Australian National University

# COMP1730/COMP6730
## Programming for Scientists

# Strings and more on sequences

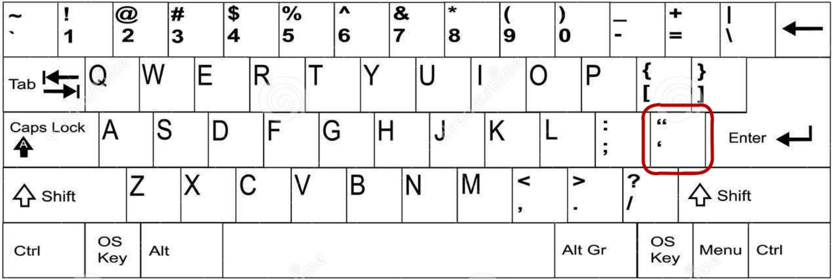
# Lecture outline

- Character encoding & strings
- Indexing, slicing & sequence operations
- Iteration over sequences

# Characters & strings

# Strings

- Strings – values of type `str` in python – are used to store and process text.
- A string is a *sequence* of *characters*.
  - `str` is a sequence type.
- String literals can be written with
  - single quotes, as in `'hello there'`
  - double quotes, as in `"hello there"`
  - triple quotes, as in `'''hello there'''`


~ ` ! 1 @ 2 # 3 $ 4 % 5 ^ 6 & 7 * 8 ( 9 ) 0 _ - + = | \ ←
Tab Q W E R T Y U I O P { [ } ]
Caps Lock A S D F G H J K L : ; “ ‘ Enter
Shift Z X C V B N M < , > . ? / Shift
Ctrl OS Key Alt Alt Gr OS Key Menu Ctrl

⋆ Beware of copy–pasting code from slides (and other PDF files or web pages).

- Quoting characters other than those enclosing a string can be used inside it:

```
>>> "it’s true!"
>>> ’"To be," said he, ...’
```

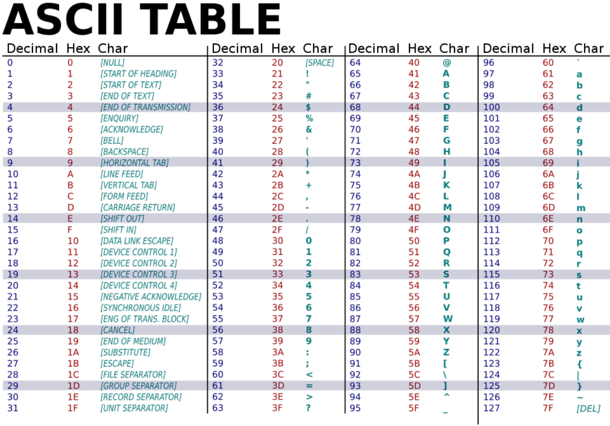
- Quoting characters of the same kind can be used inside a string if escaped by backslash (\):

```
>>> ’it\’s true’
>>> "it’s a \"quote\""
```

- Escapes are used also for some non-printing characters:

```
>>> print("\t1m\t38s\n\t12m\t9s")
```

- Character encoding
- Every character has a number.
- ASCII code (historically most common format for Western text)
- 8-bit code.

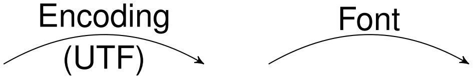


# ASCII TABLE

| Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | [NULL] | 32 | 20 | [SPACE] | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | [START OF HEADING] | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | [START OF TEXT] | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | [END OF TEXT] | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | [END OF TRANSMISSION] | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | [ENQUIRY] | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | [ACKNOWLEDGE] | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | [BELL] | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | [BACKSPACE] | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | [HORIZONTAL TAB] | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | A | [LINE FEED] | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | B | [VERTICAL TAB] | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | C | [FORM FEED] | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | D | [CARRIAGE RETURN] | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | E | [SHIFT OUT] | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | F | [SHIFT IN] | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | [DATA LINK ESCAPE] | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | [DEVICE CONTROL 1] | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | [DEVICE CONTROL 2] | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | [DEVICE CONTROL 3] | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | [DEVICE CONTROL 4] | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | [NEGATIVE ACKNOWLEDGE] | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | [SYNCHRONOUS IDLE] | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | [ENG OF TRANS. BLOCK] | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | [CANCEL] | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | [END OF MEDIUM] | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | [SUBSTITUTE] | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | [ESCAPE] | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | [FILE SEPARATOR] | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | [GROUP SEPARATOR] | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | [RECORD SEPARATOR] | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | [UNIT SEPARATOR] | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | [DEL] |

# Unicode, encoding and font

- *Unicode* defines numbers ("*code points*") for $>$140,000 characters (in a space for $>$1 million).



Encoding
(UTF)

Font

| Byte(s) | Code point | Glyph |
|---|---|---|
| 0100 0101 | 69 | EEE$\mathcal{E}$ |
| 1110 0010<br>1000 0010<br>1010 1100 | 8364 | €€€€ |

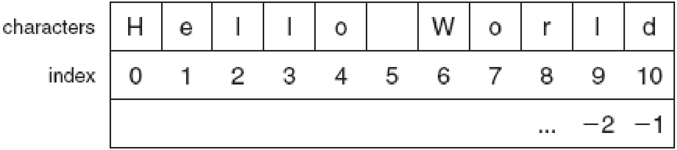
- python 3 uses the unicode character representation for all strings.
- Functions `ord` and `chr` map between the character and integer representation:

```
>>> ord(’A’)
>>> chr(65 + 4)
>>> chr(32)
>>> chr(8364)
>>> chr(20986)+chr(21475)
>>> ord(’3’)
```

- See `unicode.org/charts/`.

# More about sequences

# Indexing & length (reminder)



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| characters | H | e | l | l | o | | W | o | r | l | d |
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| | | | | | | | | | ... | −2 | −1 |

FIGURE 4.1 The index values for the string `'Hello World'`.

Image from Punch & Enbody

- In python, all sequences are indexed from 0.
- ...or from end, starting with -1.
- The index must be an integer.
- The length of a sequence is the number of elements, *not* the index of the last element.

- len(*sequence*) returns sequence length.
- Sequence elements are accessed by placing the index in square brackets, `[]`.

```
>>> s = "Hello World"
>>> s[1]
'e'
>>> s[-1]
'd'
>>> len(s)
11
>>> s[11]
```

IndexError: string index out of range

# Slicing

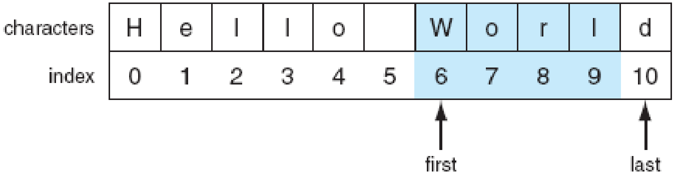
- Slicing returns a subsequence:

  `s[start:end]`

  - `start` is the index of the first element in the subsequence.
  - `end` is the index of the first element after the end of the subsequence.
- Slicing works on all built-in sequence types (`list`, `str`, `tuple`) and returns the same type.
- If `start` or `end` are left out, they default to the beginning and end (i.e., after the last element).

- The slice range is “half-open”: start index is included, end index is one after last included element.

```
>>> s = "Hello World"
>>> s[6:10]
'Worl'
```



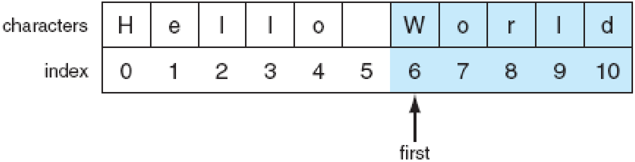
| characters | H | e | l | l | o |  | W | o | r | l | d |
|---|---|---|---|---|---|---|---|---|---|---|---|
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

FIGURE 4.2 Indexing subsequences with slicing.

Image from Punch & Enbody

- The end index defaults to the end of the sequence.

```
>>> s = "Hello World"
>>> s[6:]
'World'
```

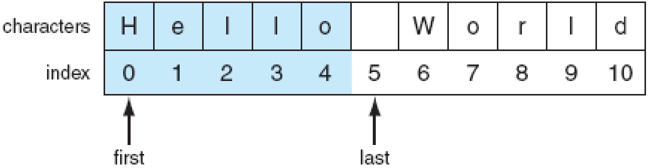


| characters | H | e | l | l | o |  | W | o | r | l | d |
|---|---|---|---|---|---|---|---|---|---|---|---|
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Image from Punch & Enbody

- The start index defaults to the beginning of the sequence.

```
>>> s = "Hello World"
>>> s[:5]
'Hello'
```



| characters | H | e | l | l | o |  | W | o | r | l | d |
|---|---|---|---|---|---|---|---|---|---|---|---|
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Image from Punch & Enbody

```
>>> s = "Hello World"
>>> s[9:1]
’’
>>> s[-100:5]
’Hello’
```

- An empty slice (index range) returns an empty sequence
- Slice indices can go past the start/end of the sequence without raising an error.

# Operations on sequences

- Reminder: *value types determine the meaning of operators applied to them*.
- Concatenation: `seq + seq`

```
>>> "comp" + "1730"
```

- Repetition: `seq * int`

```
>>> "Oi! " * 3
```

- Membership: `value in seq`
  - *Note:* `str in str` tests for substring.
- Equality: `seq == seq`, `seq != seq`.
- Comparison (same type): `seq < seq`, `seq <= seq`, `seq > seq`, `seq >= seq`.

# Sequence comparisons

- Two sequences are equal if they have the same length and equal elements in every position.
- `seq1 < seq2` if (lexicographic ordering).
  - `seq1[i] < seq2[i]` for some index *i* and the elements in each position before *i* are equal; or
  - `seq1` is a prefix of `seq2`

# String comparisons

- Each character corresponds to an integer.

```
ord(’ ’) == 32
ord(’A’) == 65
ord(’Z’) == 90
ord(’a’) == 97
ord(’z’) == 122
```

- Character comparisons are based on this.

```
>>> "the ANU" < "The anu"
>>> "the ANU" < "the anu"
>>> "nontrivial" < "non trivial"
```

# Iteration over sequences

# The `for .. in ..` statement

```
for name in expression:
    # body of for
    statement1
    statement2
    ...
```

1. Evaluate the expression, to obtain an iterable collection.
   - If value is not iterable: TypeError.
2. For each element *E* in the collection:

2.1 assign `name` the value *E*;

2.2 execute the loop block.

```python
for char in "The quick brown fox":
    print(char, "is", ord(char))
```

vs.

```python
s = "The quick brown fox"
i = 0
while i < len(s):
    char = s[i]
    print(char, "is", ord(char))
    i = i + 1
```

# Iteration over sequences

- Sequences are an instance of the general concept of an *iterable* data type.
  - An iterable type is defined by supporting the `iter()` function.
  - python also has data types that are iterable but not indexable (for example, sets and files).
- The `for .. in ..` statement works on any iterable data type.
  - On sequences, the `for` loop iterates through the elements *in order*.

# String methods

# Methods

- Methods (or member functions) are only functions with a slightly different call syntax:

```python
"Hello World".find("o")
```

instead of

```python
find("Hello World", "o")
```

- methods have an implicit first parameter ”self”
- This will be clearer when we study classes.
- python’s built-in types, like `str` or `list`, have many useful methods.
  - `help(str)` (or press tab after s. where s is a string)

# String useful functions and methods

| Operation | Returns |
|---|---|
| `str()` | Returns an empty string |
| `str(obj)` | Printable representation of obj |
| `str1.isalpha()` | True if str1 is not empty and all characters are alphabetic |
| `str1.numeric()` | True if str1 is not empty and all characters are numeric |
| `str1.isupper()` | True if string contains at least one “cased” character and all “cased” characters are upper case, else False |
| `str1.startswith(str2[,startpos, [endpos]])` | Returns true if str1 starts with str2 |
| `str1.find(str2[,startpos, [endpos]])` | Returns lowest index at which str2 is found, else returns -1 |
| `str1.count(str2[,startpos, [endpos]])` | Returns the number of occurrences of str2 in str1 |
| `str1.upper()` | Returns a string with all of its characters as uppercase |

# List useful methods

| Operation | Returns |
|---|---|
| `del lst[n]` | Remove the nth element form lst |
| `del lst[i:j]` | Remove ith through jth element of lst |
| `del lst[i:j:k]` | Remove every kth element of from i up to j from lst |
| `lst.append(x)` | Add x to end of lst |
| `lst.extend(x)` | Add elements of x to lst |
| `lst.insert(i, x)` | Insert x before the ith element of lst |
| `lst.remove(x)` | Remove the first occurrence of x from lst |
| `lst.pop([i])` | Remove ith element of lst; if i is not specified, remove the last element |
| `lst.reverse` | Reverse the list |
| `lst.sort([reverseflag[,keyfn])` | Sort the list by comparing elements.<br>If keyfn is provided, then comparison is done based on it.<br>If reverse flag is True, then reverse sort is performed. |

# Useful functions for all collection types

| Operation | Returns |
|---|---|
| `x in coll` | True if coll **contains** x |
| `x not in coll` | True if coll **does not** contain x |
| `any(coll)` | True if **any** item in coll is true, otherwise false |
| `all(coll)` | True if **every** item in coll is true, otherwise false |
| `len(coll)` | The **number of items** in coll (not supported by streams) |
| `max(coll, key=function)` | **Maximum item** in coll which may not be empty |
| `min(coll, key=function)` | **Minimum item** in coll which may not be empty |
| `sort(coll[, keyfn][, reverseflag])` | A list containing the **elements** of coll, **sorted by comparing** elements |

# Programming problem

- Find a longest repeated substring in a word:
  - 'backpack' → 'ack'
  - 'singing' → 'ing'
  - 'independent' → 'nde'
  - 'philosophically’ → 'phi'
  - 'monotone' → 'on'
  - 'wherever' → 'er'
  - 'repeated' → 'e'
  - 'programming' → 'r' (or 'g', 'm')
  - 'problem' → ''

# Take home message

- Python stores strings using unicode.
- `for` loop to iterate over elements of sequence or any iterable collection.