Python Overview – a closer look, cont’d

✓ Boolean expressions & conditional statements
✓ Monte Carlo algorithms
-- functions – a closer look
   ✓ what happens when a function is called
   -- namespaces and scope
-- Python strings are sequences
-- indefinite loops

Boolean Expressions/Conditional Statements

if <boolean expression>:
   <block of code>
else:
   <block of code>
<next Python statement>

Flow of control

if <boolean expression>:
   <block of code>
else:
   <block of code>
<next Python statement>
CS 210

Boolean expressions

logical/relational operators
return a Boolean value

True
False

CS 210

Boolean expressions

a < b     not a < b
a <= b    a <= b and c >= d
a > b     a <= b or c >= d
a <= b
a == b
a != b

order of precedence:
relational operators > logical
not > and > or – PARENS BEST

CS 210

n = 15
for i in range(1, n+1):
    m3 = (i % 3) == 0
    m5 = (i % 5) == 0
    if m3 and m5:
        print("fizzbuzz")
    else:
        print("fizz")
        print("buzz")
else:
    print(i)
n = 15
for i in range(1, n+1):
m3 = (i % 3) == 0
m5 = (i % 5) == 0
if m3 and m5:
    print('fizzbuzz')
elif m3:
    print('fizz')
elif m5:
    print('buzz')
else:
    print(i)

print('Game over!')

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

>>> type(True)
a) <class 'bool'>
>>> type('True') b) <class 'str'>
>>> type(True) c) NameError: name is not defined
>>> type('false')
>>> type(False)
>>> type(False)
Boolean expressions
lazy evaluation

\[
\begin{align*}
a &= 99 \\
b &= 88 \\
\text{if } (a < 0) \text{ and } (b < 0): \\
&\quad \text{<do something>} \\
\text{if } (a > 0) \text{ or } (b > 0): \\
&\quad \text{<do something>}
\end{align*}
\]

Boolean expressions
lazy evaluation – tricky bugs

\[
\begin{align*}
a &= -99 \\
b &= 88 \\
\text{if } (a < 0) \text{ and } (b / 0 < 0): \\
&\quad \text{<do something>} \\
\text{if } (a > 0) \text{ or } (b / 0 > 0): \\
&\quad \text{<do something>}
\end{align*}
\]

Boolean expressions - style

\[
\begin{align*}
a &= -99 \\
\text{if } (a < 0) == \text{True:} \\
&\quad \text{<do something>} \\
\text{if } a < 0: \\
&\quad \text{<do something>} \\
a &= 99
\end{align*}
\]
Boolean expressions - style

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

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

```python
>>> isInCircle(.5, .5, 1)
True
>>> isInCircle(2, 1, 1)
False
```

```python
if isInCircle(.5, .5, 1) == True:
    if isInCircle(.5, .5, 1):
```

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 (“lazy”) 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 the 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

- Simulation of 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

Figure 2.9

Monte Carlo Simulation (Problem 3-1)

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

Figure 2.10
Monte Carlo Simulation (Problem 3-1)

- the area of the circle is $\pi r^2$ and area of square is 1
- the fraction of darts that lands in the circle is $\frac{\pi r^2}{4} / 1 = \frac{\pi}{4}$
- the fraction of darts that lands in the circle is $(\text{inCircleCt} / \text{numDarts})$
- $\text{inCircleCt} / \text{numDarts} = \frac{\pi}{4}$
- $\pi = 4 \times (\text{inCircleCt} / \text{numDarts})$

Monte Carlo Simulation (Problem 3-1)

- 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

$$\pi = 4 \times \left( \frac{\text{inCircleCt}}{\text{numDarts}} \right)$$

Monte Carlo Simulation (Problem 3-1)

```python
import random
import math

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

Monte Carlo Simulation (Problem 3-1)

```python
import random
import math

def montePi(numDarts):
    inCircle = 0
    for i in range(numDarts):
        x = random.random()  # throw the dart
        y = random.random()
        d = math.sqrt(x**2 + y**2)  # check location
        if d <= 1:  # if it's in the circle
            inCircle += 1
    pi = inCircle / numDarts * 4  # return pi
```
Monte Carlo Simulation (Problem 3-1 and 3-2)

(0) type in the montePi function from the text
(1) add docstring per CIS 210 style guidelines
(2) revise montePi so that it calls a new ismCircle function (exercises 2.38 and 2.39)
(3) add docstring to showMontePi starter code (montePi + visualization); revise showMontePi so that it calls the new ismCircle function
(4) modify showMontePi to report on the error in the approximation of pi produced by the function (as for the approximate square root function) as compared to math.pi

CIS 210

how can we create our own elements?
✓ naming values (variable/assignment)
✓ user-defined functions

For example

```python
def twice(x):
    # header incl.
    # defining a function
    # x is a parameter
    result = x * 2
    return result
```

```python
>>> twice(3)
```

They must be called to execute (run):
```python
>>> twice(3)  # 3 is an argument
```

fn call is an expression
```python
# evaluates to a value
```
When a function is called/executed, Python:

1. evaluates each argument one at a time, working from left to right
2. assigns the resulting values to the function parameters
3. creates a space (activation record) to keep track of function execution – return address and local variables (local namespace)
4. executes the function until return statement
5. stops function execution and returns value specified in return statement
6. the activation record is (eventually) destroyed
7. processing resumes where the function was called

```python
def twice(x):
    x is a parameter
    """(int) -> int
    ""
    result = x * 2
    return result

>>> twice(3)
3 is an argument
>>> twice(5)
5 is an argument
```

parameters (formal parameters) are variable names supplied when the function is defined.

arguments (actual parameters) are the values supplied when the function is called.

Call by assignment parameter passing:
parameter name = argument value when the function is called.
def twice(x):
    result = x * 2
    return result

>>> twice(3)  # x = 3 when twice is executed
>>> twice(5)  # x = 5 when twice is executed

>>>

function calls are expressions
so they evaluate to a value

>>> twice(5)

def twice(x):
    result = 2 * x
    return result

def twice(x):
    result = 2 * x
    return 99

>>> twice(5)

>>> twice(5)  # x = 5 when twice is executed

??

??

>>> twice(5)  # x = 5 when twice is executed

??

>>> twice(5)  # x = 5 when twice is executed

??
Functions ALWAYS return a value
(sometimes the value is None)

Functions SOMETIMES cause a side effect
(a change that persists after the function finishes – for example, something is printed)

```python
def twice(x):
    x = 3
    result = x * 2
    return result

>>> twice(4)
8
>>> twice()

def doubleDouble(x):
    result = twice(x) + twice(x)
    return result

>>> doubleDouble(4)
16
```
functions can be composed

For example,

```python
>>> y = 3.14159
>>> z = abs(round(y))
??
```

Python keeps track of variables using namespaces – a directory of names and objects. *SCOPE* is where the namespace is accessible.

When we start Python, two namespaces are created – the built-in namespace and the global namespace.
VARIABLE SCOPE

Python keeps track of variables using namespaces – a directory of names and objects.

SCOPE is where the namespace is accessible.

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

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

Each time Python executes a function, a local namespace is created.

```
def twice(x):
    # global namespace – func def
    y = 2
    result = y * x
    return result

>>> y = 5  # global namespace - y
>>> twice(y) # local namespace – func exec
>>> y   >>> x   >>> twice # global again
```

```
def twice(x):
    # global namespace
    y = 2; print(y)  # y is local to twice
    result = y * x  # when twice is executing
    return result

>>> y = 5
>>> twice(y)  # local namespace – func exec
>>> y   >>> x   >>> twice # global again
```
Python searches namespaces in this order:

Local, then
Global, then
Built-in

```python
def twice(x):
    # global namespace
    """
    # y = 2; print(y)
    result = y * x
    return result
    # do this sparingly!!
    """

>>> y = 5
>>> twice(y)
    # local namespace — func exec
>>> y  # global again
```

```python
def twice(x):
    """
    x += 1  # x is local to twice
    m = 2
    result = m * x
    return result
    """

>>> x = 5
>>> twice(x)
```

---

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, format)
- Boolean data types and operations (e.g., , and)
- 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
Programming/Computer Science concepts

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

Good programming style: function doctstrings (type contract; description including parameters, return 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 primitive elements: Objects - value/attributes, operations, memory location (id).

Combining primitive elements: Expressions - expressions evaluate to a value.

Naming values: Variables/assignment - assignment statements are not expressions and do not return a value; namespace, variable scope.

User-defined functions - functions always return a value (sometimes None); functions sometimes have side effects.

What happens when a function is called?

Activation record/task frame added to call stack for local namespace; return address

Call-by-assignment parameter passing
Side-effects/mutating a value

Iterative algorithms, accumulator pattern
Monte Carlo algorithms

CIS 210 Learning Outcomes

• understand, develop, implement, and 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.