Python Primer 2: Functions and Control Flow

Program Structure

- Common to all control structures, the colon character is used to delimit the beginning of a block of code that acts as a body for a control structure.
- If the body can be stated as a single executable statement, it can technically placed on the same line, to the right of the colon.
- However, a body is more typically typeset as an indented block starting on the line following the colon.
- Python relies on the indentation level to designate the extent of that block of code, or any nested blocks of code within.

Conditionals

```
if first_condition:
    first_body
elif second_condition:
    second_body
elif third_condition:
    third_body
else:
    fourth_body
```

Loops

- While loop:

```
while condition:
    body
```

- For loop:

```
for element in iterable:
    body                    # body may refer to 'element' as an identifier
```

- Indexed For loop:

```
big_index = 0
for j in range(len(data)):
    if data[j] > data[big_index]:
        big_index = j
```

Break and Continue

- Python supports a **break** statement that immediately terminate a while or for loop when executed within its body.

```
found = False
for item in data:
  if item == target:
    found = True
    break
```

- Python also supports a **continue** statement that causes the current iteration of a loop body to stop, but with subsequent passes of the loop proceeding as expected.

Functions

- Functions are defined using the keyword **def**.

```
def count(data, target):
  n = 0
  for item in data:
    if item == target:
      n += 1
  return n
```

- This establishes a new identifier as the name of the function (count, in this example), and it establishes the number of parameters that it expects, which defines the function's **signature**.
- The **return** statement returns the value for this function and terminates its processing.

Information Passing

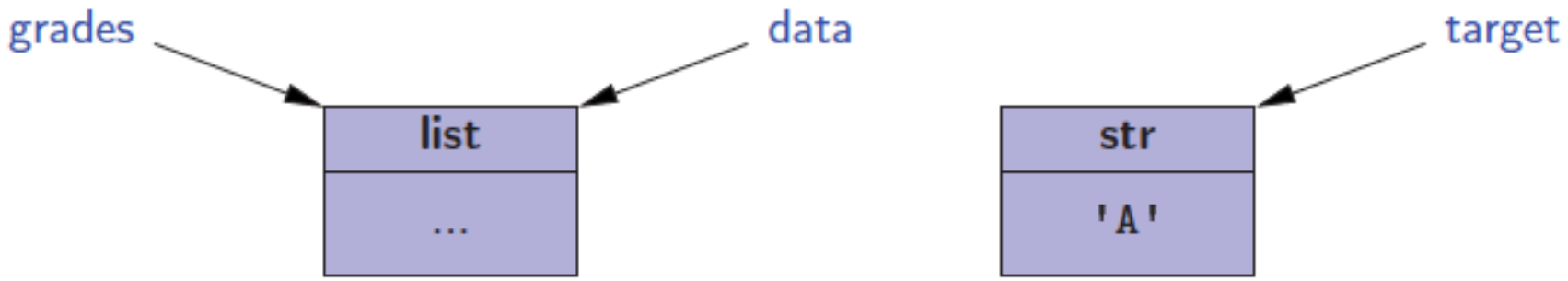
- Parameter passing in Python follows the semantics of the standard assignment statement.
- For example

```
prizes = count(grades, 'A')
```

is the same as

```
data = grades
target = 'A'
```

and results in



Simple Output

- The built-in function, **print**, is used to generate standard output to the console.
- In its simplest form, it prints an arbitrary sequence of arguments, separated by spaces, and followed by a trailing newline character.
- For example, the command print('maroon', 5) outputs the string 'maroon 5\n'.
- A nonstring argument x will be displayed as str(x).

Simple Input

- The primary means for acquiring information from the user console is a built-in function named **input**.
- This function displays a prompt, if given as an optional parameter, and then waits until the user enters some sequence of characters followed by the return key.
- The return value of the function is the string of characters that were entered strictly before the return key.
  - Such a string can immediately be converted, of course:

```
year = int(input('In what year were you born? '))
```

A Simple Program

- Here is a simple program that does some input and output:

```
age = int(input('Enter your age in years: '))
max_heart_rate = 206.9 - (0.67 * age)    # as per Med Sci Sports Exerc.
target = 0.65 * max_heart_rate
print('Your target fat-burning heart rate is', target)
```

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
```

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)
```

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.

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.