A closer look:
• Python objects
• Python assignment

-- mental model of computational processes
-- for reading/writing more advanced code
-- for analyzing (preventing!) errors

Python overview – a closer look

Recall:
• what are Python’s primitive elements?

There is only one kind of primitive element in Python

OBJECTS

✓ what are the primitive elements?
-- object, with
  -- type that determines range of values and operations (attributes)
  -- value(s) of the object
  -- memory location
Recall: Objects can be named

For example,

```python
>>> x = 10
```

Python processes an assignment statement by:

1. evaluating the expression on the rhs
2. associating name on lhs with resulting value

```
>>> b = 20
>>> b = 30
>>> b = b + 1
```

1. evaluating the expression on the rhs
2. associating name on lhs with resulting value

```
>>> b = 20
>>> a = b + 1
```

1. evaluating the expression on the rhs
2. associating name on lhs with resulting value
1. evaluating the expression on the rhs
2. associating name on lhs with resulting value

Let the variable on lhs refer to resulting object:
- allocate space in memory for the object
- search current namespace – if name on lhs is not there:
  - assign name on lhs to address of memory location
  - add it to the current namespace
- if name on lhs is there:
  - replace old reference with new reference (address)

---

```python
>>> b = 20
reference semantics:
>>> id(b)
4297645024
the name on the lhs is
a reference (pointer)
to the memory location
of the data object
>>> b = 30
>>> id(b)
4297645344
>>> b = b + 1
>>> id(b)
4297645376
```

```python
>>> b = 20
reference semantics:
>>> id(b)
4297645024
the name on the lhs is
a reference (pointer)
to the memory location
of the data object
>>> a = b + 1
>>> id(a)
4297645056
>>> id(a) == id(b)
False
>>> a
>>> b
??
??
```

```python
>>> b = 99.9
>>> a = b
>>> a
reference semantics:
the name on the lhs is
a reference (pointer)
to the memory location
of the data object
>>> b = 30
>>> id(b)
4297645344
>>> a
>>> b
??
??
```

Evaluate rhs. If it is a variable name:
Let the variable on lhs refer to resulting object:
- use address of rhs variable (no memory is allocated)
- search current namespace – if name on lhs is not there:
  - assign name on lhs to address of memory location
  - add it to the current namespace
- if name on lhs is there:
  - replace old reference with new reference (address)
CIS 210

>>> b = 99.9
>>> id(b) 4302378376
>>> a = b
>>> id(a) 4302378376
>>> id(b) = id(a)
>>> True
>>> a ??
>>> b ??

CIS 210

>>> b = 99.9
>>> id(b) 4302378376
>>> a = b
>>> id(a) 4302378376
>>> b = 88.8
>>> id(b) 4383118984
>>> id(a) 4383118984
>>> a ??
>>> b ??

CIS 210

>>> msg = 'hello'
>>> msg 'hello'
>>> greeting = msg
>>> id(msg) 4383118984
>>> id(greeting) greeting “aliases” msg

CIS 210

>>> msg = 'goodbye'
>>> msg 'goodbye'
>>> id(msg) 4383119544
>>> id(greeting) greeting “aliases” msg

CIS 210

>>> id(msg) 4383119544
>>> id(greeting) 4383118984
>>> greeting ??

CIS 210

reference semantics:
the name on the lhs is a reference (pointer)
to the memory location of the data object

CIS 210

reference semantics:
the name on the lhs is a reference (pointer)
to the memory location of the data object

CIS 210

reference semantics:
the name on the lhs is a reference (pointer)
to the memory location of the data object

CIS 210

reference semantics:
the name on the lhs is a reference (pointer)
to the memory location of the data object
WHAT IF ...

>>> msg[0] = 'j'
str were a mutable type?

>>> msg
'jello'

when the variable is given a new value

>>> id(msg)
4383118984

the reference (pointer) is updated

>>> id(greeting)
4383118984

>>> greeting

>>>

> msg = 'hello'

>>> msg = 'jello'

>>> class = 'CIS 210'

>>> class = 'CIS 211'

garbage collection

- Python interpreter

- check whether any variables are pointing at objects for which it has allocated memory

- if none are, the object is deleted and the memory is made available again

Recall:

>>> twice
<function twice at 0x105b19b70>

>>> twice
<function twice at 0x105b19b70>

>>>

int(0x105b19b70)
4390493040
Everything we do on a computer is an abstraction of a binary representation. Binary sequences are used to represent all digital data. Therefore, understanding binary representations of data is crucial for computer science. Relevant topics include limitations of data types (e.g., floating point precision), computer organization, security, and many others.

#### Binary representation of numbers

Recall:

\[ 469_{10} = (4 \times 10^2) + (6 \times 10^1) + (9 \times 10^0) \]
**Binary representation of numbers**

10/18/18

**Binary to decimal:**

\[ 469_{10} = (4 \times 10^2) + (6 \times 10^1) + (9 \times 10^0) \]

**Decimal to binary:**

\[ 111010101_2 = (1 \times 2^8) + (1 \times 2^7) + (1 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \]

- Look at each bit in the binary number, \( b \).
- Multiply it by the correct power of 2.
- Sum the results.

**Binary to decimal:**

\[ 111010101_2 = (1 \times 2^8) + (1 \times 2^7) + (1 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \]

- Look at each bit in the binary number, \( b \).
- Multiply it by the correct power of 2.
- Sum the results.

**Decimal to binary:**

\[ 469 \div 2 = q234 r1 \rightarrow \text{rightmost bit} \]

\[ 234 \div 2 = q117 r0 \rightarrow \text{next bit} \]

\[ 117 \div 2 = q58 r1 \rightarrow \text{next bit} \]

\[ 58 \div 2 = q29 r0 \rightarrow \text{next bit} \]

\[ 29 \div 2 = q14 r1 \rightarrow \text{next bit} \]

\[ 14 \div 2 = q7 r0 \rightarrow \text{next bit} \]

\[ 7 \div 2 = q3 r1 \rightarrow \text{next bit} \]

\[ 3 \div 2 = q1 r1 \rightarrow \text{next bit} \]

\[ 1 \div 2 = q0 r1 \rightarrow \text{leftmost bit} \]

The first quotient is the decimal number to be converted while quotient is greater than 0.

- Divide the decimal number by 2.
- Make the remainder the next digit to the left in the answer.
- Update the quotient.
### Hexadecimal representation of numbers

\[ 111010101_2 = (1 \times 2^8) + (1 \times 2^7) + (1 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \]

**Why hexadecimal?**

Groupings of bits are easier to read and write.

\[ \begin{align*}
(1) & \quad (1101) \quad (0101) \\
\text{binary} & \quad \text{hexadecimal}
\end{align*} \]

### Hexadecimal representation of numbers

\[ 1c5_{16} = (1 \times 16^2) + (13 \times 16^1) + (5 \times 16^0) \]

**Why hexadecimal?**

Groupings of 4 bits are easier to read and write.

\[ \begin{align*}
(1) & \quad (1101) \quad (0101) \\
\text{binary} & \quad \text{hexadecimal}
\end{align*} \]

### characters are also represented using binary ASCII -> UTF-8 standards

```python
>>> ord('Z')
90

>>> ord('a')
97
```

```python
>>> for i in range(91, 97):
    print(chr(i))
```

### binary representation of characters - ASCII

```python
>>> ord('Z')  >>> chr(90)
90  'Z'

>>> ord('a')  >>> chr(0b1100001)
97  'a'
```
binary representation of characters

```python
>>> ord('Z')
90
>>> ord('a')
97
>>> 'a' < 'Z'
False
>>> 'a' < 'Z'.lower()
True
```

binary representation of characters
UTF-8

```python
>>> '\u03a9'
'Ω'
>>> '\u2877'
'⡷'
```

Assignment statements
Python is a dynamically typed language

```python
>>> a = 10
>>> type(a)
??
>>> b = a
>>> type(b)
??
```

Assignment statements

```python
Python is a dynamically typed language

```python
>>> a = 10
>>> a = 'hello, world'
>>> type(a)
??
>>> b = a
>>> type(b)
??
```

```python
static typing

```python
var a : int
a = 4
a = 'hello' X
```
CIS 210
Static, Dynamic – for example, scope, type

static – can be determined by reading code (only)

dynamic – scope/type is determined when code executes

CIS 210
Assignment statements
dynamic typing

greenCt = 1
for ctr in range(4):
    greenCt += 1
print(greenCt)

CIS 210
Assignment statements
dynamic typing

greenCt = 1
for ctr in range(4):
    greenCt = greenCt + 1  #greenCt += 1
print(greenCt)

accumulator pattern

CIS 210
Recall: we can combine objects in expressions, which are evaluated and return a value

For example,
>>> 99 + 10
??
>>> len('hello')
??
>>> str.center('****', 10)
??

Python is a strongly typed language

CIS 210
Recall: we can combine objects in expressions, which are evaluated and return a value

For example,
>>> len(10)
??
>>> str.center(math.pi, 10)
??

Python is a strongly typed language
Recall: **expressions** are combinations of **values** (operands) and **operators**, that can be **evaluated** and **return a result**

For example,

```python
>>> 99.9 + 100
```

*Python is an extremely strongly typed language*

def check(a: int, b: str) -> None:
    """(int, str) -> None"
    print(a, b)
    return None

```python
>>> check('hello', 99)  # needs interpreter support
hello 99
```

def check(a: int, b: str, c: string) -> None:
    """
    (int, str) -> None
    
    print(a, b, c)
    return None
    """

Traceback (most recent call last):
  File "/Users/kfreeman/Documents/cis210W18/projects-W18/check.py", line 1, in <module>  
def check(a:int, b:str, c:string):NameError: name 'string' is not defined

type: 1) dynamic typing
  d) n = 4
  >>> x = 1
  >>> x = -99 < 0
  for _ in range(20):
    approx_rt = .5 * (x + n/x)
    print(approx_rt)

b) >>> 'testing' + 123
TypeError: must be str, not int

c) >>> 1 + 123
124

2) operator overloading

3) strong typing

4) dynamic typing (error)

---

**Figure 3.2**

[Diagram of symmetric key algorithm]

**Figure 3.5**

[Diagram of symmetric key algorithm]
CS 210

CIS 210

Python/programming 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)
- Booleans data type and operations (e.g., <, and)
- NoneType (None)
- print/input 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
- hand-tracing code
- program function diagrams

CS 210

Programming/Computer Science concepts

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

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, 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; namespaces – builtins and global (__main__); scope; dynamic typing; strong typing; reference semantics.

Functions are an executable data type; 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
- Functions always return a value (sometimes None)
- Functions sometimes have side effects

Iterative algorithms; accumulator pattern

CS 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.