CIS 210

- namespaces/executing functions
  → variable scope
- Boolean data type/conditionals
- Monte Carlo algorithm for approximating pi
- strings – sequences, immutable data types
- indefinite iteration

CIS 210

VARIABLE SCOPE

Recall: Python keeps track of variables using namespaces - directories of names and objects.

When we start Python, two namespaces are created – the built-in namespace and the global (__main__) namespace.

When we create names (e.g., variables, function definitions) in a Python session, they are added to __main__.

CIS 210

VARIABLE SCOPE

When we start Python, two namespaces are created – the built-in namespace and a global (__main__) namespace.

When we create names (e.g., variables, function definitions) in a Python session, they are added to __main__.

Recall: When Python executes a function, a local namespace is created to keep track of function variables.

CIS 210

VARIABLE SCOPE

def twice(x):
    """
    y = 2
    result = y * x
    #print(dir())
    return result
"""

>>> y = 5
>>> y
... 5
>>> twice(y)
... 10
>>> x
... NameError

CIS 210

VARIABLE SCOPE

def twice(x):
    """
    y = 2
    result = y * x
    print(dir())
    return result
"""

>>> y = 5
>>> y
... 5
>>> twice(y)
... 10
['result', 'x', 'y']
>>> x
... NameError
def twice(x):
    result = y * x  # no local y ->
    return result  # find global y
    # do this sparingly!!

>>> y = 5
>>> twice(y)

>>> y  # global again

def thrice(x):
    x += 1
    m = 3
    return m * x

>>> x = 5
>>> thrice(x)

>>> m

CIS 210

SCOPE

Python searches namespaces in this order:

Local, then

Global, then

Built-in

def test1(a):
    a += 5
    return a

>>> a = 6
>>> a = test1(a)

>>> a

def test2(b):
    a += 5
    print(a)
    return None

>>> a = test2(14)

>>> a

CIS 210

SCOPE

def test3(b):
    def test4(a):
        test4(b)
        a += 5
        print(a, b)
        return None
    test4(b)
    b += 1
    return b

>>> test3(99)

>>> b

??
def test3(b):
    test4(b)
    return b + 1

def test4(a):
    a += 5
    print(a, b)
    return None

>>> test3(99)
NameError: name 'b' is not defined

>>> b = 1

>>> test3(99)
local namespaces are on the same level -
104 1
100 static (lexical) scoping

Boolean Expressions/Conditional Statements

if <boolean expression>:
    <block of code>

    <next Python statement>

Flow of control

if <boolean expression>:
    <block of code>
elif <boolean expression>:
    <block of code>
elif <boolean expression>:
    <block of code>
else:
    <block of code>

Boolean expressions: relational and logical operators return Boolean values

a < b             not a < b
a <= b            a <= b and c >= d
a > b             a <= b or c >= d
a <= b            #use Boolean values only
a == b
a != b            #with logical operators
Boolean expressions

- \(a < b\)
- \(a \leq b\)
- \(a > b\)
- \(a \leq b\)
- \(a == b\)
- \(a != b\)

**Order of precedence:**
- Relational operators > logical operators
- Not > and > or

**Short circuit evaluation**

- \(a = 99\)
- \(b = 88\)

```python
if (a < 0) and (b / 0 < 0):
    print('hello')
if (a > 0) or (b / 0 > 0):
    print('hello')
```

**Exploring**

```python
>>> def temp_alert(temp):
    '''(number) -> None
    print information about the temperature
    '''
    if temp >= 90:
        print('hot')
    if temp >= 80:
        print('very warm')
    if temp >= 70:
        print('warm')
    if temp >= 60:
        print('cool')
    return None
```

```python
>>> def = 123
??
```

```python
>>> abs = 456
??
```

```python
>>> abs(-7)
??
```
Exploring

```python
>>> s = 'abc'
>>> if len(s) == 0 or 1:
    print('no')
??

```  

```python
>>> s = ''
>>> if len(s) == 1 or 0:
    print('yes')
??

```  

```python
def temp_alert(temp):
    """(number) -> None
    print information about the temperature"
    if temp >= 90:
        print('hot')
    elif temp >= 80:
        print('very warm')
        >>> temp_alert(90)
    elif temp >= 70:
        print('warm')
    elif temp >= 60:
        print('cool')
    return None

```  

```python
Nifty Python - style

a = -99
b = 88
c = 77
if a < b and b < c:
    if (a < b) and (b < c):
        print('most languages')
        print('better')
if a < b < c:
    print('Python allows chaining of relational operators')

```  

```python
Boolean expressions - style

>>> isinstance(101, int)
True
>>> isinstance(101, str)
False

if isinstance(101, int) == True:
    if isinstance(101, int):

```  

```python
if isinstance:
    <will this code be executed??>

```
Boolean data type (is trickier than you might think for a data type that has only two values)

- Boolean operations on Boolean values only
- Order of operations (use parens for clarity)
- Booleans are not strings
- Boolean short circuit evaluation can lead to hard-to-find errors
- Good style for Boolean expressions
- Double (triple) check Boolean expressions

Monte Carlo Algorithms

- Statistical simulation methods – use sequences of random numbers to perform a simulation
- Any method which solves a problem by generating random numbers and observing that fraction of the numbers obeying some property or properties
- Example of a heuristic technique – guesstimate, approximation - useful when difficult, impossible, or inefficient to use other, more exact, methods

Monte Carlo Simulation to Approximate Pi

- Simulate a game of darts
- Randomly place darts on the board
- Value of pi can be computed by keeping track of the number of darts that land on the board
Monte Carlo Simulation (Problem 3-2)

- the area of the circle is $\pi/4$ and area of square is 1
- the fraction of darts that lands in the circle is $(\pi/4) / 1 = \pi/4$

Monte Carlo Simulation (Problem 3-2)

- the area of the circle is $\pi/4$ and area of square is 1
- the fraction of darts that lands in the circle is $(\pi/4) / 1 = \pi/4$
- the fraction of darts that lands in the circle is $(\text{inCircleCt} / \text{numDarts})$
- $\text{inCircleCt} / \text{numDarts} = \pi / 4 \Rightarrow \pi = 4 \times (\text{inCircleCt} / \text{numDarts})$

- to determine whether a dart has landed in the circle – use formula for finding the distance between the point and the origin: $d = \sqrt{x^2 + y^2}$
- how do we throw darts at the board??

Monte Carlo Simulation (Problem 3-2)

- to generate an approximate value for pi:
  - throw a dart (random.random)
  - test whether it is in the circle (distance formula)
  - keep track of number of darts that land in the circle (accumulator pattern again)
  - approximate pi using $\pi = 4 \times (\text{inCircleCt} / \text{numDarts})$
Monte Carlo Simulation (Problem 3-2)

```python
import random
import math

def montePi(numDarts):
    inCircle = 0  # initialize accumulator variable
    for i in range(numDarts):
        x = random.random()  # throw the darts
        y = random.random()
        d = math.sqrt(x**2 + y**2)  # check location
        if d <= 1:  # it's in the circle
            inCircle += 1  # increment accumulator variable
    pi = inCircle / numDarts * 4  # approximating pi
    return pi
```

Monte Carlo Simulation (Problem 3-2)

```python
import random
import math

def montePi(numDarts):
    inCircle = 0
    for i in range(numDarts):
        x = random.random()
        y = random.random()
        d = math.sqrt(x**2 + y**2)
        if d <= 1:
            inCircle += 1
    pi = inCircle / numDarts * 4
    return pi
```

Monte Carlo Simulation (Problem 3-2 and 3-3)

(0) type in the `montePi` function from the text; add docstring per CIS 210 style guidelines

(1) revise `montePi` so that it calls a new `isInCircle` function

(2) write another new function, `reportPi`, which will be called from `montePi`, to compare the approximate value of pi generated by the Monte Carlo method to the value of `math.pi`, and report on any error in the approximation

(3) add docstring to `showMontePi` starter code (original `montePi + visualization`)

(4) write new function, `drawBoard`, to draw the “dartboard” for the graphical output

(5) revise `showMontePi` so that it calls the new `isInCircle`, `reportPi`, and `drawBoard` functions; add visualization code

CIS 210

**STRINGS**

```python
>>> x = 'PYTHON ROCKS'
```

CIS 210

**Operators**
- Concatenation +
- Repetition *

**Methods**
- upper
- lower
- center
- count
- index
- find
- replace

Strings are sequences of characters.
Operators
Concatenation +
Repetition *
Indexing [
Slicing [:

Note: “overloaded operators”

>>> 99 + 100
>>> 2 * 10

>>> 'hello' + 'goodbye'
>>> 'hello' * 4

CIS 210

Strings

>>> x = 'PYTHON ROCKS'

find method, for example:

>>> str.find(x, 'O')
4

operators: +, *, /, //, %, <, !=, and, in, [], [:], ...

functions: len, round, abs, range, ...

methods: str.replace, str.index, ...

you are all methods
CIS 210

for is a sequential operator

```python
x = "We can't stop for gas, we're already late."
>>> for ch in x:
    print(ch)
??
```

CIS 210

for is a sequential operator

```python
>>> o_ctr = 0
>>> for letter in 'hello':
    if letter == 'o':
        o_ctr += 1
```

CIS 210

Programming/Computer Science concepts

Computational Problem Solving: designing, implementing, checking, revising algorithms/programs.

Good programming style: function docstrings (type contract; description including parameters, returned value, and side effects if any; examples of function use); well-named variables, use of whitespace between operators and sections of code, judicious use of inline comments (why not what).

Python is a programming language and Python is an interpreter (program)

Python Shell is a REPL (read-evaluate-print loop)

Python primitive elements: Objects - value/attributes, type
Combining primitive elements: Expressions - expressions evaluate to a value; short circuit evaluation of boolean expressions; overloaded operators

Naming values: Variables/assignment - assignment statements are not expressions and do not return a value; namespaces – builtins and global (__main__) scope.

Functions are an executable data type; what happens when a function – method – is called:
Activation record/stack frame added to call stack for local namespace; return address
Call-by-assignment parameter passing
Functions always return a value (sometimes None)
Functions sometimes have side effects
Iterative algorithms; accumulator pattern; Monte Carlo algorithms

CIS 210

Python toolkit so far

numeric data types (int, float) and operations (e.g., +, **, round, abs)
string data type and operations (e.g., +, len, count, find)
Boolean data type and operations (e.g., <, and)
NoneType (None)
print
expressions
Python Standard Library – math, turtle, random modules; import
assignment statement
Python repetition – for, while
Python conditionals – if
variable assignment
user-defined functions; function design; docstrings
IDLE interactive development environment; help function

CIS 210

CIS 210 Learning Outcomes

• understand, develop, implement algorithms for computational problem solving;
• use structured design and testing methods to develop and implement programs;
• read, write, revise, document, test, and debug code;
• demonstrate robust mental models of data representation and code execution;
• demonstrate good understanding of a high level programming language;
• introduce and/or implement a sampling of classic computer science problem domains and algorithms.