different types

```
>>> x = 10 / 2
>>> type(x)
<type 'int'>

>>> y = 9 / 2.0
>>> type(y)
<type 'float'>

>>> int(1.3)
1
```

CAREFUL

VARIABLES, EXPRESSIONS AND STATEMENTS

ORDER OF OPERATIONS

- PEMDAS
  - Parenthesis are always respected
  - Exponentiation (raise to a power)
  - Multiplication, Division, and Remainder
  - Addition and Subtraction
- Left to right

```
>>> x = 1 + 2 ** 3 / 4 * 5

>>> print x
11

>>> x = 1 + (2 ** 3) / 4 * 5

>>> print x
11
```

VARIABLES, EXPRESSIONS AND STATEMENTS

OPERATORS AND TYPES

- Operators can have different behaviors based on the type
- "+"
  - Number = addition
  - String = concatenate
- "*"
  - Number = multiply
  - String = repeat

```
>>> x = 1 + 2
>>> x
3

>>> x = "hello " + "world"
>>> x
hello world


>>> x = "hi " * 4
>>> x
'hi hi hi hi '
```

VARIABLES, EXPRESSIONS AND STATEMENTS

COMMENTS

- Anything after a # is ignored
- Comments help to explain what is happening in your code
- Document who wrote the code or other ancillary information
- Turn off a line of code - perhaps temporarily (debugging)

```
# Assignment 1

# Length and width as defined
by the problem
length = 1 + 3
width = x + 3

#print length
#print width

# Compute the area
area = length * width
```

VARIABLES, EXPRESSIONS AND STATEMENTS

PUTTING IT ALL TOGETHER

```
# Calculate the surface area of a cube
length = 5
width = 5

# Compute the area of a single face
single_face_area = length * width

# Compute the surface area of all faces
number_of_faces = 6
surface_area = number_of_faces * single_face_area
C
print("The surface area is ")
print(surface_area)
```

CONTROL FLOW AND CONDITIONAL EXECUTION

MODULE 2
MPCS 50101

THE UNIVERSITY OF CHICAGO

CONTROL FLOW

- Sequential execution
  - Everything runs
- Conditional execution
  - Input and values determine what code gets executed

```
x = 5

if x < 10:
    print("x is less than 5")
    if x > 4:
        print("x is greater than 4")
```

CONTROL FLOW

- Control structures
  - Direct the behavior of your program

```
x = 5

if x < 10:
    print("x is less than 5")
    if x > 4:
        print("x is greater than 4")
```

CONDITIONAL EXECUTION

- Not all code is executed in a program
- The same program may need to executed different code paths depending on the data

```
x = 2

if x < 10:
    print("x is less than 5")

    if x > 4:
        print("x is greater than 4")
```

THIS WILL NOT BE EXECUTED

CONDITIONAL EXECUTION

- Debugging Tip
  - Make sure that you test that your program works under all code paths
  - Ask someone else to help test your program

```
x = 2

if x < 10:
    print("x is less than 5")

    if x > 4:
        print("x is greater than 4")
```

BOOLEAN EXPRESSION

BOOLEAN EXPRESSION

- Boolean expression
  - An expression that is either true or false

```
>>> 5 == 5
True

>>> 3 == 2
False

>>> 1 < 10
True
```

BOOLEAN EXPRESSION

- `bool` is a type that has values of True or False

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

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

BOOLEAN EXPRESSION

- `True` is not the same as `true`

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


>>> type(true)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
```

SYNTAX ERROR

BOOLEAN EXPRESSION

- Comparison Operators
  - Compare values and evaluate as Boolean

```
x == y  # x is equal to y

x != y  # x is not equal to y

x > y   # x is greater than y

x < y   # x is less than y

x >= y  # x greater than or equal to y

x <= y  # x is less than or equal to y
```

BOOLEAN EXPRESSION

- Not!

```
5 == 5  # True

5 != 5  # False

4 != 3  # True
```

BOOLEAN EXPRESSION

- Chained comparison operators

```
>>> x = 2

>>> 1 < x < 3
True

>>> 10 < x < 20
False

>>> 3 > x <= 2
True

>>> 2 == x < 4
True
```

BOOLEAN EXPRESSION

- Logical operators combine conditions
  - `and`
  - `or`
  - `not`

```
# Both expressions must evaluate
# to `True` for the statement to
# be evaluated

x = 3
y = 5

if x == 3 and y == 5:
    print("This would be evaluated")

if x == 3 and y == 4:
    print("This is not evaluated")
```

BOOLEAN EXPRESSION

- Logical operators combine conditions
  - `and`
  - `or`
  - `not`

```
# One expressions must evaluate
# to `True` for the statement to
# be evaluated

x = 3
y = 5

if x == 3 or y == 10:
    print("This would be evaluated")

if x == 0 or y == 5:
    print("This would be evaluated")
```

BOOLEAN EXPRESSION

- Logical operators combine conditions
  - `and`
  - `or`
  - `not`

```
# Not returns True is statements are
# not True, otherwise returns False


# Evaluates to `True`
x = False
print(not x)
```

BOOLEAN EXPRESSION

- Common error
  - "=" is for assignment
  - "==" is for comparison

```
>>> y = 10

>>> x = 100

>>> x == y
False

>>> x = y

>>> x
10

>>> x == y
True
```

ASSIGNMENT

COMPARISON

ASSIGNMENT

COMPARISON

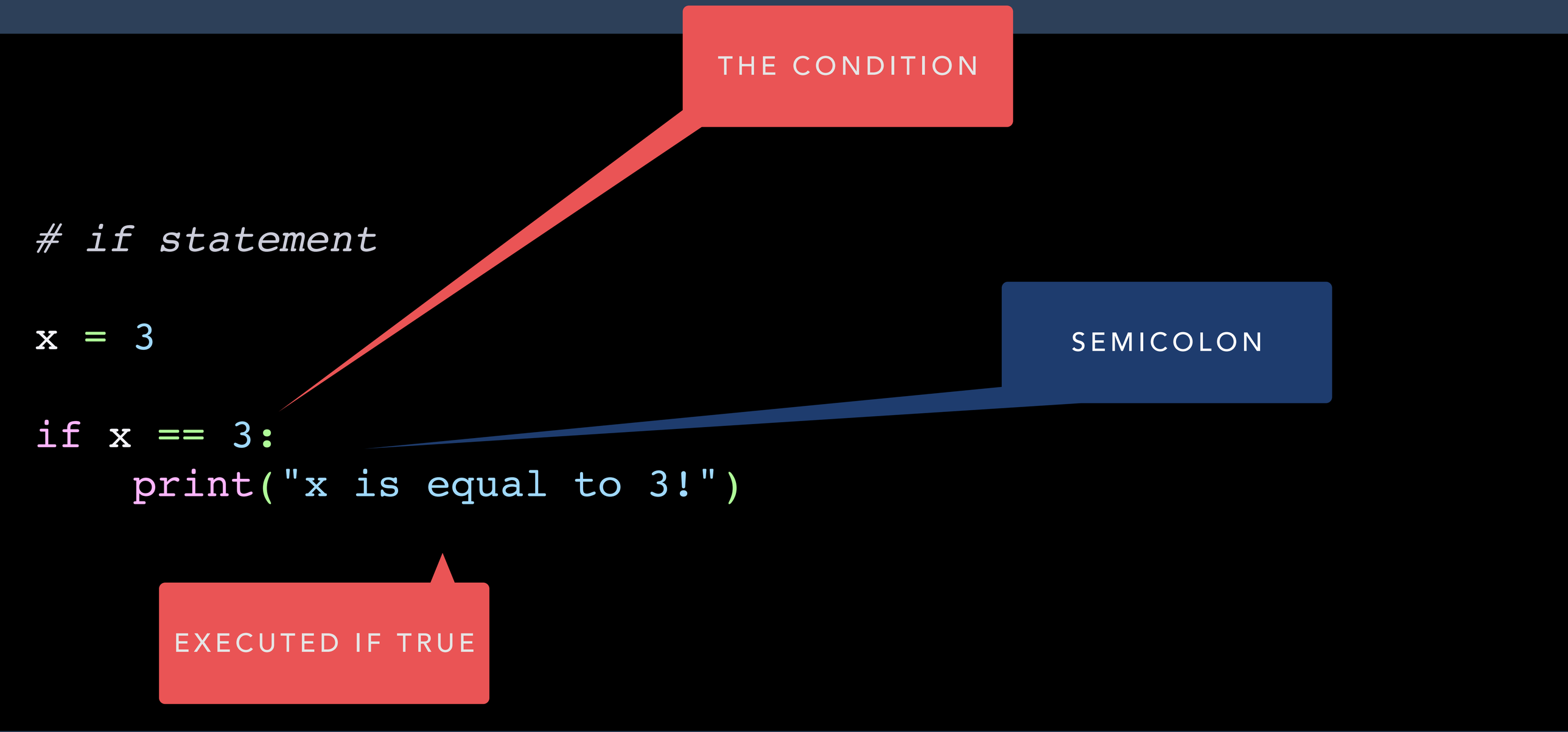
CONDITIONAL STATEMENTS

CONDITIONAL STATEMENTS

- Conditional statements
  - Check values
  - Change code paths

```
# Conditional checks using the `if`
# statement

x = 3

if x == 3:
    print("x is equal to 3!")
```

CONDITIONAL STATEMENTS




```
# if statement

x = 3

if x == 3:
    print("x is equal to 3!")
```

CONDITIONAL STATEMENTS

```
if x != y:
  print("x does NOT equal y")

if x > y:
  print("x is greater than y")

if x < y:
  print("x is less than y")

if x >= y:
  print("x is greater than or equal to y")

if x <= y:
  print("x is less than or equal to y")
```

CONDITIONAL STATEMENTS

- Condition must be indented
- Python allows tabs or spaces
  - Stick to spaces
  - Editors can help

```
if x == 3:
    print("x is equal to 3!")
```

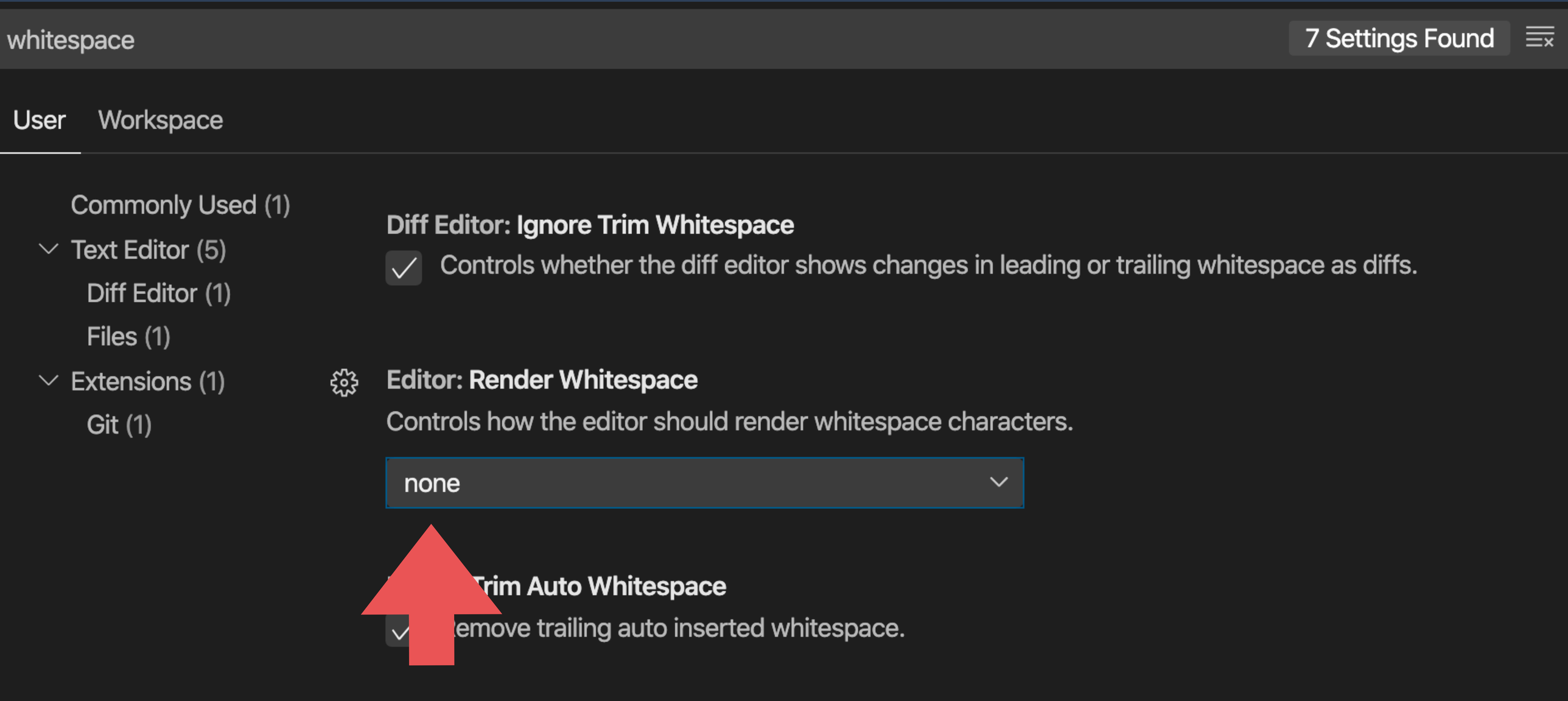
```
if x == 3:
print("x is equal to 3!")

File "<string>", line 7
    print("x is equal to 3!")
        ^
IndentationError: expected an indented
block
```

CONDITIONAL STATEMENTS

0-conditional-statements.py > ...

```
1
2   x = 2
3
4   if x < 10:
5       print("x is less than 5")
6
7       if x > 4:
8           print("x is greater than 4")
9
10
11
12
```

CONDITIONAL STATEMENTS



CONDITIONAL STATEMENTS

18

PEP indentation using 4 spaces vs. Google's 2 spaces (self.learnpython)

submitted 3 years ago * by [deleted]

I'm currently reading the Python introduction by google, and I'm intrigued as to why the standard spacing is recommended to be 4 but google uses 2:

> A common question beginners ask is, "How many spaces should I indent?" According to the official Python style guidelines (PEP 8), you should indent with 4 spaces. (Fun fact: Google's internal style guideline dictates indenting by 2 spaces!)

Why would their internal style guideline recommend a standard of two spaces

username password

remember me reset password login

CONDITIONAL STATEMENTS

- Maintain indent to indicate the scope of the block
  - Lines affected
- Reduce indent back to indicate the end of the block

```
1   x = 5
2   if x > 2 :
3       print 'Bigger than 2'
4       print 'Still bigger'
5   print 'Done with 2'
6
7   for i in range(5) :
8       print i
9       if i > 2 :
10          print 'Bigger than 2'
11      print 'Done with i', i
12  print 'All Done'
13
```

CONDITIONAL STATEMENTS

- Blank lines are ignored
- Comments lines are always ignored

```
# The number of stuff
x = 2

# This comment is ignored by the interpreter
if x < 10:
    print("x is less than 5")

    # This comment is also ignored by the interpreter
    if x > 4:
        print("x is greater than 4")
```

CONDITIONAL STATEMENTS

- Variables modified in the scope maintain new value

```
x = 4
print("x equals", x)

if x % 2 == 0:
    print("Bad even numbers")
    x = x + 1
    print("x equals", x)


print("x equals", x)
#x=5
```

CONDITIONAL STATEMENTS

```
x = True

if x == True:
  print("x is True")
  print("This is a compound statement")
  print("You can put as many statements as you like")
  print("Below the conditional")
```

CONDITIONAL STATEMENTS

```
x = 3
y = 5

if x == 3 and y == 5:
    print("True")

if x == 3 or y == 10:
    print("True")

if x == 2 or y == 2:
    print("False")
```

LOGICAL OPERATORS

CONDITIONAL STATEMENTS

- The `pass` statement does nothing
- Can be useful for debugging or stubbing out your program

```
x = 100

# Check if even
if x % 2 == 0:
 print("This number is even")

# Check if divisible by 10
if x % 10 == 0:
    print("X is divisible by 10")

# Check if divisible by 100
if x % 100 == 0:
    # Finish this later
    pass
```

CONDITIONAL STATEMENTS

- Alternative execution
- Execute a statement if condition is False

```
# Alternative execution

if x % 2 == 0:
    print("This number is even")
else:
    print("x is odd")
```

CONDITIONAL STATEMENTS

- Which is better?

```
if x % 2 == 0:
    print("This number is even")
else:
    print("X is odd")
```

```
if x % 2 != 0:
    print("This number is odd")
else:
    print("X is even")
```

CONDITIONAL STATEMENTS

- Golden path

```
if x % 2 == 0:
    print("This number is even")
else:
    print("X is odd")
```

```
if x % 2 != 0:
    print("This number is odd")
else:
    print("X is even")
```

CONDITIONAL STATEMENTS

- Chained conditionals

```
x = 3
y = 10

if x < y:
    print("x is less than y")

elif x > y:
    print("x is greater than y")

else:
    print("x equals y")
```

CONDITIONAL STATEMENTS

- Multiple `elif` statements
- Only the first `True` is executed

```
age = 2

if age < 1:
    print("newborn")
elif age < 16 :
    print("Can't drive")
elif age < 21:
    print("Can't drink")
elif age > 40:
    print("You're over the hill")
elif age >= 65:
    print("You can retire")
else:
    print("You're dead")
```

CONDITIONAL STATEMENTS

- Can you find he logic error?

```
age = 2

if age < 1:
    print("newborn")
elif age < 16 :
    print("Can't drive")
elif age < 21:
    print("Can't drink")
elif age > 40:
    print("You're over the hill")
elif age >= 65:
    print("You can retire")
else:
    print("You're dead")
```

CONDITIONAL STATEMENTS

- You do not need an `else`

```
age = 2

if age < 1:
    print("newborn")
elif age < 16 :
    print("Can't drive")
elif age < 21:
    print("Can't drink")
elif age > 40:
    print("You're over the hill")
elif age >= 65:
    print("You can retire")
else:
    print("You're dead")
```

CONDITIONAL STATEMENTS

- Conditionals can be nested

```
x = 4
if x % 2 == 0:
    if x > 2:
        print("x greater than 2")
    else:
        print("x less than 2")
```

CONDITIONAL STATEMENTS

- Which is better?

```
if x % 2 == 0:
    if x > 2:
        print("x greater than 2")
    else:
        print("x less than 2")
```

```
if x % 2 == 0 and x > 0:
    print("x greater than 2")
else:
    print("x less than 2")
```

FUNCTIONS

MODULE 2
MPCS 50101

THE UNIVERSITY OF CHICAGO

FUNCTIONS

- Function is a piece of reusable code
  - Takes arguments(s) as input
  - Does some computation
  - Optionally returns results

```
>>> type("Hello")
<type 'str'>
```

FUNCTIONS

- Built-in functions as part of Python
  - input(), type(), float(), int(), ...
- Functions we define and use

```
# Built-function
>>> type("Hello")
<type 'str'>
```

```
# User defined
def say_hello():
  print("Hello")
```

FUNCTIONS

- Why functions?
  - Reduce repetitive code
  - Organization
  - Debugging
  - Code sharing

```
def say_hello():
  print("Hello")
```

FUNCTIONS

- Define a function
  - Use the `def` reserved word
  - Specify the name
  - Input parameters (optional)
- Names must be unique
  - Can not use function names as variable names

```
def say_hello():
  print("Hello")
```

FUNCTIONS

KEYWORD

FUNCTION NAME

```
def say_hello():
  print("Hello")
```

FUNCTION NAME

PARAMETER

```
def say_personalized_hello(name):
    print("Hello " + name)
```

FUNCTIONS

- "call" or "invoke" the function
  - Function name
  - Parentheses
  - Arguments in an expression

```
def say_hello():
  print("Hello")
```

```
print("Running a function"
say_hello()
print("Function ran.")

# Running a function
# Hello
# Function ran.
```

FUNCTIONS

PARAMETER

```
def say_personalized_hello(name):
    print("Hello " + name)
```

PARAMETERS

```
def say_personalized_hello(name1, name2, name3):
    print("Hello " + name1)
    print("Hello " + name2)
    print("Hello " + name3)
```

ARGUMENT

```
say_personalized_hello("Jane")
```

ARGUMENT

```
say_personalized_hello("Jane","Bob","Lola")
```

FUNCTIONS

```
>>> def greet(lang):
...     if lang == 'es':
...         print 'Hola'
...     elif lang == 'fr':
...         print 'Bonjour'
...     else:
...         print 'Hello'
...

>>> greet('en')
Hello
>>> greet('es')
Hola
>>> greet('fr')
Bonjour
```

FUNCTIONS

- The call must match the declaration

```
def say_hello():
    print("Hello")

say_hello(1234)
```

```
Traceback (most recent call last):
  File "workspace.py", line 6, in
<module>
    say_hello(1234)
TypeError: say_hello() takes no
arguments (1 given)
```

FUNCTIONS

- You are responsible for managing the type of the argument passed

```
def plus_one(number):
    return number + 1


print plus_one(1)
>>> 2



print plus_one("Lola")

Traceback (most recent call last):
  File "workspace.py", line 6, in
<module>
    return number + 1
TypeError: cannot concatenate 'str' and
'int' objects
```

FUNCTIONS

- You are responsible for managing the type of the argument passed

```
def plus_one(number):
    return number + 1


print plus_one(1)
>>> 2



print plus_one("Lola")

Traceback (most recent call last):
  File "workspace.py", line 6, in
<module>
      return number + 1
TypeError: cannot concatenate 'str' and
'int' objects
```

IN OTHER LANGUAGES, YOU MAY SPECIFY THE ARGUMENT TYPES

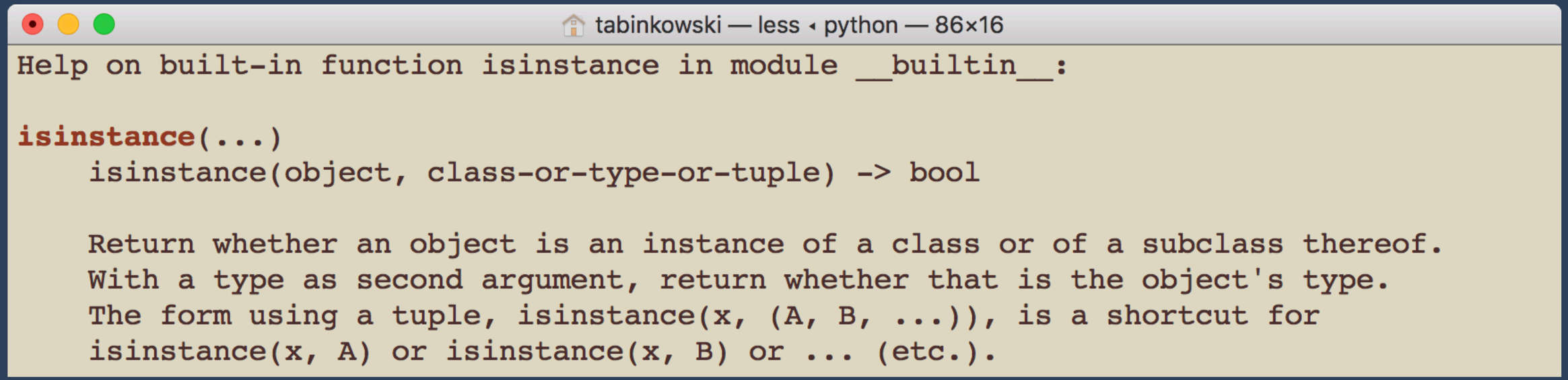
FUNCTIONS

- Use builtin function `isinstance` to validate types
- help(instance) at interpreter

```
def plus_one(number):
    if isinstance(number,int):
        return number + 1
    else:
        print("Not a valid argument"
```

```
print plus_one(1)
# >>> 2

print plus_one('hello')
# >>> Not a valid argument
# None
```

FUNCTIONS



- help(instance) at interpreter

FUNCTIONS

- A “fruitful” function is one that produces a result (or return value)
- The `return` statement ends the function execution and “sends back” the result of the function

```
def add(x,y):
    summed = x + y
    return summed
```

```
>>> add(1,3)
4
```

FUNCTIONS

- You are responsible for the managing the type of the returned value

```
def add(x,y):
    summed = x + y
    return summed
```

```
>>> print("Hello" + add(1,3)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and
'int' objects
```

FUNCTIONS

- Variables declared in a function are local scope
  - Only accessible in that function

```
def add(x,y):
    summed = x + y
    return summed

print add(1,4)
print summed
```

SUMMED IS ONLY AVAILABLE IN THE FUNCTION

```
>>> 5
Traceback (most recent call last):
  File "workspace.py", line 8, in
<module>
    print summed
NameError: name 'summed' is not defined
```

FUNCTIONS

```
summed = 100
print("Initial value of summed = ",summed)

def add(x,y):
    summed = x + y
    print("In Function: summed = ",summed)
    return summed

add(1,4)
print("Final value of summed = ",summed)

# >>> Initial value of summed =  100
# >>  In Function: summed =  5
# >>> Final value of summed =  100
```

SUMMED IN FUNCTION SCOPE IS NOT SAME AS SUMMED IN MAIN PROGRAM

FUNCTIONS

- You can make a variable global in a function using `global`

```
x = 0

def plus_one():
    global x
    x = x + 1


print("Initial value of x = ",x)
plus_one()
print("Final value x = ",x)


# Initial value of x =  0
# Final value x =  1
```

FUNCTIONS

```
x = 0

def plus_one():
    global x
    x = x + 1

def plus_two():
    plus_one()
    plus_one()

print("Initial value of x = ",x)
plus_two()
print("Final value x = ",x)
```

YOU CAN CALL OTHER FUNCTIONS FROM WITHIN FUNCTIONS

FUNCTIONS

```
x = 0

def plus_one():
    global x
    x = x + 1

def plus_two():
    plus_one()
    plus_one()

print("Initial value of x = "
plus_two()
print("Final value x = ",x)
```

```
Traceback (most recent call last):
  File "workspace.py", line 14, in <module>
    plus_two()
  File "workspace.py", line 10, in plus_two
    plus_one()
  File "workspace.py", line 6, in plus_one
    x = x/0
ZeroDivisionError: integer division or modulo by zero
```

ERROR LOGGING IS HIERARCHY OF CALL ORDER. THE LAST ONE IS BEST PLACE TO START.

VARIABLE ARGUMENTS

VARIABLE ARGUMENTS

- Allow your function to take a variable number of arguments
  - Add items to a collection
  - ?

```
# Python program to illustrate
# *args for variable number of
# arguments

def wordsToPrint(*argv):
  for arg in argv:
    print (arg)

wordsToPrint('Hello','World')


>>> Hello World
```

VARIABLE ARGUMENTS

- Mix and match

```
# Python program to illustrate
# *args with first extra argument

def fun(arg1, *argv):
  print ("First argument :", arg1)
  for arg in argv:
    print("*argv :", arg)

fun('Hello', 'World', "!")
```

```
First argument : Hello
Next argument through *argv : World
Next argument through *argv : ! !
```

VARIABLE ARGUMENTS

- Keyword (kw) arguments (args)

```
# Python program to illustrate
# *kargs for variable number of keyword
arguments

def myFun(**kwargs):
  for key, value in kwargs.items():
    print ("%s == %s" %(key, value))

# Driver code
 myFun(first ='Hello', mid ='World',\
        last='!')

last == Hello
mid == World
first == !
```

ITERATION

MODULE 2
MPCS 50101

THE UNIVERSITY OF CHICAGO

ITERATION

- Iteration
  - Performing repeated tasks
- Python has built-in language features for different types of iteration

```
for i in range(4):
    print(i)
```

FOR

ITERATION

- "Definite loops" execute an exact number of times
- Use the `for` structure to iterate through the members of a set

```
for i in range(0,5):
    print(i)
print('Blastoff!')


# 5
# 4
# 3
# 2
# 1
# Blastoff!
```

ITERATION

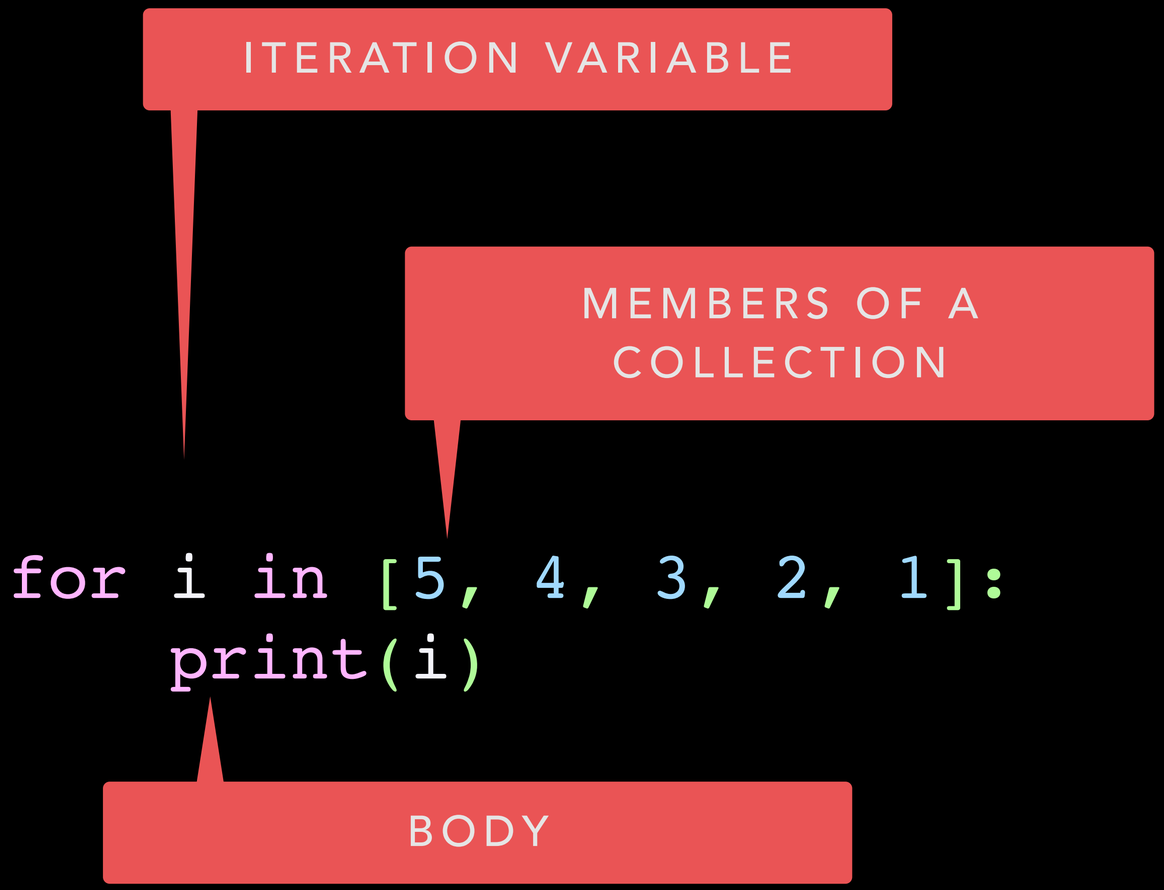
- `for` loops have explicit iteration variables that change each time through a loop

ITERATION VARIABLE

```
for i in range(0,5):
    print(i)
print('Blastoff!')


# 5
# 4
# 3
# 2
# 1
# Blastoff!
```

ITERATION

- 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

- We will discuss data structures later, but a collection of objects can be represented as a list using `[]`

LIST OF INTS

```
x = [5, 4, 3, 2, 1]
print(x)
```

LIST OF STRINGS

```
y = ["dog", "cat", "wolf"]
print(y)
```

LIST OF INTS AND STRING

```
z = ["dog", 1, "cat", 2]
print(z)
```

ITERATION

```
x = [5, 4, 3, 2, 1]
for item in x:
    print(item)
# 5 4 3 2 1

for animal in ["dog", "cat", "wolf"]:
    print(animal)
# dog cat wolf

for item in ["dog", 1, "cat", 2]:
    print(item)
# dog 1 cat 2
```

ITERATION

- Best practice to use meaningful names for iteration variables
  - for animal in animals:
  - for user in users:

```
x = [5, 4, 3, 2, 1]
for item in x:
    print(item)
# 5 4 3 2 1
```

```
for animal in ["dog", "cat"]:
    print(animal)
# dog cat
```

ITERATION

- Find the total of a collection of integers

```
total = 0
print('Total: ', total)

for thing in [9, 41, 12, 3, 74, 15]:
    total = total + thing
    print(total, thing)
print('After', total)

# Total:  0
# 9 9
# 50 41
# 62 12
# 65 3
# 139 74
# 154 15
# After: 154
```

ITERATION

- Find the biggest number in a collection of ints

```
biggest_number = -1
print('Start:', biggest_number)

for value in [9, 41, 12, 3, 74, 15] :
    if value > biggest_number:
        biggest_number = value
        print('New biggest number: ',value)

print 'After:', biggest_number

# Start: -1
# New biggest number:  9
# New biggest number:  41
# New biggest number:  74
# After: 74
```

ITERATION

```
biggest_number = -1

# Iterate through the collection
for value in [9, 41, 12, 3, 74, 15]:

    # If the value is bigger than the current biggest,
    # change its value
    if value > biggest_number:
        biggest_number = value

print 'Biggest number:', biggest_number
```

WHILE

ITERATION

- The `while` statement evaluates a condition before executing code
  - Continues to execute until condition is False

```
n = 5
while n > 0:
    print n
    n = n - 1
print('Blastoff!')


# >>> 5
# >>> 4
# >>> 3
# >>> 2
# >>> 1
# >>> Blastoff!
```

ITERATION

- Common to use variables within iteration to keep state
  - Increment = + 1
  - Decrement = - 1

```
n = 5
while n > 0:
    print n
    n = n - 1
print('Blastoff!')


# >>> 5
# >>> 4
# >>> 3
# >>> 2
# >>> 1
# >>> Blastoff!
```

ITERATION

- Augmented assignment operators

Augmented assignment statement	Equivalent assignment statement
`spam += 1`	`spam = spam + 1`
`spam -= 1`	`spam = spam - 1`
`spam *= 1`	`spam = spam * 1`
`spam /= 1`	`spam = spam / 1`
`spam %= 1`	`spam = spam % 1`

http://automatetheboringstuff.com/chapter3/

ITERATION

- While loops are called “indefinite loops” because they keep going until a logical condition becomes False

```
# Infinite loop
n = 5

while n > 0:
    print("This will go on forever")

print("You'll never get here")
```

ITERATION

- Dead loop never runs
- Special considerations for your code

```
# Dead loop

while n < 0:
    print("This will never run")

print("This will be executed.")
```

ITERATION

- The `break` statement ends the current loop and jumps to the statement immediately following the loop
- Why exit early?
  - Sometimes you won't know the values until something is running

```
n = 0
while n > 0:
    print n
    n = n + 1

    if n == 50:
      break

print("after the break")
```

ITERATION

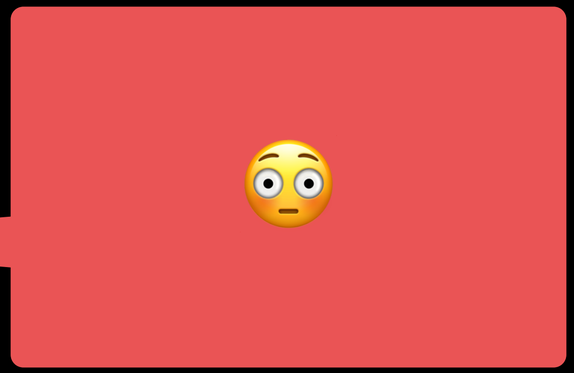
```
n = 0
while n > 0:
    print(n)
    n = n + 1
    if n == 50:
       break


for i in range(0,10):
    print(i)
    if i == 3:
        break
```

ITERATION

- The `continue` statement
  - Ends the current iteration
  - Jumps to the top of the loop and starts the next iteration

```
for i in range(0,10):
    if i == 3:
        continue
    print(i)
```

```
0
1
2
4
5
6
7
8
9
```



STRINGS

MODULE 2
MPCS 50101

THE UNIVERSITY OF CHICAGO

STRINGS

- A string is a sequence of characters
- A string literal uses quotes
  'Hello' or "Hello"
- For strings, + means
  “concatenate”

```
str1 = "Hello "
str2 = 'World'
greeting = str1 + str2

print(greeting)
#Hello World
```

STRINGS

- When a string contains numbers, it is still a string

```
>>> str3 = '123'
124

>>> str3 = str3 + 1
Traceback (most recent call last):
File "<stdin>", line 1, in
<module>TypeError: cannot concatenate
'str' and 'int' objects
```

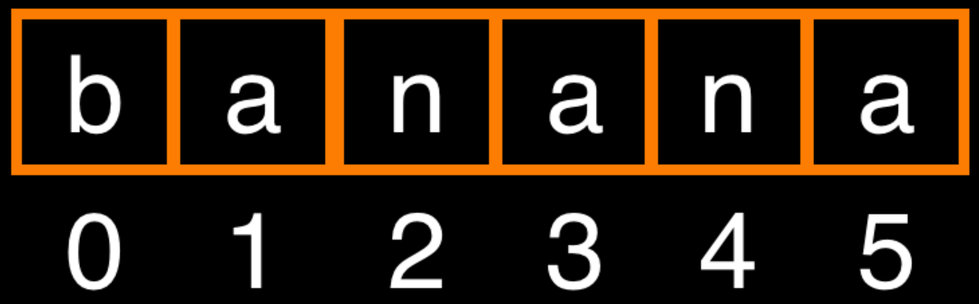
STRINGS

- We can convert numbers in a string into a number using int()

```
>>> str3 = '123'

>>> x = int(str3) + 1
>>> print(x)
124
```

STRINGS

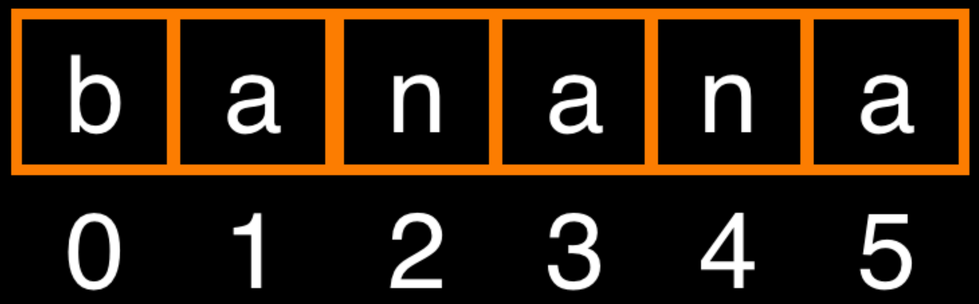
- Strings are a collection of characters
- Get characters with index specified in square brackets




```
fruit = 'banana'
letter = fruit[1]
print(letter)

# a
```

STRINGS

- The index value must be an integer and starts at zero




```
fruit = 'banana'
letter = fruit[1]
print(letter)

# a
```

STRINGS

```
x = "abcdefghijklmnopqrstuvwxyz"

print(x[0])
print(x[25])


# a
# z
```

STRINGS

- The index value can be an expression that is computed

```
x = "abcdefghijklmnopqrstuvwxyz"
print(x[0])
# a

print(x[20+5])
# a

y = 5
print(x[20 + y])
# z
```

STRINGS

- You will get a Python error if you attempt to index beyond the end of a string

```
x = "abcdefghijklmnopqrstuvwxyz"
print(x[0])
print(x[26])

Traceback (most recent call last):
File "<stdin>", line 1, in
<module>IndexError: string index out of
range
```

STRINGS

- There is a built-in function len() that gives us the length of a string

```
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
```

STRINGS

- Note index vs. length

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

# Len is 6
# index is 0...5
```

STRINGS

- A definite loop using a `for` statement is much more concise (but not necessarily better)
- The iteration variable is completely taken care of by the for loop
  - We loose the index

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


# b
# a
# n
# a
# n
# a
```

STRINGS

- Same thing different ways

```
# While loop
fruit = 'banana'
index = 0

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

```
# For loop
fruit = 'banana'
for letter in fruit:
    print("#", letter)
```

STRINGS

- Slicing strings allows us to select a subset
- string[start:stop]

```
>>> quote = "Four score and seven years ago."

>>> quote[5:10]
'score'

>>> quote[11:12]
'a'

>>> quote[0:4]
'Four'
```

STRINGS

- Beginning to index
  - string[:stop]
- Index from beginning to end
  - string[start:]

```
>>> quote[:4]
'Four'

>>> quote[15:]
'seven years ago.'
```

STRINGS

- The `in` keyword checks to see if one string is "in" another string

```
fruit = 'banana'

# Find a character in string
if 'a' in fruit:
    print('Found it!')

# Find a string in a string
if 'ban' in fruit:
    print('Found it!')
```

THE `IN` EXPRESSION RETURNS TRUE OR FALSE

STRINGS

- Relational operators work on strings
- Based on lexicographical order
  - Alphabetical

```
'Apple' == "Apple" # True

'apple' == "Apple" # False

'chalk' < 'chalkboard' # True

'a' > 'A' # True
```

STRINGS

- Upper and lowercase compared by ordinal values

```
'a' > "A"       # True

print(ord("A")) # 65

print(ord("a")) # 97
```

SUMMARY

MODULE 1
MPCS 50101

THE UNIVERSITY OF CHICAGO

SUMMARY

- Control structures

```
# For
for i in [0,1]:

# While
while x > 0:

# If/Elif/Else
if x > 0:

elif:

else:
```

SUMMARY

- Functions

```
# Void function with no arguments
def myFunction():
  print("Hello world")

# Function returning a value
def myFunction2():
  return "Hello world")


# Function taking argument and
# returning a value
def myFunction3(name):
  return "Hello world" + name)
```

SUMMARY

```
num = raw_input("Enter a number: ")
mod = int(num) % 2

if mod > 0:
    print("You picked an odd number.")
else:
    print("You picked an even number.")
```

SUMMARY

```
import random
random_number = random.randint(1, 20)

print('I am thinking of a number between 1 and 20.')

while True:
    string_input = raw_input('>  ')
    guess = int(string_input)
    if guess == 1:
        break
print('You guessed it!')
```

THE END

MPCS 50101
MODULE 2

THE UNIVERSITY OF
CHICAGO