CSSE1001 — Week 4 Notes

Table of contents

Non-Primitive Data

See csse1001[1] for course logistics — this note covers Lecture 4B's technical content.

Today's outline

- Lists: mutable ordered collections
- Dictionaries: hash tables

Learning objectives

1. Lists are mutable ordered collections; tuples (python-primitive-data-types[2]) are immutable ordered collections.
2. A dictionary maps keys to values via a hash function, and is unordered and mutable.

Lists

See python-lists[3] for the full reference on lists: creation, mutability, comparison, membership, `append`/`extend`, aliasing, copying (shallow/deep), passing to functions, slicing, nested lists/matrices, type hints, and unpacking into function arguments.

Exercise: counting occurrences

```
def count(xs: list[int], ys: list[int]) -> int:
    """ Return the number of times members of ys are in xs.
    >>> count([1, 2], [1, 2, 3, 4])
    2
    >>> count([1, 2], [1, 2, 2, 2, 3, 1, 4])
    5
    """
```

A while-loop solution:

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-primitive-data-types.pdf
[3] courses/csse1001/python-lists.pdf

```
def count(xs: list[int], ys: list[int]) -> int:
    ans, k = 0, 0
    while k < len(ys):
        y = ys[k]
        k += 1
        if y in xs:
            ans += 1
    return ans
```

A for-loop is more natural for this problem:

```
def count(xs: list[int], ys: list[int]) -> int:
    ans = 0
    for y in ys:
        if y in xs:
            ans += 1
    return ans
```

There is also a one-line solution:

```
def count(xs: list[int], ys: list[int]) -> int:
    return sum(y in xs for y in ys)
```

Exercise: rotating a list

We say the list `[0, 1, 2, 3, 4, 5]` *rotated by 2* is `[2, 3, 4, 5, 0, 1]`. Write a function `def rotate(xs: list, k: int) -> list` that rotates a list by `k`.

> No worked solution is given in the source for this exercise — left as an open exercise rather than invented here.

Question: Caesar cipher

Write code for encrypting and decrypting messages using the Caesar cipher, with function headers:

```
def encrypt_caeser(plaintext: str, shift: int) -> str:

def decrypt_caeser(ciphertext: str, shift: int) -> str:
```

Left unsolved in the source — the lecture moves directly on to dictionaries afterwards. A working `caesar_cipher` implementation (a single shift parameter instead of separate encrypt/decrypt functions) appears later, in Lecture 5C — see week-five-exercises[4].

Dictionaries

Here is a deliberately vague question: write a function that returns the *character frequency* of a string, ignoring case — e.g. the character frequency of `"hello world"` would encode that there is one `h`, three `l`s, and so on.

What are some viable representations?

Two lists (characters paired positionally with their counts):

```
[['h', 'e', 'l', 'o', ' ', 'w', 'r', 'd'],
 [ 1,   1,   3,   2,   1,   1,   1,   1 ]]
```

One-to-one encoding (a count for every letter of the alphabet, in order):

```
# a b c d e f g h i j k l m n o p q r s t u v w x y z
 [0,0,0,1,1,0,0,1,0,0,0,3,0,0,2,0,0,1,0,0,0,0,1,0,0,0]
```

Hash functions

In both representations we provided a way to map a **key** (a character) to a **value** (a count) — this mapping is called a **hash function**:

$$\texttt{str} \to \texttt{int}, \quad \texttt{h} \mapsto 1,\ \texttt{e} \mapsto 1,\ \texttt{l} \mapsto 3,\ \ldots$$

The hash function for the two-list encoding looks up the key's *position* in the first list, then indexes the second list at that position:

```
def hash_1(key: str) -> int:
    """Two list encoding."""
    xs = ['h', 'e', 'l', 'o', ' ', 'w', 'r', 'd']
    ys = [1, 1, 3, 2, 1, 1, 1, 1]
    k = xs.index(key)   # position of key in xs
    return ys[k]
```

[4] courses/csse1001/week-five-exercises.pdf

The hash function for the one-to-one encoding computes the key's position in the alphabet directly, via `ord`:

```
def hash_2(key: str) -> int:
    """One-to-one encoding."""
    xs = [0,0,0,1,1,0,0,1,0,0,0,3,0,0,2,0,0,1,0,0,0,0,1,0,0,0]
    pos_in_alpha = ord(key) - ord('a')   # position of key in alphabet
    return xs[pos_in_alpha]
```

Generally, we can index a collection by *any* type of key given some hash function that maps keys to values. Python has a "magic" hash function that indexes anything appropriately — implemented as the `dict` type. See python-dictionaries[5] for the full reference on dictionaries: creation, indexing, `keys`/`values`/`items`, `clear`/`copy` (and its shallow-copy gotcha), `get`, and the requirement that keys be immutable.

The "hello world" character frequency, encoded as a dictionary:

```
>>> char_to_freq = {
...     'd': 1, 'o': 1, ' ': 1, 'r': 1,
...     'w': 1, 'e': 1, 'l': 3, 'h': 1
... }
```

Summary

Lists are mutable ordered collections; dictionaries store key-value pairings, found quickly via a "magically" fast hash function.

Next lecture

For-loops and list comprehensions.

While-Loops

See csse1001[6] for course logistics — this note covers Lecture 4A's technical content.

[5] courses/csse1001/python-dictionaries.pdf
[6] courses/csse1001/csse1001.pdf

Today's outline

- Why we need to repeat code, possibly indefinitely
- The while-loop
- Simulating a do-while, and reading user input

Learning objectives

1. Loops are a control structure for repeating code.
2. A while-loop repeats code while a condition holds.
3. `input()` reads from the keyboard; `random.randint` generates random integers.

Motivation

There are (at least) two scenarios where repeating code, possibly indefinitely, is necessary:

1. prompting the user for valid input, and
2. playing a random game (e.g. guessing dice rolls).

See python-while-loops[7] for the full reference on loops, while-loops, `break`, augmented assignment operators (`+=`, `*=`, `/=`, `%=`), simulating a do-while, `input()`, and `random.randint`.

Accumulator pattern

```
>>> x = 0
>>> while x < 10:
...     x = x + 1
>>> x
10
```

Using an augmented assignment operator (see python-while-loops[8]) makes the accumulation more concise:

```
>>> x = 0
>>> while x < 10:
...     x += 1
>>> x
10
```

[7] courses/csse1001/python-while-loops.pdf
[8] courses/csse1001/python-while-loops.pdf

There is usually a key-stroke (typically `ctrl+c`) that terminates a runaway loop (e.g. `while True: print(x); x += 1`) — it is a good idea to learn what it is in your IDE.

Exercise: guess the number

Write a function `foo(target: int) -> int` that prompts the user to input a number until the user guesses some (secret) target. For each guess the function should print `Too high`, `Too low`, or `That's it!` depending on the guess, and return the number of guesses required.

```
def foo(target: int) -> int:
    number_of_guesses = 0
    while True:
        guess = int(input("Guess: "))
        number_of_guesses += 1
        if guess > target:
            print("Too high")
        elif guess < target:
            print("Too low")
        else:
            print("That's it!")
            return number_of_guesses
```

Exercise: guess the dice roll

Write a function `bar() -> None` that prompts the user to predict the result of a six-sided dice roll, repeating the prompt until the user predicts correctly:

```
from random import randint

def bar() -> None:
    """
    Prompt user to predict the result of a six sided
    dice roll until the guess is correct.
    """
    while True:
        guess = int(input("Guess in [1..6]: "))
        roll = randint(1, 6)
        if roll == guess:
            break
    return
```

Further exercises

The source poses these without a worked solution — left here as open exercises rather than guessed at:

- Write a function that prints the result of a six-sided dice roll until a six is rolled, and returns the number of rolls required.
- Write a function that accumulates the result of dice rolls until some threshold is met/passed, and returns the number of rolls required.
- Develop functions for adding and multiplying positive integers together, restricting yourself to exactly two arithmetic operations — `succ` (adding one) and `pred` (subtracting one). The source only gives the two building blocks:

```
def succ(x: int) -> int:
    return x+1

def pred(x: int) -> int:
    return x-1
```

(The add/multiply functions built from `succ`/`pred` are not solved in the source.)

Summary

The while-loop is a control structure for repeating code a possibly infinite number of times.

Next lecture

2025-08-18-non-primitive-data[9] — lists and dictionaries.

Reference material

Python Dictionaries

A **dictionary** is a data structure that stores `(key, value)` pairs. Python uses a "magic" hash function to find a key among the stored pairs quickly (see 2025-08-18-non-primitive-data[10] for how a hash function generalizes indexing-by-position to indexing-by-any-key). Dictionaries are **not ordered** (keys can be of mixed type) and are **mutable**.

Hash tables are a candidate for the most important/useful data structure in computer science.

[9] courses/csse1001/2025-08-18-non-primitive-data.pdf
[10] courses/csse1001/2025-08-18-non-primitive-data.pdf

Creating and indexing

```
>>> h = {
...     "red": ["apple", "firetrucks", "cars"],
...     "yellow": ["banana", "cars"],
...     "blue": ["sky", "cars"]
... }
>>> h["blue"]
['sky', 'cars']
>>> h["green"]
KeyError: 'green'
>>> h["green"] = ["leaves"]   # fine -- we're assigning, not retrieving
```

keys, values, items

```
>>> h.keys()
dict_keys(['red', 'yellow', 'blue'])
>>> h.values()
dict_values([['apple', 'firetrucks', 'cars'], ['banana', 'cars'], ['sky', 'cars']])
>>> h.items()
dict_items([('red', ['apple', 'firetrucks', 'cars']), ('yellow', ['banana', 'cars']), ('blue
```

clear and copy

```
>>> f = h.copy()
>>> h.clear()
>>> h["blue"]
KeyError: 'blue'
>>> f["blue"]
['sky', 'cars']
```

Like list's `.copy()` (see python-lists[11]), dict's `.copy()` is only a **shallow copy** — mutable values are still shared:

```
>>> f = h.copy()
>>> h["blue"].append("windex")
>>> h["blue"]
['sky', 'cars', 'windex']
>>> f["blue"]
['sky', 'cars', 'windex']   # shallow copy -- same underlying list
```

[11] courses/csse1001/python-lists.pdf

get

`.get(key)` is a safer alternative to `h[key]` — it returns `None` instead of raising `KeyError` if the key is missing:

```
>>> h.get("red")
['apple', 'firetrucks', 'cars']
>>> h["green"]
KeyError: 'green'
>>> h.get("green")
None
```

Without a method, the same "avoid a `KeyError`" pattern can be written explicitly:

```
>>> if key in table:
...     table[key] += 1
... else:
...     table[key] = 0
```

Warning: keys must be immutable

The keys of a dictionary must be an immutable type (see python-primitive-data-types[12] and python-lists[13]):

```
>>> d = dict()
>>> d[[1, 2, 3]] = 1
TypeError: unhashable type: 'list'
>>> d[(1, 2, 3)] = 1   # tuples are immutable -- fine
>>> d[dict()] = 1
TypeError: unhashable type: 'dict'
```

Python Lists

A **list** is a mutable ordered collection of elements — elements are not necessarily all the same type. Square brackets `[]` create a list in Python.

[12] courses/csse1001/python-primitive-data-types.pdf
[13] courses/csse1001/python-lists.pdf

```
>>> xs = [1, "apple"]
>>> type(xs)
<class 'list'>
>>> xs[0]
1
>>> xs[0] = 2*xs[1]
>>> xs[0]
'appleapple'
```

Comparison

Lists compare point-wise from position zero:

```
>>> [1, 2, 3] < [4, 5, 6]
True
>>> [7, 2, 3] < [4, 5, 6]
False
>>> [] < [1]
True
```

Membership

```
>>> 1 in [1, 2, 3]
True
>>> 0 in [1, 2, 3]
False
>>> [1] in [1, 2, 3]
False
```

Adding elements: `append` vs. concatenation

`xs.append(y)` mutates `xs` **in place**, adding `y` as a single new element (even if `y` is itself a list):

```
>>> xs = [0, 1, 2]
>>> xs.append(3)
>>> xs
[0, 1, 2, 3]
>>> xs.append([4, 5])
>>> xs
[0, 1, 2, 3, [4, 5]]
```

`xs + [y]` instead **creates a new list**, which must be reassigned back to `xs` to have an effect:

```
>>> xs = [0, 1, 2]
>>> xs = xs + [3]
>>> xs
[0, 1, 2, 3]
```

`xs.extend(ys)` mutates `xs` in place, appending every element of `ys`:

```
>>> xs = [1, 2, 3]
>>> xs.extend([4, 5])
>>> xs
[1, 2, 3, 4, 5]
```

Aliasing

Assigning one list variable to another does **not** copy it — both names refer to the *same* list object:

```
>>> xs = [1, 2, 3]
>>> ys = xs
>>> ys[-1] = 9
>>> ys
[1, 2, 9]
>>> xs
[1, 2, 9]
```

Whether an operation preserves this aliasing relationship depends on whether it mutates the list in place or creates a new one:

```
>>> xs = [1, 2, 3]
>>> ys = xs           # ys is an alias of xs
>>> xs.append(4)      # mutates in place
>>> ys
[1, 2, 3, 4]
>>> xs += [5]         # also mutates in place
>>> ys
[1, 2, 3, 4, 5]
>>> xs = xs + [6]     # creates a NEW list, only rebinds xs
>>> ys
[1, 2, 3, 4, 5]        # aliasing relationship broken!
```

`.copy()` breaks the alias for a flat list:

```
>>> xs = [1, 2, 3]
>>> ys = xs.copy()
>>> ys[-1] = 9
>>> xs
[1, 2, 3]
>>> ys
[1, 2, 9]
```

Passing lists to functions

Lists are passed to functions by reference, so mutating a parameter inside a function mutates the caller's list too:

```
>>> def foo(ys):
...     ys[0] = -100
>>> xs = [1, 2, 3]
>>> foo(xs)
>>> xs
[-100, 2, 3]
```

Nested lists and matrices

A list can contain another list as an element, e.g. representing a matrix as a list-of-lists:

$$\mathbb{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \equiv \texttt{[[1, 2, 3], [4, 5, 6], [7, 8, 9]]}$$

```
>>> ass = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ass[-1]
[7, 8, 9]
>>> ass[1:3]
[[4, 5, 6], [7, 8, 9]]
>>> ass[1][-1]
6
>>> ass[1, -1]
TypeError: list indices must be integers or slices, not tuple
```

Deep copying

`.copy()` only performs a **shallow copy** — nested mutable elements (like inner lists) are still shared between the original and the copy:

```
>>> ass = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> bss = ass.copy()
>>> bss[0][0] = 0
>>> bss
[[0, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ass
[[0, 2, 3], [4, 5, 6], [7, 8, 9]]   # ass was changed too!
```

A true independent copy of nested structures needs a **deep copy**, which isn't built into `list` — see `copy.deepcopy`.

Slicing

```
>>> xs = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> xs[3:6]
[3, 4, 5]
>>> xs[::2]
[0, 2, 4, 6, 8]
>>> xs[7:2:-2]
[7, 5, 3]
```

Type hints

Type-hinting the elements of a list (e.g. `list[int]`) is a feature new in Python 3.9 and greater.

Unpacking into function arguments

A list can be "unbracketed" into positional arguments with `*`:

```
>>> def f(x, y, z):
...     return x + y + z
>>> xs = [1, 2, 3]
>>> f(xs)
TypeError: f() missing 2 required positional arguments: 'y' and 'z'
```

```
>>> f(*xs)
6
```

Python Primitive Data Types

The types "built-in" to Python by default are called **primitive data**: booleans, integers, floats (not covered here), strings, and tuples.

Booleans

The boolean type comprehensively provides the values `True` and `False` (`type(True)` and `type(False)` are both `bool`).

Comparison operators

All comparison operators have equal precedence and are left-associative; when mixed with arithmetic, comparison is evaluated *last*.

Operator	Description
`==`	Equal
`!=`	Not equal
`<`, `<=`	Less than, less than or equal
`>`, `>=`	Greater than, greater than or equal

```
>>> 3 + 2 > 1 + 3   # arithmetic first
True
>>> 3+2 > 1+3       # better to group like this
True
>>> 3 + (2>1) + 3   # True has int value 1
7
```

Logical connectors

Functions that return booleans are called **predicates**. More sophisticated predicates can be built with the logical connectors `and`, `or`, and `not` (covered in detail in the if-statement lecture).

`bool` — truthy / falsy

Any object can be converted to a boolean with `bool`. Data that converts to `True` is **truthy**; data that converts to `False` is **falsy** — usually the falsy element is whatever acts as zero for the type, and everything else is truthy.

```
>>> bool(0)
False
>>> bool(1)
True
```

Integers

An **integer** is a number without a fractional part — zero, positive, or negative. Integers are unbounded in Python, so we can work with arbitrarily large integers without overflow (which is unusual amongst languages):

```
>>> 2 ** 256 - 1
115792089237316195423570985008687907853269984665640564039457584007913129639935
```

Booleans convert to integers with `int` (`True` behaves as `1`, `False` as `0`):

```
>>> int(True)
1
>>> int(False)
0
>>> True + False
1
>>> True * False
0
```

Strings

A **string** is (with some exceptions) anything enclosed by single- or double-quotes — an ordered collection of the characters (e.g. unicode/ascii) the computer allows.

```
>>> "hello world"
'hello world'
>>> type("hello world")
<class 'str'>
```

```
>>> hello world   # note the lack of quotes
SyntaxError: invalid syntax
>>> hello         # note the lack of quotes
NameError: name 'hello' is not defined
```

`str` conversion and formatting

```
>>> str(1)
'1'
>>> str(True)
'True'
```

Formatted strings (`f"..."`) substitute variables into a string:

```
>>> x = 1
>>> y = "two"
>>> f"x is {x} y is {y}"
'x is 1 y is two'
>>> print(f"x is {x} y is {y}")
x is 1 y is two
>>> f"{x}"   # alternative to str
'1'
```

Concatenation and scalar multiplication

Adding strings creates a new string; multiplying a string by a positive integer repeats it:

```
>>> "hello" + "world"
'helloworld'
>>> space = " "
>>> "hello" + space + "world"
'hello world'
>>> 3 * "Hello World!"
'Hello World!Hello World!Hello World!'
```

Comparing strings

Strings compare lexicographically by character order (`ord`/`chr` give a character's order and the character at an order):

```
>>> "a" < "b"
True
>>> ord("a"), ord("b")
(97, 98)
>>> chr(97)
'a'
>>> "A" < "a"
True
>>> "Z" < "a"
True
```

A shorter string is less than an extension of itself, but otherwise comparison proceeds character-by-character:

```
>>> "a" < "aa"
True
>>> "b" < "aa"
False
>>> "aba" < "ab"
False
>>> "aZ" < "aa"
True
```

> The lecture previews an exercise (`string_less_than(cs, ds)`, restricted to comparing integer character codes) that it explicitly defers ("we will return to this") without giving a worked solution — flagged here rather than invented.

Escape characters

`\n` (new line) and `\t` (tab) are escape characters — a string can be *stored* differently than it is *printed*:

```
>>> print("hello\nworld")
hello
world
>>> print("hello\tworld")
hello   world
```

Tabs display as a fixed amount of horizontal space, but exactly how much depends on the program displaying them.

Numbers versus strings

`+` does not silently convert between `int` and `str` — mixing them raises a `TypeError`:

```
>>> "3" + "7"
'37'
>>> 3 + "7"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> str(3) + "7"
'37'
>>> 3 + int("7")
10
```

`int()`/`float()` only work on strings that are actually numbers expressed as digits:

```
>>> int("3.14")
ValueError: invalid literal for int() with base 10: '3.14'
>>> float("123.456")
123.456
>>> int("seven")
ValueError: invalid literal for int() with base 10: 'seven'
```

Length and inclusion

```
>>> len("hello")
5
>>> cs = "world"
>>> len(cs+"world") == len(cs) + len("world")
True
>>> "h" in "hello world"
True
>>> "ow" in "hello world"
False
```

Indexing and slicing

Strings are ordered, so characters are numbered from zero and accessed with square brackets. Negative indices count from the end:

```
>>> cs = "hello world"
>>> cs[0]
'h'
>>> cs[-1]
'd'
>>> cs[len(cs)]
IndexError: string index out of range
```

A slice `cs[start:stop:step]` grabs the *start*-inclusive, *stop*-exclusive characters, stepping by *step* (default 1); omitted endpoints default to the whole string in that direction, and a negative step reverses direction:

```
>>> cs = "0123456789"
>>> cs[1:4]
'123'
>>> cs[:-1]
'012345678'
>>> cs[::2]
'02468'
>>> cs[::-1]
'9876543210'
```

Immutability

Strings are **immutable** — they cannot be changed in place:

```
>>> cs = "hello"
>>> cs[0] = "H"
TypeError: 'str' object does not support item assignment
```

Strings as booleans

```
>>> bool("Hello")
True
>>> bool("")
False
```

The empty string is "smaller" than every other string under comparison (`"" < "A"` is `True`) — note it is distinct from a single space (`len(" ") == 1`, `bool(" ") == True`).

Tuples

A **tuple** is an immutable ordered collection of elements — elements need not share a type or be distinct. Round brackets `()` construct tuples.

```
>>> xs = (0, 1, 2, 3, 4, 5)
>>> type(xs)
<class 'tuple'>
>>> xs[-1]
5
>>> xs[0:4]
(0, 1, 2, 3)
>>> xs[0] = -1
TypeError: 'tuple' object does not support item assignment
```

Tuples support the same +/scalar-multiplication as strings:

```
>>> xs = (1, 'a', 2, 'b')
>>> ys = (3, 'c', 4, 'd')
>>> xs + ys
(1, 'a', 2, 'b', 3, 'c', 4, 'd')
>>> 2*xs
(1, 'a', 2, 'b', 1, 'a', 2, 'b')
```

Nomenclature

Tuples are named by size: couple (2), triple (3), quadruple (4), quintuple (5), sextuple (6), septuple (7), octuple (8), … an n-tuple in general (`(0, 1, 2, ..., n-1)` isn't valid Python syntax itself — it's just informal notation for the pattern).

Singleton and empty tuples

A single value in round brackets *without* a trailing comma is not a tuple — it's just that value in parentheses. The trailing comma is what makes it a tuple:

```
>>> type((1))
<class 'int'>
>>> type((1,))
<class 'tuple'>
>>> (1,) + (2,)
(1, 2)
>>> (1) + (2,3)   # common error
TypeError: unsupported operand type(s)
```

The empty tuple `()` is falsy:

```
>>> type(())
<class 'tuple'>
>>> () + (1,)
(1,)
>>> bool(())
False
```

Comparing tuples

Tuples compare element-by-element, lexicographically (like strings) — a shorter tuple is less than a longer tuple that extends it:

```
>>> (1, 2, 3) == (1, 2, 3)
True
>>> (1, 2, 3) < (1, 2, 4)
True
>>> (1, 2) < (1, 2, 3)
True
```

Packing and unpacking

Multiple assignment/printing in one line, via an (implicit) tuple:

```
>>> x, y, z = 2, 3, 4
>>> x, y, z
2, 3, 4
```

Python While Loops

A **loop** is a control structure that repeats code that belongs to it. A **while-loop** is a loop that repeats code *while some condition is satisfied*:

```
while <condition>:
    <code>
```

The condition is checked before every repetition (including the first) — if it's false to begin with, the loop body never runs at all:

```
>>> x = 0
>>> while False:
...     print(x)
...     x += 1
>>> # nothing prints
```

Augmented assignment operators

Operator	Equivalent to
`x += y`	`x = x + y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x %= y`	`x = x % y`

`break`

The `break` keyword terminates the (innermost) loop immediately and continues with the rest of the program:

```
>>> while True:
...     print("hello")
...     break
...     print("world")
hello
```

Simulating a do-while

A **do-while** (or **repeat-until**) is a while-loop variant, available in other languages but not natively supported in Python, that runs its code *at least once* before checking the condition:

```
do:
    <code>
while <condition>   # not valid Python syntax
```

We can simulate one with `while True` (some programmers prefer `1` to `True`) and a `break`:

```
>>> x = 0
>>> while 1:
...     x += 1
...     if x > 0:
...         break
>>> x   # x retains its value outside the while
1
```

User input

`x = input()` waits for input from the keyboard and assigns it to `x`, always with `type(x) is str`.

Random number generation

Random number generation is handled by an external library (not built-in), so it must be imported:

```
>>> from random import randint
>>> randint(1, 6)   # chosen uniformly over the interval
2
```

or, importing the entire library:

```
>>> import random
>>> random.randint(1, 6)
2
```