## Python Overview – a closer look, cont’d

-- Boolean expressions & conditional statements
-- Python strings are sequences
-- indefinite loops

-- functions – a closer look
   -- what happens when a function is called
   -- namespaces and scope

### Monte Carlo algorithms

<table>
<thead>
<tr>
<th>CIS 210</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Python Overview – a closer look, cont’d</strong></td>
</tr>
<tr>
<td>a) n = 4 d) &gt;&gt;&gt; x = 10</td>
</tr>
<tr>
<td>x = 1 &gt;&gt;&gt; x = 'hi'</td>
</tr>
<tr>
<td>for _ in range(20): ______________________________</td>
</tr>
<tr>
<td>approx rt = .5 * (x + n/x) ______________________</td>
</tr>
<tr>
<td>print(approx rt) ________________________________</td>
</tr>
<tr>
<td>1) dynamic typing</td>
</tr>
<tr>
<td>b) &gt;&gt;&gt; 'testing' + 123</td>
</tr>
<tr>
<td>TypeError: must be str, not int</td>
</tr>
<tr>
<td>2) operator overloading</td>
</tr>
<tr>
<td>c) &gt;&gt;&gt; 1 + 123</td>
</tr>
<tr>
<td>124</td>
</tr>
<tr>
<td>3) strong typing</td>
</tr>
<tr>
<td>&gt;&gt;&gt; 'testing' + '123'</td>
</tr>
<tr>
<td>'testing123'</td>
</tr>
<tr>
<td>4) dynamic typing (error)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CIS 210</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>n = 15</strong></td>
</tr>
<tr>
<td>for i in range(1, n+1):</td>
</tr>
<tr>
<td>m3 = (i % 3) == 0</td>
</tr>
<tr>
<td>m5 = (i % 5) == 0</td>
</tr>
<tr>
<td>if m3 and m5:</td>
</tr>
<tr>
<td>print('fizzbuzz')</td>
</tr>
<tr>
<td>if m3:</td>
</tr>
<tr>
<td>print('fizz')</td>
</tr>
<tr>
<td>if m5:</td>
</tr>
<tr>
<td>print('buzz')</td>
</tr>
<tr>
<td>else:</td>
</tr>
<tr>
<td>print(i)</td>
</tr>
<tr>
<td>print('Game over!')</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CIS 210</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>n = 15</strong></td>
</tr>
<tr>
<td>for i in range(1, n+1):</td>
</tr>
<tr>
<td>m3 = (i % 3) == 0</td>
</tr>
<tr>
<td>m5 = (i % 5) == 0</td>
</tr>
<tr>
<td>if m3 and m5:</td>
</tr>
<tr>
<td>print('fizzbuzz')</td>
</tr>
<tr>
<td><strong>elif</strong> m3:</td>
</tr>
<tr>
<td>print('fizz')</td>
</tr>
<tr>
<td><strong>elif</strong> m5:</td>
</tr>
<tr>
<td>print('buzz')</td>
</tr>
<tr>
<td>else:</td>
</tr>
<tr>
<td>print(i)</td>
</tr>
<tr>
<td>print('Game over!')</td>
</tr>
</tbody>
</table>
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>

<next Python statement>

Boolean expressions

logical/relational operators

return a Boolean value

    True
    False

Boolean expressions: relational and logical operators return Boolean values

    a < b
    a <= b
    a > b
    a <= b
    a == b
    a != b

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

#use logical operators with #Boolean values only!
Boolean expressions

- \( a < b \)
- \( a <= b \) \((a <= b) \text{ and } (c >= d)\)
- \( a > b \) \((a <= b) \text{ or } (c >= d)\)
- \( a <= b \)
- \( a == b \)
- \( a != b \)

**Order of precedence:**

- relational operators > logical
- not > and > or – PARENS BEST

---

**Short circuit evaluation**

\[ a = 99 \]
\[ b = 88 \]

if \((a < 0) \text{ and } (b < 0)\):
    
    \(<\text{do something}>\)

if \((a > 0) \text{ or } (b > 0)\):
    
    \(<\text{do something}>\)
Boolean expressions
short circuit evaluation – **tricky bugs**

```python
a = -99
b = 88

if (a < 0) and (b / 0 < 0):
    <do something>

if (a > 0) or (b / 0 > 0):
    <do something>
```

---

Boolean expressions - **style**

```python
a = -99

if (a < 0) == True:
    <do something>

if a < 0:
    <do something>
```

---

Boolean expressions - **style**

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

    if isinstance(101, int) == True:
        <do something>  

    if isinstance(101, int):

    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

Strings are sequences of characters.

Operators
- Concatenation +
- Repetition *

Methods
- upper
- lower
- center
- count
- index
- find
- replace
CIS 210

<table>
<thead>
<tr>
<th>Positive indexes</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>P</td>
<td>Y</td>
<td>T</td>
<td>H</td>
<td>O</td>
<td>N</td>
<td>R</td>
<td>O</td>
<td>C</td>
<td>K</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>Negative indexes</td>
<td>-12</td>
<td>-11</td>
<td>-10</td>
<td>-9</td>
<td>-8</td>
<td>-7</td>
<td>-6</td>
<td>-5</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
</tr>
</tbody>
</table>

```python
>>> len(x)
??
>>> x[4]   >>> x[-1]
??        ??
>>> x[1:4] >>> x * 3
??        ??
>>> x + 'yes'
??
```

CIS 210

**STRINGS**

```python
>>> x = 'Python rocks'

find method, for example:

```python
>>> str.find(x, 'o')
4
>>> x.find('o')
4
```

```python
>>> str.find('hello', 'e')
1
>>> int.__add__(4, 3)
7
>>> str.__len__('abcd')
4
>>> 'abcd'.__len__()
```

CIS 210

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

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

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

they are all methods
for is a sequential operator

x = "We can't stop for gas, we're already late."

```python
>>> for ch in x:
    print(ch)

W e c a
```

strings are immutable sequences

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

```

```python
>>> s = 'jello, world'
>>> s[0] = 'h'

```

```python
```
```
strings are immutable sequences

```python
globals() = {'s': 'jello, world', 't': None, 'i': 0}
s[0] = 'h'
```

```
>>> s = 'jello, world'
>>> s[0] = 'h'
TypeError: 'str' object does not support item assignment
```

```python
globals() = {'s': 'jello, world', 't': None, 'i': 0}
s[0] = 'h'
```

```
>>> s = 'hello, world'
```

```
>>> s = 'hello, world'
```

```
?? – same or different?  ?? same or different?
```
indefinite iteration
most general type of loop

while <boolean expression>:
  statement1
  statement2
  ...
  statement

p = 10
i = 1
ctr = 1  # initialize loop variable
while ctr <= p:
  i = i * 2  # check end condition
  ctr += 1  # move loop var toward
print(i)  # the end condition
```python
CIS 210
while loop
print('Enter numbers you wish to add, quit to end.')
sum = 0
next = input('next: ')
while next != 'quit':
    sum += int(next)
    next = input('next: ')
print('Sum is:', sum)

can't specify ahead of time how many times the loop will run
```

```python
CIS 210
Functions – a closer look
Recall: how can we create our own elements?
✓ naming values (variables/assignment)
  – user-defined functions

For example
def twice(x):
    """ (int) -> int """
    result = x * 2
    return result
```

```python
CIS 210
how can we create our own elements?
✓ naming values (variables/assignment)
  – user-defined functions

For example
def twice(x):
    """ (int) -> int """
    result = x * 2
    return result
```

```python
CIS 210
Every docstring of every function should include:
• type contract provides type of each parameter and the type of the value returned by the function
• brief description that mentions each parameter by name
• side effects (e.g., print), if any
• returned value
• simple examples of use
• calls/called by (if any, if helpful)
```
reflects thoughtful design
→ contributes to usability/maintainability of code
→ integrated with Python help function
→ automated testing

“One of the characteristics of a well-written function is the ability to read the code [including documentation] and see the underlying algorithm.”

Defining a function is like defining a variable name –
the function name refers to the function value (the body of the function).

Functions are an executable data type.

```python
>>> def twice(x):
    '''
    # defining a function
    # x is a parameter
    result = x * 2
    return result

>>> twice
<function twice at 0x105b19b70>
```

They must be called to execute (run):
```python
def twice(x):
    # defining a function
    # x is a parameter
    result = x * 2
    return result

>>> twice(3)
# 3 is an argument
6
# fn call is an expression
# 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

---

def twice(x):
    """(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
    ***
    >>> twice(3)  # function calls are expressions
    function calls are expressions
    6
    >>> twice(5)  # so they evaluate to a value
    10
    def twice(x):
        result = 2 * x
        result
        return None
        >>> twice(5)
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)

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

>>> twice(4)
??

>>> twice(5)
??

>>> twice()  
??

>>> twice
??
functions can be composed

For example,

```python
>>> y = 4.5
>>> z = abs(round(y))
```

Example for Python Visualizer

def is_odd(y):
    """
    return not(y % 2) == 0
    """
def foo(x):
    """
    print(id(x))
    x += 1
    print(id(x))
    if is_odd(x):
        return x
    else:
        return x * 2
foo(10)
x = foo(13)

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

- 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 3-1)

- 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: $d = \text{math.sqrt}(x**2 + y**2)$
- how do we throw darts at the board??
Monte Carlo Simulation (Problem 3-1)

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

\[ \pi = 4 \times \left( \frac{\text{inCircleCt}}{\text{numDarts}} \right) \]
Monte Carlo Simulation (Problem 3-1 and 3-2)

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

(1) revise `montePi` so that it calls a new `isinCircle` function (exercises 2.38 and 2.39)

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

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

(4) revise `showMontePi` so that it calls the new `isinCircle` and `drawBoard` functions

(5) write new function, `reportPi`, which will be called from `showMontePi`, 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

(6) modify `showMontePi` to call `reportPi`

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, format)
- Boolean data type and operations (e.g., <, and)
- `NoneType` (None)
- 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

Programming/Computer Science concepts

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

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 primitive elements: Objects - value/attributes, operations, memory location (id).
- Combining primitive elements: Expressions - expressions evaluate to a value; overloaded operators; methods/functions/operators; short circuit evaluation of boolean expressions
- Naming values: Variables/assignment - assignment statements are not expressions and do not return a value; namespace; variable scope; dynamic typing; strong typing.
- User-defined functions – functions always return a value (sometimes None); functions sometimes have side effects.

What happens when a function is called?
- Activation record/stack frame added to call stack for local namespace; return address
- Call-by-assignment parameter passing
- Side effects/returning a value

Iterative algorithms; accumulator pattern
- Monte Carlo algorithms

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.