Python Introduction II
Week 2

Sanja Damjanovic

CPME 180A DSA in Python

07/22/2025

Week 2 Outline

- Introduction to Python II:
  - Control flow recap
  - Python built-in datatypes recap
  - Shallow and deep copying
  - Comprehension sytax
  - String formatting
  - Naming convention
  - Namespaces
  - Functions
  - Algorithm analysis and Big-O notation
  - Object-oriented programming
- Week 2 lab

Week 2 Intro Quiz

- Q1. Select all proper variable names: A) good = 4. B) bAd = 5. C) print = 3 D) 1loop =1
- Q2: Match: 3//4, 3/4 and 3%4 to: 0.75 or 3/4
- Q3: Is it possible to run into an overflow error in Python e.g. if calculating 10**100? Yes/no
- Q4: Select all immutable Python data types: A) list, B) int, C) str, D) set
- Q5: What does the code evaluates to?
  - foo = 17
  - bool(foo)
  - A) foo, B) 17, C)True, D) False

List Slicing

- A = [1,6,3,4,5,2,7]

```
a1 = A[2:4]
a2 = A[2:]
a3 = A[:4]
a4 = A[:-1]
a5 = A[-3:]
a6 = A[-3:-1]
a7 = A[1:5:2]
a8 =A[5:1:-2]
a9 = A[::-1]
```

String Slicing

- S=‘Python3’

```
s1 = S[2:4]
s2 = S[2:]
s3 = S[:4]
s4 = S[:-1]
s5 = S[-3:]
s6 = S[-3:-1]
s7 = S[1:5:2]
s8 = S[5:1:-2]
s9 = S[::-1]
```

Python Resources

- Python documentation: https://www.python.org/doc/
- Python language reference: https://docs.python.org/3/reference/
- Python tutorials: https://docs.python.org/3/tutorial/

Week 2 Reading

- [1] Chapter 2, 3, 4
- [2] Chapter 2, 4

[1] Problem Solving with Algorithms and Data Structures using Python, by Brad Miller and David Ranum, Luther College, freely available online via link [1]

[2] Data Structures and Algorithms in Python, by Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, available via SJSU library via link [2]

Python Built-in Data Types (Classes)

- Atomic data types: int, float, bool
- Relational and Logical Operators

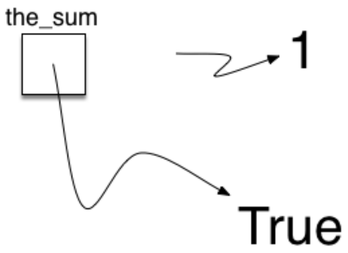
Operation Name	Operator	Explanation
less than	$<$	Less than operator
greater than	$>$	Greater than operator
less than or equal	$<=$	Less than or equal to operator
greater than or equal	$>=$	Greater than or equal to operator
equal	$==$	Equality operator
not equal	$!=$	Not equal operator
logical and	and	Both operands True for result to be True
logical or	or	One or the other operand is True for the result to be True
logical not	not	Negates the truth value, False becomes True, True becomes False

Python variable and reference

- A Python variable is created when a name is used for the first time on the left-hand side of an assignment statement
- Variables hold references to data objects


```
>>> the_sum = 0
>>> the_sum
0
>>> the_sum = the_sum + 1
>>> the_sum
1
>>> the_sum = True
>>> the_sum
True
```

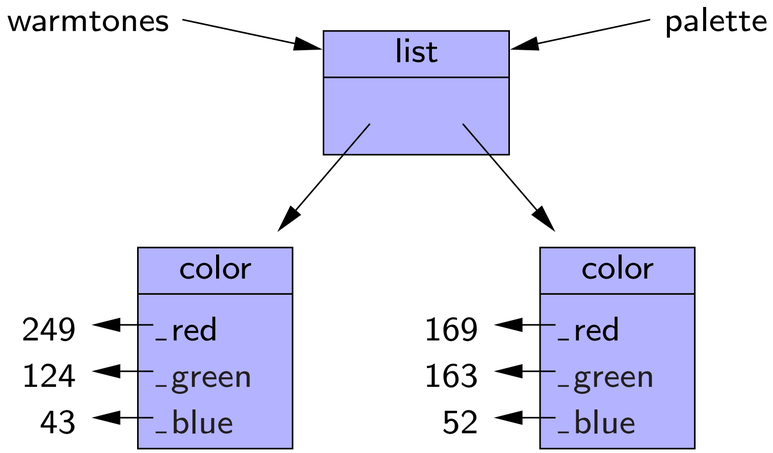
- Assignment changes the reference



Alias

- Two aliases for the same list of colors:
  - palette = warmtones

.



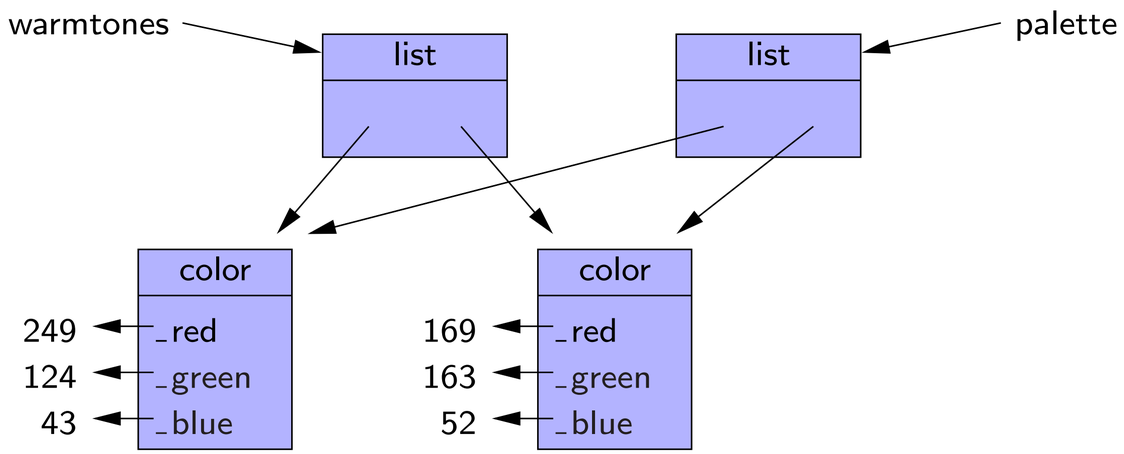
Shallow & Deep Copy

Shallow Copy

- palette = list(warmtones)

- the new list represents a sequence of references to the same elements as in the first

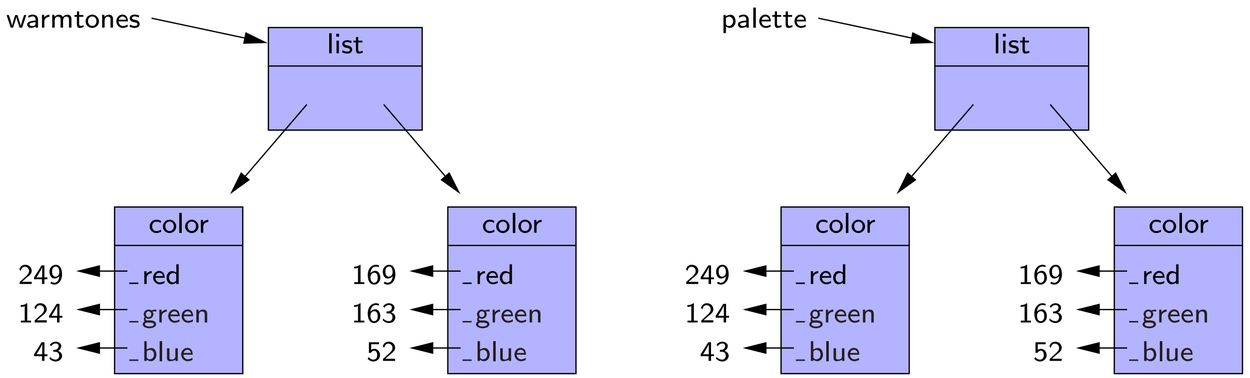
A shallow copy of a list of colors



Deep Copy

- Import copy
- Function **copy**: shallow copying
- Function deep copy: deep copying
  - palette = copy.**deepcopy**(warmtones)

.



Built-in Collection Data Types

Built-in Collection Data Types

- Sequences:
  - List
  - Strings
  - Tuples
- Containers:
  - Sets, frozen sets
  - Dictionaries

Operations on Any Sequence in Python

Lists, strings, tupples

Operation Name	Operator	Explanation
indexing	[ ]	Access an element of a sequence
concatenation	+	Combine sequences together
repetition	*	Concatenate a repeated number of times
membership	in	Ask whether an item is in a sequence
length	len	Ask the number of items in the sequence
slicing	[:]	Extract a part of a sequence

Lists

- an ordered collection of zero or more references to Python data objects
- comma-delimited values enclosed in square brackets
- The empty list is simply [ ]
- Lists are heterogeneous
- Mutable

```
>>> [1, 3, True, 6.5]
[1, 3, True, 6.5]
>>> my_list = [1, 3, True, 6.5]
>>> my_list
[1, 3, True, 6.5]
```

Lists

Methods

Method Name	Use	Explanation
`append`	`a_list.append(item)`	Adds a new item to the end of a list
`insert`	`a_list.insert(i,item)`	Inserts an item at the ith position in a list
`pop`	`a_list.pop()`	Removes and returns the last item in a list
`pop`	`a_list.pop(i)`	Removes and returns the ith item in a list
`sort`	`a_list.sort()`	Sorts a list in place
`reverse`	`a_list.reverse()`	Modifies a list to be in reverse order
`del`	`del a_list[i]`	Deletes the item in the ith position
`index`	`a_list.index(item)`	Returns the index of the first occurrence of `item`
`count`	`a_list.count(item)`	Returns the number of occurrences of `item`
`remove`	`a_list.remove(item)`	Removes the first occurrence of `item`

Conditional Expressions

- Python supports a conditional expression syntax that can replace a simple control structure.
- The general syntax is an expression of the form:

expr1 **if** *condition* **else** *expr2*

- This compound expression evaluates to expr1 if the condition is true, and otherwise evaluates to expr2.
- For example:

```
param = n if n >= 0 else -n      # pick the appropriate value
result = foo(param)              # call the function
```

- Or even

```
result = foo(n if n >= 0 else -n)
```

Comprehension Syntax

- A very common programming task is to produce one series of values based upon the processing of another series.
- Often, this task can be accomplished quite simply in Python using what is known as a comprehension syntax.

```
[ expression for value in iterable if condition ]
```

- This is the same as

```
result = [ ]
for value in iterable:
  if condition:
    result.append(expression)
```

Comprehension Syntax

Comprehension Syntax: List

```
squares = [ ]
for k in range(1, n+1):
  squares.append(k*k)
```

```
squares = [k*k for k in range(1, n+1)]
```

Comprehension Syntax

```
[ k*k for k in range(1, n+1) ]
{ k*k for k in range(1, n+1) }
( k*k for k in range(1, n+1) )
{ k : k*k for k in range(1, n+1) }
```

```
[ k*k for k in range(1, n+1) ]          list comprehension
{ k*k for k in range(1, n+1) }          set comprehension
( k*k for k in range(1, n+1) )          generator comprehension
{ k : k*k for k in range(1, n+1) }      dictionary comprehension
```

Packing

- If a series of comma-separated expressions are given in a larger context, they will be treated as a single tuple, even if no enclosing parentheses are provided.
- For example, consider the assignment

$$\text{data} = 2, 4, 6, 8$$

- This results in identifier, data, being assigned to the tuple (2, 4, 6, 8). This behavior is called **automatic packing** of a tuple.

Unpacking

- As a dual to the packing behavior, Python can automatically unpack a sequence, allowing one to assign a series of individual identifiers to the elements of sequence.
- As an example, we can write

```
a, b, c, d = range(7, 11)
```

- This has the effect of assigning a=7, b=8, c=9, and d=10.

Swapping Values

- a=5, b=6
- Swap: a,b = b,a

Strings

Strings

- sequential collections of *characters i.e.* zero or more letters, numbers, and other symbols
- Literal string values are differentiated from identifiers by using quotation marks (either single or double)
- immutable

Strings

Searching for Substrings

Calling Syntax	Description
s.count(pattern)	Return the number of non-overlapping occurrences of pattern
s.find(pattern)	Return the index starting the leftmost occurrence of pattern; else -1
s.index(pattern)	Similar to find, but raise ValueError if not found
s.rfind(pattern)	Return the index starting the rightmost occurrence of pattern; else -1
s.rindex(pattern)	Similar to rfind, but raise ValueError if not found

Strings

String methods that produce related strings

Calling Syntax	Description
s.replace(old, new)	Return a copy of s with all occurrences of old replaced by new
s.capitalize( )	Return a copy of s with its first character having uppercase
s.upper( )	Return a copy of s with all alphabetic characters in uppercase
s.lower( )	Return a copy of s with all alphabetic characters in lowercase
s.center(width)	Return a copy of s, padded to width, centered among spaces
s.ljust(width)	Return a copy of s, padded to width with trailing spaces
s.rjust(width)	Return a copy of s, padded to width with leading spaces
s.zfill(width)	Return a copy of s, padded to width with leading zeros
s.strip( )	Return a copy of s, with leading and trailing whitespace removed
s.lstrip( )	Return a copy of s, with leading whitespace removed
s.rstrip( )	Return a copy of s, with trailing whitespace removed

Strings

Methods for splitting and joining strings

Calling Syntax	Description
sep.join(strings)	Return the composition of the given sequence of strings, inserting sep as delimiter between each pair
s.splitlines( )	Return a list of substrings of s, as delimited by newlines
s.split(sep, count)	Return a list of substrings of s, as delimited by the first count occurrences of sep. If count is not specified, split on all occurrences. If sep is not specified, use whitespace as delimiter.
s.rsplit(sep, count)	Similar to split, but using the rightmost occurrences of sep
s.partition(sep)	Return (head, sep, tail) such that s = head + sep + tail, using leftmost occurrence of sep, if any; else return (s, '', '')
s.rpartition(sep)	Return (head, sep, tail) such that s = head + sep + tail, using rightmost occurrence of sep, if any; else return ('', '', s)

String Formatting Conversion Characters

Character	Output Format
`d`, `i`	Integer
`u`	Unsigned integer
`f`	Floating point as m.ddddd
`e`	Floating point as m.dddddе+/-xx
`E`	Floating point as m.dddddE+/-xx
`g`	Use `%e` for exponents less than -4 or greater than $+5$, otherwise use `%f`
`c`	Single character
`s`	String, or any Python data object that can be converted to a string by using the `str` function
`%`	Insert a literal `%` character

String Formating

Additional Formatting Options

Modifier	Example	Description
number	`%20d`	Put the value in a field width of 20
`-`	`%-20d`	Put the value in a field 20 characters wide, left-justified
`+`	`%+20d`	Put the value in a field 20 characters wide, right-justified
`0`	`%020d`	Put the value in a field 20 characters wide, fill in with leading zeros
`.`	`%20.2f`	Put the value in a field 20 characters wide with 2 characters to the right of the decimal point
`(name)`	`%(name)d`	Get the value from the supplied dictionary using `name` as the key

String Formatting Example

```
>>> price = 24
>>> item = "banana"
>>> print("The %s costs %d cents" % (item, price))
The banana costs 24 cents
>>> print("The %+10s costs %5.2f cents" % (item, price))
The     banana costs 24.00 cents
>>> print("The %+10s costs %10.2f cents" % (item, price))
The     banana costs      24.00 cents
>>> itemdict = {"item": "banana", "cost": 24}
>>> print("The %(item)s costs %(cost)7.1f cents" % itemdict)
The banana costs    24.0 cents
```

The str.format( ) method:

```
>>> print("The {} costs {} cents".format(item, price))
The banana costs 24 cents
>>> print("The {:s} costs {:d} cents".format(item, price))
The banana costs 24 cents
```

Formatted String Literals

f-strings

- Python doc: https://docs.python.org/3/tutorial/inputoutput.html
- begin a string with **f** or **F** before the opening quotation mark or triple quotation mark
- Inside this string, write a Python expression between **{** and **}** characters that can refer to variables or literal values.

f-String Formating

Modifier	Example	Description
number	`:20d`	Put the value in a field width of 20
`<`	`:<20d`	Put the value in a field 20 characters wide, left-aligned
`>`	`:>20d`	Put the value in a field 20 characters wide, right-aligned
`^`	`:^20d`	Put the value in a field 20 characters wide, center-aligned
`0`	`:020d`	Put the value in a field 20 characters wide, fill in with leading zeros.
`.`	`:20.2f`	Put the value in a field 20 characters wide with 2 characters to the right of the decimal point.

f-String Formatting Example

: [[fill]align][sign][prefix] [0][width][thousands sep: , or _][.precision][type]

```
>>> print(f"The {item:10} costs {price:10.2f} cents")
The banana     costs      24.00 cents
>>> print(f"The {item:<10} costs {price:<10.2f} cents")
The banana     costs 24.00      cents
>>> print(f"The {item:^10} costs {price:^10.2f} cents")
The   banana   costs   24.00    cents
>>> print(f"The {item:>10} costs {price:>10.2f} cents")
The     banana costs      24.00 cents
>>> print(f"The {item:>10} costs {price:>010.2f} cents")
The     banana costs 0000024.00 cents
>>> itemdict = {"item": "banana", "price": 24}
>>> print(f"Item:{itemdict['item']:.>10}\n" +
... f"Price:{'$':.>4}{itemdict['price']:5.2f}")
Item:....banana
Price:...$24.00
```

Prefix: 0b, 0o, 0x
Type: s, d, f

Q: String Formatting

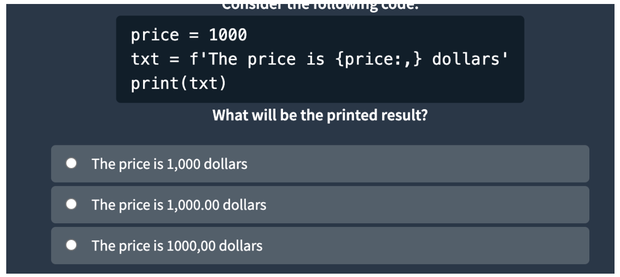
- 1. What placeholders are used when dealing with f-strings? A) (), B) [], C) {}
- 2. Use the `price` variable and f-string syntax to display the price with two decimals:

```
price = 59
txt = f"The price is {          } dollars"
print(txt)
```

- 3. Use an `if` statement inside the f-string placeholders to return 'perfect' if the price is 100, and 'ok' if the price is not 100:

```
price = 100
txt = f"It is {'perfect'                    'ok'}"
print(txt)
```

- 4.



A: String Formatting

- 1. What placeholders are used when dealing with f-strings? A) (), B) [], C) {}
- 2. Use the `price` variable and f-string syntax to display the price with two decimals:

```
price = 59
txt = f"The price is {          } dollars"
print(txt)
```

- price:.2f
- 3. if price == 100 else
- 4. 1,000

Tuples

- heterogeneous sequences of data - similar to lists
- Immutable - similar to strings

Naming Convention

- **Classes** (other than Python's built-in classes):
  - a singular noun, capitalized: e.g., *Date* rather than *date* or *Dates*
  - multiple words concatenated to form a class name, follow the so-called "CamelCase" convention in which the first letter of each word is capitalized (e.g., CreditCard).

- **Functions**, including member functions of a class, should be lowercase (e.g., make_payment)

- **Names** that identify an individual **object** (a parameter, instance variable, or local variable) : a lowercase noun (e.g., price).

- Identifiers that represent a value considered to be a constant are traditionally identified using all capital letters and with underscores to separate words (e.g., MAX_SIZE).

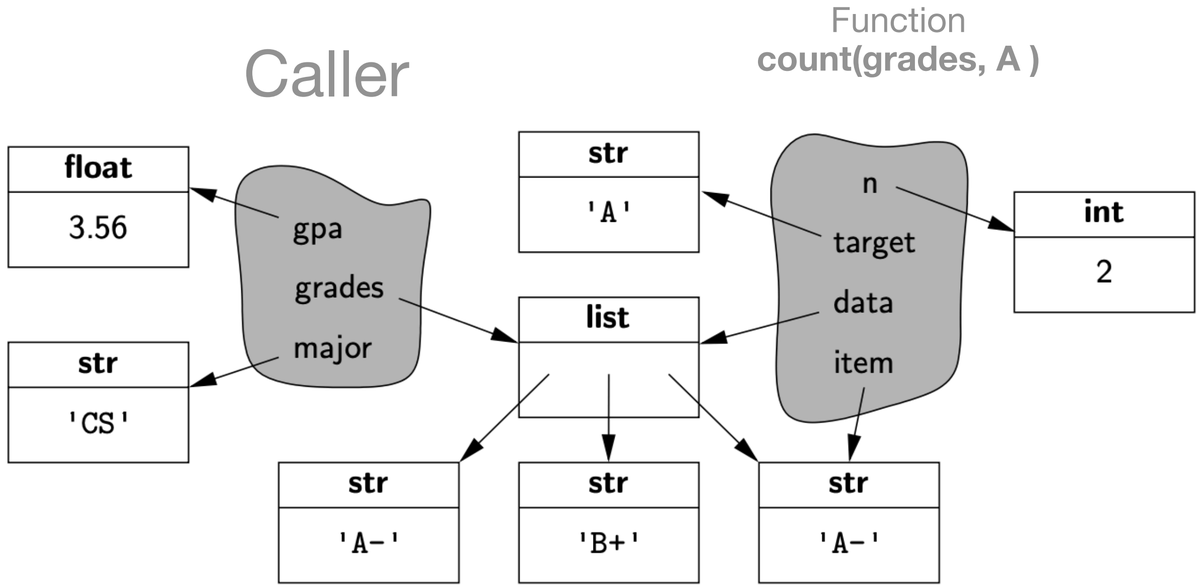
Scopes and Namespaces

- Name resolution:
  - The process of determining the value associated with an identifier x+y
  - names have to be associated with objects before use (NameError) e.g. z=x+y
  - an identifier assigned to a value has a scope I.e. a level
- **Scope: global or local, namespace - distinct scopes**
- **Namespace**: an abstraction representing each **distinct scope** in Python
  - manages all identifiers that are currently defined in a given scope
  - Python implements a namespace with its own **dictionary** that maps each identifying string (e.g., n) to its associated value
  - When an identifier is indicated in a command, Python searches a series of namespaces in the process of name resolution. First, the most locally enclosing scope is searched for a given name. If not found there, the next outer scope is searched, and so on
- ***first-class objects*** are instances of a type that can be assigned to an identifier, passed as a parameter, or returned by a function: all built in data types + functions

Name Spaces

- A portrayal of the two namespaces associated with a user's call **count(grades, A)**
- The left namespace is the caller's and the right namespace represents the local scope of the function.

.



Exercise:
Print the value of the object/variable within and outside of the function

Files

- Files are opened with a built-in function, **open**, that returns an object for the underlying file.
- For example, the command, fp = open('sample.txt'), attempts to open a file named sample.txt.
- Methods for files:

Calling Syntax	Description
fp.read( )	Return the (remaining) contents of a readable file as a string.
fp.read(k)	Return the next *k* bytes of a readable file as a string.
fp.readline( )	Return (remainder of) the current line of a readable file as a string.
fp.readlines( )	Return all (remaining) lines of a readable file as a list of strings.
for line **in** fp:	Iterate all (remaining) lines of a readable file.
fp.seek(k)	Change the current position to be at the k^{th} byte of the file.
fp.tell( )	Return the current position, measured as byte-offset from the start.
fp.write(string)	Write given string at current position of the writable file.
fp.writelines(seq)	Write each of the strings of the given sequence at the current position of the writable file. This command does *not* insert any newlines, beyond those that are embedded in the strings.
print(..., file=fp)	Redirect output of print function to the file.

Exception Handling

- Exceptions are unexpected events that occur during the execution of a program.
- An exception might result from a logical error or an unanticipated situation.
- In Python, exceptions (also known as errors) are objects that are raised (or thrown) by code that encounters an unexpected circumstance.
  - The Python interpreter can also raise an exception.
- A raised error may be caught by a surrounding context that "handles" the exception in an appropriate fashion. If uncaught, an exception causes the interpreter to stop executing the program and to report an appropriate message to the console.

Common Exceptions

- Python includes a rich hierarchy of exception classes that designate various categories of errors

Class	Description
Exception	A base class for most error types
AttributeError	Raised by syntax obj.foo, if obj has no member named foo
EOFError	Raised if "end of file" reached for console or file input
IOError	Raised upon failure of I/O operation (e.g., opening file)
IndexError	Raised if index to sequence is out of bounds
KeyError	Raised if nonexistent key requested for set or dictionary
KeyboardInterrupt	Raised if user types ctrl-C while program is executing
NameError	Raised if nonexistent identifier used
StopIteration	Raised by next(iterator) if no element; see Section 1.8
TypeError	Raised when wrong type of parameter is sent to a function
ValueError	Raised when parameter has invalid value (e.g., sqrt(−5))
ZeroDivisionError	Raised when any division operator used with 0 as divisor

Raising an Exception

- An exception is thrown by executing the raise statement, with an appropriate instance of an exception class as an argument that designates the problem.
- For example, if a function for computing a square root is sent a negative value as a parameter, it can raise an exception with the command:

```
raise ValueError('x cannot be negative')
```

Catching an Exception

- In Python, exceptions can be tested and caught using a try-except control structure.

```
try:
  ratio = x / y
except ZeroDivisionError:
  ... do something else ...
```

- In this structure, the "try" block is the primary code to be executed.
- Although it is a single command in this example, it can more generally be a larger block of indented code.
- Following the try-block are one or more "except" cases, each with an identified error type and an indented block of code that should be executed if the designated error is raised within the try-block.

Iterators

- Basic container types, such as list, tuple, and set, qualify as iterable types, which allows them to be used as an iterable object in a for loop.

```
for element in iterable:
```

- An iterator is an object that manages an iteration through a series of values. If variable, **i**, identifies an iterator object, then each call to the built-in function, **next(i)**, produces a subsequent element from the underlying series, with a **StopIteration** exception raised to indicate that there are no further elements.
- An iterable is an object, **obj**, that produces an iterator via the syntax **iter(obj)**.

Generators

- The most convenient technique for creating iterators in Python is through the use of generators.
- A generator is implemented with a syntax that is very similar to a function, but instead of returning values, a yield statement is executed to indicate each element of the series.
- For example, a generator for the factors of n:

```
def factors(n):                    # generator that computes factors
  for k in range(1,n+1):
    if n % k == 0:                 # divides evenly, thus k is a factor
      yield k                      # yield this factor as next result
```

Modules

- Beyond the built-in definitions, the standard Python distribution includes perhaps tens of thousands of other values, functions, and classes that are organized in additional libraries, known as **modules**, that can be imported from within a program.

```
import math
```

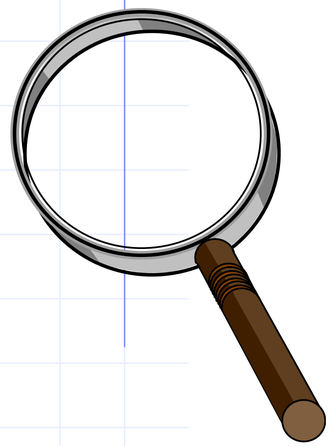
Existing Modules

- Some useful existing modules include the following:

Existing Modules	
Module Name	**Description**
array	Provides compact array storage for primitive types.
collections	Defines additional data structures and abstract base classes involving collections of objects.
copy	Defines general functions for making copies of objects.
heapq	Provides heap-based priority queue functions (see Section 9.3.7).
math	Defines common mathematical constants and functions.
os	Provides support for interactions with the operating system.
random	Provides random number generation.
re	Provides support for processing regular expressions.
sys	Provides additional level of interaction with the Python interpreter.
time	Provides support for measuring time, or delaying a program.

Algorithm Analysis and Big-O Notation

Theoretical Algorithm Analysis



- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/ software environment

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

```
1  def find_max(data):
2    """Return the maximum element from a nonempty Python list."""
3    biggest = data[0]                  # The initial value to beat
4    for val in data:                   # For each value:
5      if val > biggest                 # if it is greater than the best so far,
6        biggest = val                  # we have found a new best (so far)
7    return biggest                     # When loop ends, biggest is the max
```

  - Step 1: 2 ops, 3: 2 ops, 4: 2n ops, 5: 2n ops, 6: 0 to n ops, 7: 1 op

Estimating Running Time

- Algorithm **find_max** executes $5n + 5$ primitive operations in the worst case, $4n + 5$ in the best case. Define:

  a = Time taken by the fastest primitive operation

  b = Time taken by the slowest primitive operation

- Let $T(n)$ be worst-case time of **find_max**. Then

$$a(4n + 5) \leq T(n) \leq b(5n + 5)$$

- Hence, the running time $T(n)$ is bounded by two linear functions.

Big O Notation

- The **order of magnitude** function describes the part of *T(n)* that increases the fastest as the value of *n* increases
- **Big O notation** (for *order*) and written as *O(f(n))*
- provides a useful approximation of the actual number of steps in the computation
- The function f(n) provides a simple representation of the dominant part of the original T(n)

Why Growth Rate Matters

if runtime is...	time for n + 1	time for 2 n	time for 4 n
c lg n	c lg (n + 1)	c (lg n + 1)	c(lg n + 2)
c n	c (n + 1)	2c n	4c n
c n lg n	~ c n lg n + c n	2c n lg n + 2cn	4c n lg n + 4cn
c n^2	~ c n^2 + 2c n	**4c n^2**	16c n^2
c n^3	~ c n^3 + 3c n^2	8c n^3	64c n^3
c 2^n	c 2^{n+1}	c 2^{2n}	c 2^{4n}

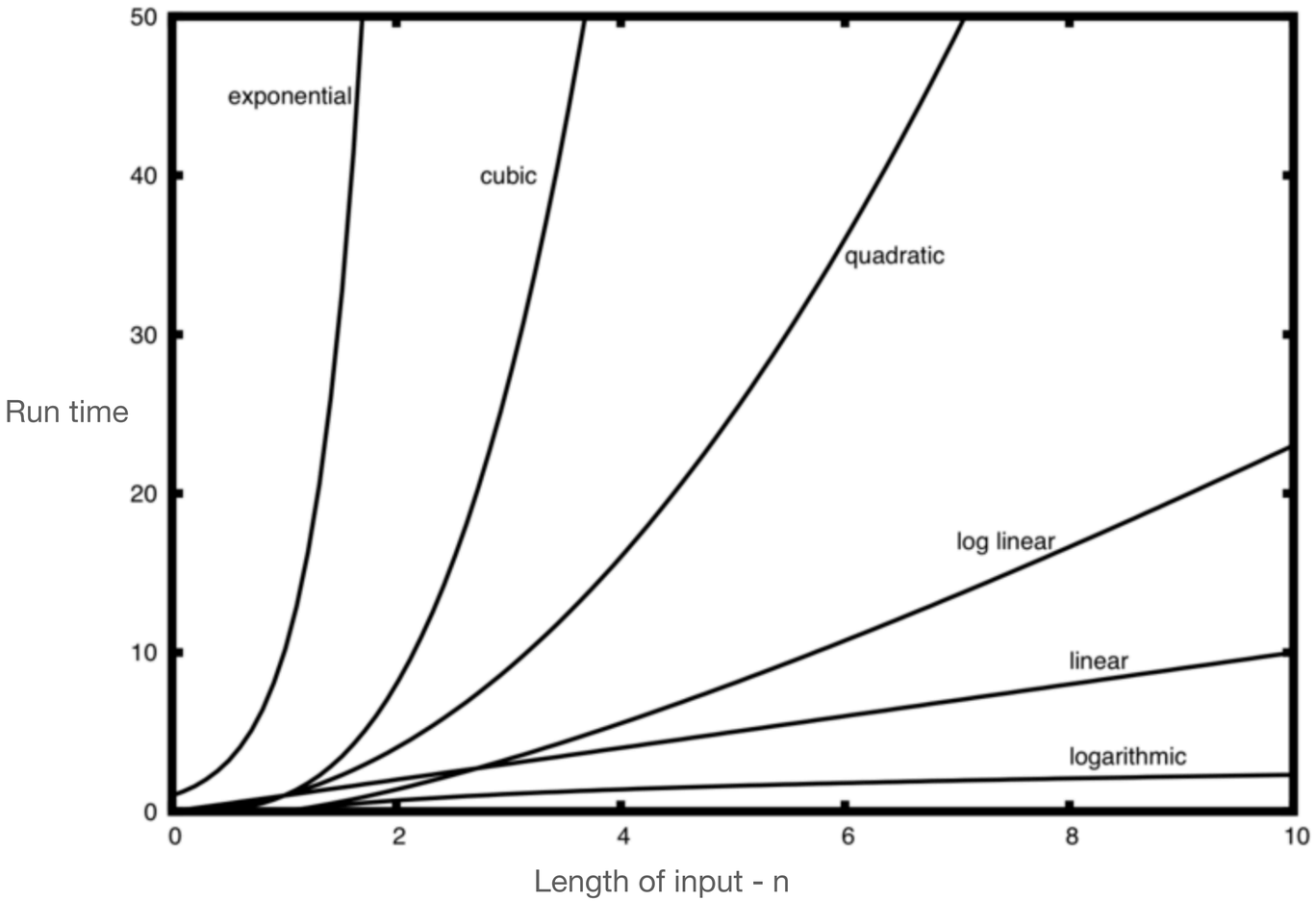
runtime quadruples when problem size doubles

Common Functions for Big O

f(n)	Name
1	**Constant**
$\log n$	**Logarithmic**
n	**Linear**
$n \log n$	**Log linear**
n^2	**Quadratic**
n^3	**Cubic**
2^n	**Exponential**

Common Functions for Big O

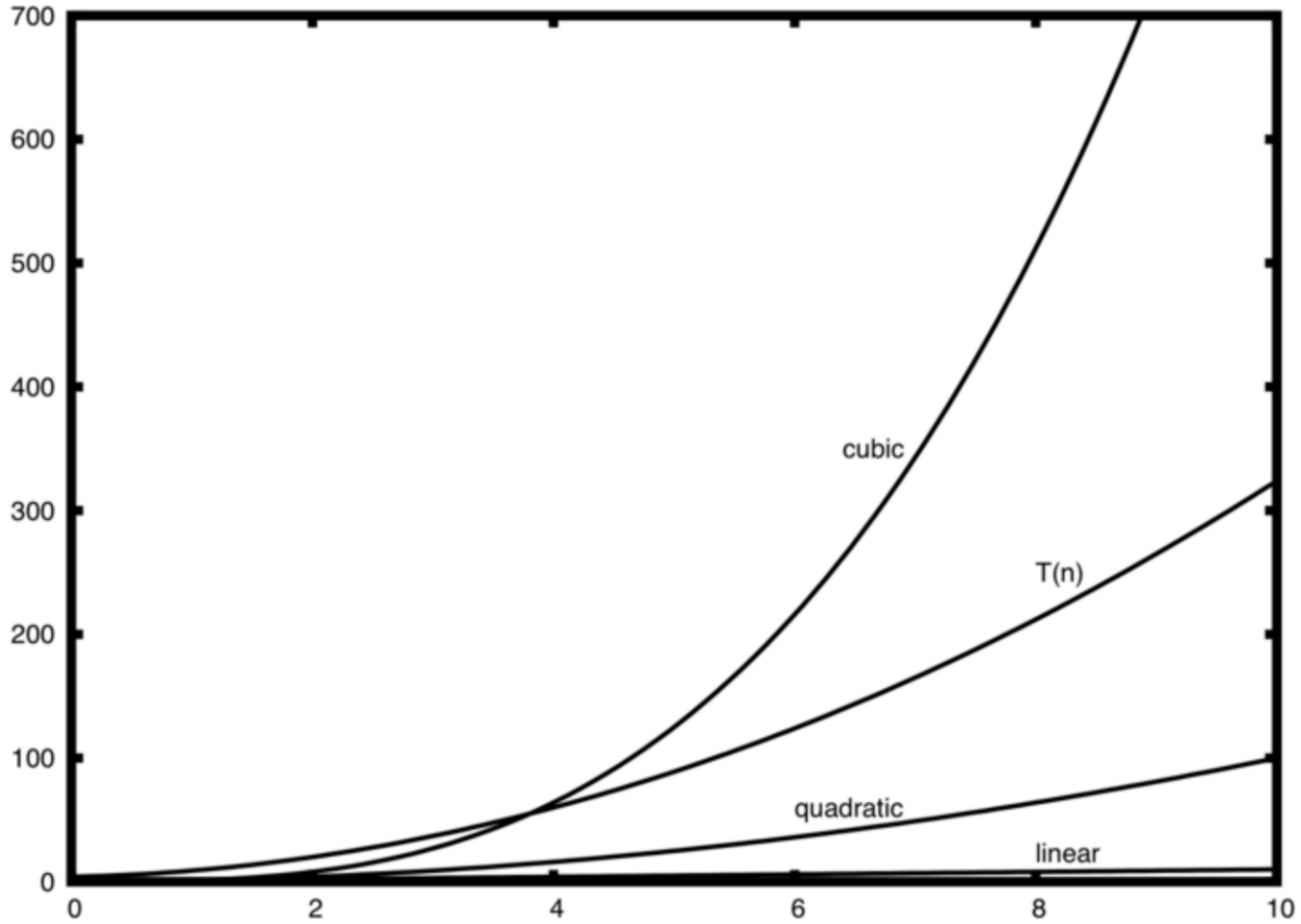
Graph



Calculate T(n) and O(n)

```
a = 5
b = 6
c = 10
for i in range(n):
    for j in range(n):
        x = i * i
        y = j * j
        z = i * j
for k in range(n):
    w = a * k + 45
    v = b * b
d = 33
```

Comparing T(n) and Big O




```
a = 5
b = 6
c = 10
for i in range(n):
    for j in range(n):
        x = i * i
        y = j * j
        z = i * j
for k in range(n):
    w = a * k + 45
    v = b * b
d = 33
```

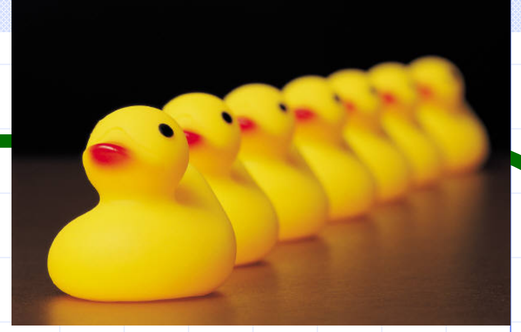
$$T(n) = 3 + 3n^2 + 2n + 1 = 3n^2 + 2n + 4.$$

$$O(n^2).$$

Object Oriented Programming, Python Classes

Note: Significant slides have green frame

Duck Typing



- Python treats abstractions implicitly using a mechanism known as **duck typing**.
  - A program can treat objects as having certain functionality and they will behave correctly provided those objects provide this expected functionality.
- As an interpreted and dynamically typed language, there is no "compile time" checking of data types in Python, and no formal requirement for declarations of abstract base classes.
- The term "duck typing" comes from an adage attributed to poet James Whitcomb Riley, stating that "**when I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck**."

Terminology

- Each **object** created in a program is an **instance** of a **class**.
- Each class presents to the outside world a concise and consistent view of the objects that are instances of this class, without going into too much unnecessary detail or giving others access to the inner workings of the objects.
- The class definition typically specifies **instance variables**, also known as **data member**s, that the object contains, as well as the **methods**, also known as **member functions**, that the object can execute.

Goals

- Robustness
  - We want software to be capable of handling unexpected inputs that are not explicitly defined for its application.
- Adaptability
  - Software needs to be able to evolve over time in response to changing conditions in its environment.
- Reusability
  - The same code should be usable as a component of different systems in various applications.

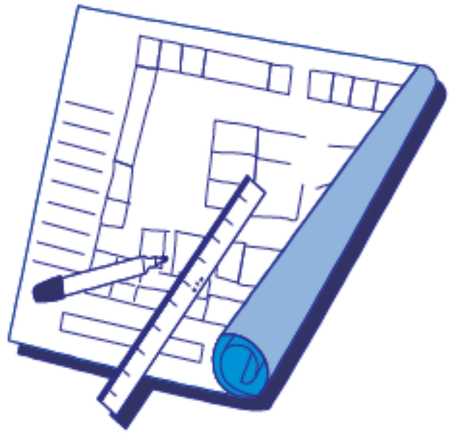
Abstract Data Types

- **Abstraction** is to distill a system to its most fundamental parts.
- Applying the abstraction paradigm to the design of data structures gives rise to **abstract data types** (ADTs).
- An ADT is a model of a data structure that specifies the **type** of data stored, the **operations** supported on them, and the types of parameters of the operations.
- An ADT specifies what each operation does, but not how it does it.
- The collective set of behaviors supported by an ADT is its **public interface**.

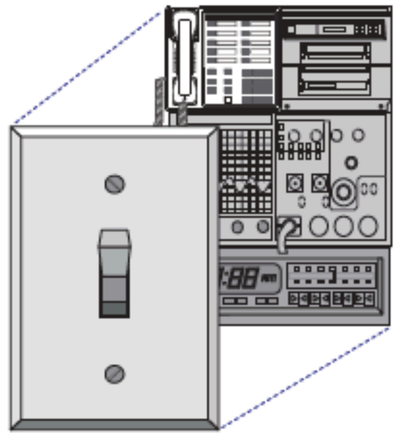
Object-Oriented Design Principles

- Modularity
- Abstraction
- Encapsulation

Modularity

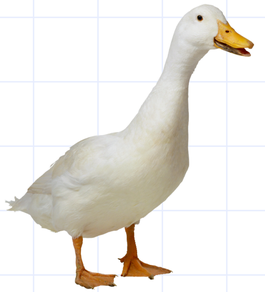


Abstraction



Encapsulation

Abstract Base Classes



- Python supports abstract data types using a mechanism known as an **abstract base class (ABC)**.
- An abstract base class cannot be instantiated, but it defines one or more common methods that all implementations of the abstraction must have.
- An ABC is realized by one or more concrete classes that inherit from the abstract base class while providing implementations for those method declared by the ABC.
- We can make use of several existing abstract base classes coming from Python's collections module, which includes definitions for several common data structure ADTs, and concrete implementations of some of these.

Encapsulation

- Another important principle of object-oriented design is **encapsulation**.
  - Different components of a software system should not reveal the internal details of their respective implementations.
- Some aspects of a data structure are assumed to be public and some others are intended to be internal details.
- Python provides only loose support for encapsulation.
  - By convention, names of members of a class (both data members and member functions) that start with a single underscore character (e.g., _secret) are assumed to be nonpublic and should not be relied upon.

Object-Oriented Software Design

- **Responsibilities**: Divide the work into different actors, each with a different responsibility.
- **Independence**: Define the work for each class to be as independent from other classes as possible.
- **Behaviors**: Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it.

Unified Modeling Language (UML)

A **class diagram** has three portions.

1. The name of the class
2. The recommended instance variables
3. The recommended methods of the class.

Class:	CreditCard	
Fields:	_customer _bank _account	_balance _limit
Behaviors:	get_customer( ) get_bank( ) get_account( ) make_payment(amount)	get_balance( ) get_limit( ) charge(price)

Class Definitions

- A class serves as the primary means for abstraction in object-oriented programming.
- In Python, every piece of data is represented as an instance of some class.
- A class provides a set of behaviors in the form of member functions (also known as **methods**), with implementations that belong to all its instances.
- A class also serves as a blueprint for its instances, effectively determining the way that state information for each instance is represented in the form of **attributes** (also known as **fields**, **instance variables**, or **data members**).

The **self** Identifier

- In Python, the **self** identifier plays a key role.
- In any class, there can possibly be many different instances, and each must maintain its own instance variables.
- Therefore, each instance stores its own instance variables to reflect its current state. Syntactically, **self** identifies the instance upon which a method is invoked.

Example

```
1  class CreditCard:
2    """A consumer credit card."""
3
4    def __init__(self, customer, bank, acnt, limit):
5      """Create a new credit card instance.
6
7      The initial balance is zero.
8
9      customer  the name of the customer (e.g., 'John Bowman')
10     bank      the name of the bank (e.g., 'California Savings')
11     acnt      the acount identifier (e.g., '5391 0375 9387 5309')
12     limit     credit limit (measured in dollars)
13     """
14     self._customer = customer
15     self._bank = bank
16     self._account = acnt
17     self._limit = limit
18     self._balance = 0
19
```

Example, Part 2

```
20    def get_customer(self):
21      """Return name of the customer."""
22      return self._customer
23
24    def get_bank(self):
25      """Return the bank's name."""
26      return self._bank
27
28    def get_account(self):
29      """Return the card identifying number (typically stored as a string)."""
30      return self._account
31
32    def get_limit(self):
33      """Return current credit limit."""
34      return self._limit
35
36    def get_balance(self):
37      """Return current balance."""
38      return self._balance
```

Example, Part 3

```
39    def charge(self, price):
40      """Charge given price to the card, assuming sufficient credit limit.
41
42      Return True if charge was processed; False if charge was denied.
43      """
44      if price + self._balance > self._limit:          # if charge would exceed limit,
45        return False                                   # cannot accept charge
46      else:
47        self._balance += price
48        return True
49
50    def make_payment(self, amount):
51      """Process customer payment that reduces balance."""
52      self._balance -= amount
```

Constructors

- A user can create an instance of the CreditCard class using a syntax as:

```
cc = CreditCard('John Doe, '1st Bank', '5391 0375 9387 5309', 1000)
```

- Internally, this results in a call to the specially named __init__ method that serves as the constructor of the class.
- Its primary responsibility is to establish the state of a newly created credit card object with appropriate instance variables.

Operator Overloading

- Python's built-in classes provide natural semantics for many operators.
- For example, the syntax a + b invokes addition for numeric types, yet concatenation for sequence types.
- When defining a new class, we must consider whether a syntax like a + b should be defined when a or b is an instance of that class.

Iterators

- Iteration is an important concept in the design of data structures.
- An **iterator** for a collection provides one key behavior:
  - It supports a special method named `__next__` that returns the next element of the collection, if any, or raises a StopIteration exception to indicate that there are no further elements.

Automatic Iterators

- Python also helps by providing an automatic iterator implementation for any class that defines both __len__ and __getitem__.

```
1  class Range:
2    """A class that mimic's the built-in range class."""
3
4    def __init__(self, start, stop=None, step=1):
5      """Initialize a Range instance.
6
7      Semantics is similar to built-in range class.
8      """
9      if step == 0:
10       raise ValueError('step cannot be 0')
11
12     if stop is None:                    # special case of range(n)
13       start, stop = 0, start            # should be treated as if range(0,n)
14
15     # calculate the effective length once
16     self._length = max(0, (stop - start + step - 1) // step)
17
18     # need knowledge of start and step (but not stop) to support __getitem__
19     self._start = start
20     self._step = step
21
22   def __len__(self):
23     """Return number of entries in the range."""
24     return self._length
25
26   def __getitem__(self, k):
27     """Return entry at index k (using standard interpretation if negative)."""
28     if k < 0:
29       k += len(self)                    # attempt to convert negative index
30
31     if not 0 <= k < self._length:
32       raise IndexError('index out of range')
33
34     return self._start + k * self._step
```

Inheritance

- A mechanism for a modular and hierarchical organization is **inheritance**.
- This allows a new class to be defined based upon an existing class as the starting point.
- The existing class is typically described as the **base class**, parent class, or superclass, while the newly defined class is known as the **subclass** or child class.
- There are two ways in which a subclass can differentiate itself from its superclass:
  - A subclass may specialize an existing behavior by providing a new implementation that overrides an existing method.
  - A subclass may also extend its superclass by providing brand new methods.

Inheritance is Built into Python

- A portion of Python's hierarchy of exception types: