CS 1112 information sheet

Doing assignments

- Normally each part of an assignment corresponds to something recently covered in class. If this does not seem to be the case, look again.

Syntax, semantics, and style

- A syntax rule is a requirement imposed by a programming language; e.g., a left parenthesis must have a matching right parenthesis.
- A semantics rule defines how something works; e.g., the `+` operator performs addition when it has numeric operands.
- A style rule is a convention for developing code; e.g., variables names composed of more than one word have the words separated by underscores; e.g., `snake_case`.

Program execution

- Programs are stored in files with the suffix `.py`.
- Python is an interpreted language. A program is first compiled to determine whether it is syntactically correct.
- If a program has no syntax errors, the code is translated into bytecodes, a lower-level instruction language.
- An interpreter converts the bytecodes into machine instructions and runs (executes / carries out) the instructions.

Comments

- The `#` indicates the rest of the line is not a programming instruction. The text is instead a comment documenting some aspect of the code.
- Programs often have header comment section that identifies the purpose of the program, the authors, and contact information.
- A function normally begins with a comment indicating its purpose.
- A major section of code within a function is typically preceded by a comment indicating its task.

Whitespace

- Whitespace between programming elements is ignored during compilation.
- Whitespace is used to separate program elements for increased readability.
- Statement lists within control constructs must be indented. The indentation indicates that the statements are part of a decision or looping statement. Standard indentation is 3 or 4 spaces per level. The number of spaces must be consistent within a program element.
- For readability, long statements (typically 72 or more characters) are generally broken up into multiple lines. The additional lines are indented with respect to the starting line of the statement.
- Long statements are generally broken immediately after an operator.

Identifiers

- Identifier is the computing term for a name.
- An identifier must begin with a letter. Subsequent characters (if any) can be letters or numbers. The underscore (`_`) is considered to be a letter.
- Identifiers are case-dependent; e.g., `count` and `COUNT` are different identifiers.

Reserved words

- Reserved words cannot be used as names.
- A keyword is a word reserved for special purposes. Keywords are case sensitive. Python keywords include `and`, `break`, `def`, `del`, `elif`, `else`, `False`, `for`, `if`, `import`, `in`, `is`, `None`, `not`, `or`, `pass`, `print`, `True`, and `while`.

Variables

- A variable is a symbolic name for a memory location containing a reference (think pointer) where to find a value (e.g., a number or string).
- Variable names must be identifiers.
- A variable must be initialized when it is first introduced into a program.
- The standard variable naming style rule in Python is called snake case. In snake case a name is composed of all lowercase letters except for a single underscore between each word in the name; e.g., `peas_per_pod`.

Escape sequence

- An escape sequence allows a special character to be easily represented.

- Some escape sequences are
  - `\t` represents the tab character.
  - `\n` represents the newline character.
  - `\\` represents the backslash character.
  - `\'` represents the single quote character.
  - `\"` represents the double quote character.

Types and casting

- A type is a collection of values along with the operators, functions, and methods that manipulate the values.
- The built-in numeric types are `int` and `float`. Type `int` is for integer; type `float` is for decimals. Built-in string type `str` is for character sequences. Built-in logical type `bool` is for representing logical values.
- The built-in types have cast functions of the same name that produce values of that type. The cast functions are listed below.
- `int(x)`

  If `x` is a number, returns the integer gotten by dropping the fractional part of `x`. If x is literal string of digits, returns the number represented by the string.
- `float(x)`

  If `x` is a number, returns the decimal equivalent of the number. If x is literal decimal string, returns the number represented by the string.
- `str(x)`

  Returns a text representation of `x` suitable for printing.
- `bool(x)`

  Returns `False` if `x` equals `False`, `0`, `0.0`, `''`, `""`, or `None`; otherwise, returns `True`.

Values and objects

- An explicit numeric, character string, or logical value (e.g., `19`, `2.8`, `'aardvark'`, or `True)` is called a literal value.
- The value of a variable is a reference to an object.
- Reserved word `None` indicates a no reference.
- There is a difference between uninitialized and `None`. The former indicates no value as of yet; the latter indicates the no reference value.

- The dot operator (`.`) is the selection operator. A dot gives access to an element of an object; e.g., `s.f()` is the `f()` method of the object referenced by `s`.

Standard output

- Standard output is where program output goes by default. The default standard output is the window running the program.
- Built-in function `print()` displays the values of its parameters along with a newline character to standard output. There is a single blank separating the values from one another.

Standard input

- Standard input is where program input comes from by default. The default standard input is the program user.
- Built-in function `input()` gets the next line of data from standard input and returns it as a string. The function can take a string as an optional parameter. The string is displayed to standard output to prompt for user input.
- If non-string data is wanted, use built-in cast functions `int()`, `float()`, and `bool()` to convert from string data.

Built-in functions

- Besides the `print()`, `input()`, and cast functions, there are many other handy built-in functions. Several are listed below.
  - `abs(x)`

    Returns the absolute value of `x`
  - `help(x)`

    Prints the help page for `x`.
  - `len(x)`

    Returns the length of `x`.
  - `max(x1, x2, ...)`

    Returns the maximum value in sequence `x`.
  - `min(x1, x2, ...)`

    Returns the minimum value in sequence `x`.
  - `open(f, m)`

    Returns a file type object for processing file name `f`. The type of processing is determined by mode value `m`. The mode is optionable. By default the mode is reading ('r'). Some of the other modes are 'a' for appending text to the file and 'w' for writing the file (any existing data in the file is removed).

- round(x, d)

  Rounds x to d digits after the decimal point. Precision d is optional. If no precision is given, x rounds to its nearest integer; otherwise, it rounds to a decimal.
- sorted(x)

  Returns a new sorted version of x.
- sum(x1, x2, ...)

  Returns the sum of the values in sequence x.

Assignment

- The assignment operator = replaces the value of its left operand with the value of its right operand. In an assignment, the left operand is called the target.
- The compound operators +=, -=, *=, /=, //=, **=, and %= respectively increment, decrement, scale, divide, divide, and modulate the target by the value of the right operand; e.g., n += 5 increments the value of variable n by 5.

Operators and evaluation

- The act of determining the value of an expression is called evaluation.
- When in doubt in writing an expression, use parentheses to make your intention explicit. In fact, when even close to doubt, use parentheses.
- Python provides the standard numeric operators +, -, *, and / along with the integer division operator //, remainder (modulus) operator %, and the exponentiation operator **.
- Except for the division operator /, the numeric operators when given two integer operands produce an integer result. The division operator / always produces a decimal result.
- Except for the integer division operator //, the numeric operators when given at least one decimal operand produce a decimal result. The integer division operator // always produces an integer result.
- The + operator performs concatenation when both of its operands are strings.
- In expressions composed of more than one operator, precedence and associativity rules determine the order of operator evaluation.
- The grouping operator () has higher precedence than unary, numeric, relational, logical, and assignment operators.
- The unary operators +, -, and ! have higher precedence than numeric, relational, logical, and assignment operators.
- The numeric operators have precedence than relational, logical, and assignment operators.
- The multiplicative operators *, /, //, and % have higher precedence than additive operators +, and -.
- The relational operators <, <=, >, and >=, !=, and ==, have higher precedence than logical and assignment operators.
- The ordering operators <, <=, >, and >= have higher precedence than equality operators != and ==,.
- Logical operators and and or have higher precedence than assignment operators.
- Operator and has higher precedence than operator or.
- When operators have equal precedence, associativity rules determine the order of evaluation.
- Unary and assignment operators are evaluated right to left. Other operators are evaluated left to right.
- If the left operand of an and operator evaluates to false, the right operand is not evaluated – the operation must be false.
- If the left operand of an or operator evaluates to true, the right operand is not evaluated – the operation must be true.

Equality

- The operators == and != test respectively whether two values are the same or different.
- The operators is and is not test respectively whether two values reference or do not reference the same object memory location.

Functions

- A function is a named piece of code that can take parameters and produce a value.

- A request for a function to carry out its task is called an invocation, calling, or execution.
- There are three kinds of functions: built-in function, module functions, and message method functions.
- To invoke function `f()` from module `m`, the invocation has form `m.f()`.
- A message method sends a message to its object to carry out a behavior (action).
- Suppose object `s` has a message method `f()`. To have `f()` manipulate `s`, the method invocation has form `s.f()`.
- Without an object to be the target of its manipulation, message methods do not make sense.

Strings

- Python uses class `str` for representing character strings.
- Strings are immutable; that is, there is no way to modify a string after it has been created. A manipulation of a string produces a new string and leaves the original alone.

  A character string within single or double quotes is a string literal; e.g., `x'` and `"x"` are both literal representations of the string composed of the character x.
- `None` and `''` are different. The former indicates a no reference; the latter, a string of length 0.
- `'x'` and `x` are different. The former is a literal; the latter is an identifier.
- The characters in a string are accessible by their index. The first character has index 0, the second character has index 1, and so on.
- `len(s)`

  Returns the length of string `s`.
- `s.split(c)`

  If optional `c` is not present, returns the words in `s` as a list. If `c` is present, the words are split using `c` as the separator.
- `s.lower()`

  Returns a new string whose characters are the lowercase equivalents of `s`.
- `s.upper()`

  Returns a new string whose characters are the uppercase equivalents of `s`.
- `s. capitalize()`

  Returns a new string whose characters are the lowercase equivalents of `s` except for the first character, which is uppercase.
- `s.isalpha()`

  If `s` is empty, it returns `False`; otherwise, returns whether all characters in its string are alphabetic.
- `s.isdigit()`

  If `s` is empty, it returns `False`; otherwise, returns whether all characters in its string are base 10 digits (i.e., 0 … 9).
- `s.count(t, i, j)`

  Returns the number of occurrence of `t` in `s`, where the search is limited to its slice interval [`i : j`]. Parameters `i` and `j` are optional. If `j` is not present, `n` is used, where `n` is the length of `s`. If `i` and `j` are both not present, 0 and `n` are used respectively for `i` and `j`.
- `s.find(t, i, j)`

  Returns the first occurrence of `t` in `s`, where the search is limited to its slice interval [`i : j`]. If there are no occurrences, returns -1. Parameters `i` and `j` are optional. If `j` is not present, `n` is used, where `n` is the length of `s`. If `i` and `j` are both not present, 0 and `n` are used respectively for `i` and `j`.
- `s.rfind(t, i, j)`

  Returns the last occurrence of `t` in `s`, where the search is limited to its slice interval [`i : j`]. If there are no occurrences, returns -1. Parameters `i` and `j` are optional. If `j` is not present, `n` is used, where `n` is the length of `s`. If `i` and `j` are both not present, 0 and `n` are used respectively for `i` and `j`.
- `s.replace(t, u, n)`

  Parameter `n` is optional. If optional `n` is not present, it returns a new copy of its string with all occurrences of `t` replaced by `u`. If `n` is given, it returns a new copy with the first `n` occurrences of `t` replaced by `u`.

Math functionality

- The standard module `math` provides functions for computing power, exponential, logarithmic, and trigonometric functions. To make of the library, an import statement is needed at the beginning of the program.

  ```
  import math
  ```

CS 1112 information sheet

Random number generation

- Random number generators use seed values to start up their random value sequences. By default `random` uses a seed value based on the current time.
- The difference in seed values from program run to program run causes the generator to produced different sequences.
- To have access to random number generation the module `random` is imported.

  ```
  import random
  ```
- Module random provides a variety of ways to produce random values.
- `random.seed(s)`

  Initializes the random number generator. Optional parameter `s` is used to configure the random number generator to a particular start state. Parameter `s` can be a value of any type.This ability is useful during program development as it allows reproducibility during testing.
- `random.randrange(a, b)`

  Returns a random integer *v* such that `a` ≤ *v* < `b`.
- `random.randrange(a, b, s)`

  Returns a random integer *v* such that `a` ≤ *v* < `b`, and *v* equals `a + s *` *i* for some integer *i*.
- `random.random()`

  Returns a random float value *v* such that 0.0 ≤ v < 1.0
- `random.uniform(a, b)`

  Returns a random float value *v* such that `a` ≤ v < `b`.
- `random.gauss(m, s)`

  Returns a normally distributed float *v* such that the distribution of its possible values has mean 0 and standard deviation 1 (standard bell-shaped curve).
- `random.choice(s)`

  Returns a random value from the sequence `s`.
- `random.shuffle(s)`

  Returns a random shuffling of sequence `s`.

URL access

- Module `urllib` is a package of sub-modules for working with URLs. Our only interest is its sub-module `request`. To access the sub-module you must import the sub-module.

  ```
  Import random
  ```
- `urllib.request.urlopen(link)`

  Returns a connector providing access to the URL resource (think web page) named by string `link`. We call the connector a stream.
- `stream.read()`

  If `stream` is a connector returned by `urlopen()`, the contents of the URL resource are returned as encoded string.
- `page.decode('UTF-8')`

  If page is encoded string, the function returns a text-based version of page.
- Sample code segment for accessing the CS 1112 home page would be

  ```
  link = 'http://www.cs.virginia.edu/~cs1112'
  stream = urllib.request.urlopen(link)
  page = stream.read()
  text = page.decode('UTF-8')
  ```

  When the segment completes, string variable `text` is the web page contents.

Collections

- An important part of a programming language is the ability to store and manipulate collections of values.
- A collection of values can either be ordered or unordered. An ordered collection is a sequence of values. A sequence has a first element, second element, and so on. A string is a sequence of characters.
- Beside strings, Python has other types for representing sequences. They include ranges and lists.
- The number of elements in collection `c` can be gotten through `len(c)`.
- The element of maximum value in collection `c` can be gotten through `max(c)`.
- The element of minimum value in collection `c` can be gotten through `min(c)`.

- The number of elements in collection `c` equal to `x` can be gotten through `count(c, x)`.

Ranges

- A range is a consecutive sequence of integers.
- Built-in function `range()` can produce new ranges. Its usage has form `range(a, b)`, where `a` is optional (if `a` is not provided, 0 is used).
- The range produced by `range(a, b)` corresponds to the sequence of values `a`, `a+1`, ... `b-1`.
- The range produced by `range(b)` corresponds to the sequence of values `0`, `1`, ... `b-1`.
- Ranges are immutable. Once built, a range cannot be modified.

Lists

- Unlike ranges, lists are mutable — their elements can be modified, new elements can be added, and existing elements can be removed, they can also be sliced and subscripted.
- The empty list literal is `[]`.
- If `s` is a list then `len(s)` is the number of elements in `s`.
- If `s` is a list then `del s[a : b]` removes the elements from `s` with indices `a` through `b-1`.
- The `+` operator can combine two lists to produce a new list. If `s` and `t` are lists, then `u = s + t` makes `u` a new list. The number of elements in `u` is `len(s) + len(t).` List `u` corresponds to the list of values in `s` concatenated with the list of values in `t`.
- The `append()` method can be used to add a new element to the end of a list. If `s` is a list, the `s.append(x)` grows the size of `s` by one, where `x` is now the last value in `s`.
- The `count()` method can determine a number of occurrences. If `s` is a list, then `s.count( x)` is the number of occurrences of `x` in `s`.
- The `insert()` method can be used to insert a new element into a list, If `s` is a list, then `s.insert(a, x)` grows the size of `s` by one by inserting `x` into `s` at index `a`.
- Method `index(x, i, j)` determines the index of the first occurrence of `x` in its collection among the indices `i` ... `j-1`. Parameter `i` is optional. If not supplied, 0 is used. Parameter `j` is also optional. If not supplied, then `n` is used, where `n` is the number of elements in the collection.
- The `in` and `not in` operators determine whether values are or are not part of a collection. If `s` is a collection, then `x in s` is true when `x` is one of the elements of `s`, and `x not is` is true when `x` is not one of the elements of `s`.
- The `remove()` method can be used to remove a value from a list. If `s` is a list, then `s.remove(x)` removes the first occurrence of `x` from `s`.
- The `reverse()` method can be used to reverse values in a list. If `s` is a list, then `s.reverse()` reverses the order of element values in `s`.
- The `pop()` can be used to pop and element from a list. If `s` is a list, then `s.pop(i)` returns the element with index `i` from its list and also removes it from the list. Parameter `i` is optional, if it is not supplied `n-1` is used, where `n` is the number of elements in its list.
- Method `sort()`puts its list into sorted order. If `s` is a list, then `s.sort()` rearranges the elements of `s` into non-descending order.
- Method `copy()` produces a copy of its list. If `s` is a list, then `s.copy()` returns a new list, which has the same values as `s`.
- Method `clear()` can clear out the elements of a list. If `s` is a list, then `s.clear()` removes all elements from `s`.

Slicing and subscripting (also see slicing in string section)

- A sequence can be sliced to produce a new sequence formed out of a contiguous sub-section of the sequence.
- An element of an ordered collection can be accessed via its index.
- A string is a character sequence.
- The forms sequence slicing can take are listed below.
- `s[i : j]`

  Returns a new slice corresponding to the subsequence of sequence `s` from indices `i` to index `j-1`.
- `s[i :]`

  Returns a new slice corresponding to the subsequence of sequence `s` from index `i` on

- `s[: j]`

  Returns a new slice corresponding to the subsequence of sequence `s` from indices 0 to `j-1`.
- `s[:]`

  Returns a new copy of the sequence s.
- `s[i]`

  Returns the value at index `i` in sequence `s`. We say here that `i` is a subscript.

Mappings

- The built-in type `dict` supports sets of mappings from one value to another (i.e., a dictionary). In the description below, suppose `d` is a dictionary.
- The empty dictionary literal is `{}`.
- `len(d)` is the number of mappings in `d`.
- `d[k] = v` sets `d` to have a mapping from `k` to `v`. If there was a prior mapping from `k` in `d`, then that mapping is removed. In dictionary parlance, there is a now a mapping from key `k` to value `v` in `d`.
- `d[k]` is the value that key `k` maps to in `d`.
- `d.get(k)` returns the value that key `k` maps to in `d`. If there is no such value, returns `None`.
- `del d[k]` removes `k`'s mapping in `d`.
- `(k in d)` indicates whether `k` is one of key values in `d`.
- `(k not in d)` indicates whether `k` is not one of key values in `d`.
- `d.clear()` removes all mappings in `d`.
- `d.keys()` are the keys for `d`. There are no duplicates among the keys.
- `d.values()` are the values of the keys in `d`. There are duplicate values if more than one key maps to the same value.
- `d.pop(k)` removes `k`'s mapping in `d` and returns the value of that mapping.
- `d.popitem()` removes and returns an arbitrary mapping from `d`.

Control constructs

- For general problem solving a program the ability to control which statements are executed and how often.
- Python provides two iterative control constructs for statement repetition, the `for` and `while` statements.
- Python provides one conditional control construct, the `if` statement, for determining whether statements should be executed at all.
- The `for`, `while`, and `if` are all keywords.

For loop

- A `for` statement is a looping statement that iterates over a collection of values. The collection can be a range, sequence, or set.
- A `for` statement has syntax

  `for` *x* `in` *collection* `:`

      Action

  where
  - The action is a statement list of at least one statement. The action is repeated once for each value in the collection.
  - Each time through the loop, variable *x* takes on another value of the collection. For a range or sequence, the *x* values come in order.
  - The action statements are indented one-level further than the start of the `for` statement.
  - The value of *x* when the loop completes is the value assigned to `x` for the last iteration.
- For example, the following code segments, prints values `a` through `b-1`.

```
for x in range( a , b ) :
    print( x )
```

If statement

- An `if` statement is not a loop. Its action is executed once.
- The `if` statement uses a logical expression to determine the next action executed by a program.
- The most common form of the `if` statement has syntax

  `if` *logical-expression* `:`

      Action$_1$

```
else :
   $Action_2$
```

where

- The test expression evaluates to `True` or `false`.
- The actions are non-empty statement lists.
- The actions are indented one-level further than the start of the `if` statement. The `else` is indented at the same level as the `if`.

- The statement semantics are that the test expression is evaluated. If the expression is `True`, the first action executes; otherwise, the second action executes.
- Sometimes the action to be taken by a program depends on which of several logical expressions is `True`. The `if` statement has `elif` components for such processing. The syntax here has form

```
if $test\text{-}expression_1$ :
   $Action_1$
elif $test\text{-}expression_2$ :
   $Action_2$
...
elif $test\ \text{-}expression_n$ :
   $Action_n$
else :
   $Action_{n+1}$
```

where

- The test expression evaluate to `True` or `false`.
- The actions are non-empty statement lists.
- The actions are indented one-level further than the start of the `if` statement. The `elif`’s and `else` are indented at the same level as the `if`.

- The statement semantics are that *test-expression*$_1$ is evaluated first. If `True`, *Action*$_1$ executes; otherwise, *test-expression*$_2$ is evaluated. If `True`, *Action*$_2$ executes; otherwise, *test-expression*$_3$ is evaluated, and so on. If none of the test expressions are true, *Action*$_{n+1}$ executes.
- The `else` part of an `if` statement is optional.

Image

- `Image` is defined in the `PIL` module.
- In the descriptions below, suppose
  - `drawing` is an `Image`.
  - `(x, y)` is a two-tuple representation of a coordinate.
  - `(w, h)` is a two-tuple representation of the width and height of an `Image`.
  - `(r, g, b)` is a three-tuple representation of the red, green, and blue levels of a pixel (i.e., a color).
  - `fn` is a string whose value is the name of a file.
- `Image.new( 'RGB', (w, h) )` returns a new `Image` whose dimensions are `w` by `h`, and whose pixels are all black.
- `Image.new( 'RGB', (w, h), (r, g, b) )` returns a new `Image` whose dimensions are `w` by `h`, and whose pixels are all color `(r, g, b)`.
- `drawing.size` returns a two-tuple. The value of the tuple is the width and height of `drawing`.
- `drawing.getpixel( (x, y) )` returns a three-tuple. The three-tuple is the RGB representation of the pixel at location `(x, y)` in `drawing`.
- `drawing.putpixel( (x, y), (r, g, b) )` sets the pixel at location `(x, y)` in `drawing` to color `(r, g, b)`.
- `drawing.save( fn, 'PNG' )` saves a `'PNG'` representation of `drawing` in the file named by string `fn`.